

Cours



VB.net

Introduction.

[Visitez le Site LDF:](#)

et téléchargez Ldf logiciel shareware de **compta**, un **dictionnaire médical gratuit** pour Word, la page de [Médiwin...](#)

[Qui est l'auteur](#)



26453

Visiteurs

Il s'agit d'un cours de **Visual Basic.Net © de Microsoft**, [.net](#) pour débutants ou anciens voulant passer à la version .net . C'est la somme des connaissances que j'aurais aimées trouver à mes débuts et la manière dont j'aurais aimée qu'on m'explique les choses (Au départ c'était mes notes puis petit à petit...).

Visual Basic.Net apporte une puissance inégalée et nécessite une rigueur importante mais il devient vite complexe et technique. La documentation et les livres sont totalement hermétiques pour les novices et rebutent totalement les débutants. Les articles sur le Web sont très techniques et traitent d'emblée de problèmes complexes, ils sont nécessaires mais pas pour le débutant. J'explique donc dans ce cours, à ma manière, très simplement, comment créer un programme afin de permettre un bon démarrage même à celui qui n'a jamais fait d'informatique. (Je traite des programmes Windows:Windows Forms et pas ASP Web).

J'encourage par ce cours sans prétention, l'amateur à développer ses propres programmes.

Soyez un utilisateur actif :

Retournez les bugs et erreurs, et même les fautes d'orthographe.

Envoyez moi vos critiques et remarques.

Adressez moi vos idées, du code original, des infos à mettre sur le site.

Ou simplement indiquez moi que vous avez lu mon cours, cela fait toujours plaisir et m'incite à poursuivre.

Merci à ceux qui m'envoient un petit mot, et à ceux qui me donnent un coup de main.

[Chargez la version PDF contenant la totalité du cours.](#)

3.3Mo V1 du 06/12/2004 334 pages (nécessite Acrobat Reader). Merci à
Alexandre Freire pour son aide

OU

[Allez au 'Sommaire' de la version On line](#)

Questions à l'auteur: Avant de poser une question, cherchez s'il n'y a pas la réponse sur le site. Si je connais la réponse et qu'elle est didactique, je la mets sur le site et je l'indique à l'auteur. Je ne peux pas répondre à des questions très particulières et spécifiques car je n'ai pas d'expérience poussée dans tous les aspects de VB et les questions sont nombreuses.

Ne pas hésiter à chercher la réponse à vos problèmes sur le site <http://developpez.com> qui est très sérieux, complet et didactique.(nombreux didacticiels)



Developpez.com

des centaines de **tutoriels** sur tous les domaines
à consulter ou à télécharger **gratuitement**



**Voir les nouvelles pages ou modifications qui ne sont pas dans
le pdf:**

[Variables 'object' Soyons Strict](#)

[Démarrez le
Cours.](#)

ou

[Allez au
Sommaire](#)



[Envoyer un mail à P. LASSERRE](#)



Accueil Site



Qui est l'auteur du site.

✘ **Le Dr LASSERRE Philippe** est médecin généraliste exerçant en groupe dans le Rhône (à Toussieu), il développe des logiciels depuis des années..:

✘ Il a travaillé avec des ordinateurs:

ZX81, New-Brain, Vic20, Apple II, puis PC avec l'aide de Bill.

✘ Il utilise le Visual Basic de Microsoft ® VB6 et VB.Net, a fait un peu d'assembleur Z80 il y a longtemps.

✘ Il a fait partie de **MEDITRA**, association de médecins informatisés du Rhône pionnière en la matière à l'époque, il a été co-fondateur d'un club d'informatique local (Microzon) ou on programmait dur!!.

✘ Ensuite il a écrit des logiciels, pour cela outre le côté technique informatique , il a beaucoup travaillé sur le dossier médical informatisé, les plans de soins.

✘ Plein d'idées et de projets, un seul problème: il n'y a que 24h dans une journée.

Auteur de:

CREEMED, *Il y a quelques années.. C'était un utilitaire pour Medigest ® Dos .*

MEDIWIN® distribué par [Polytel](#) , logiciel de gestion complète de cabinet médical dont il est le co-auteur.

Logiciel agréé **Sesam-Vitale**, très complet, innovant, incluant les notions de "**dossier vivant**", "**procédures de soins**" (contenu médical validé par des thèses), travaillant avec la base de médicament **BCB** de Résip©, contenant le **dictionnaire de la SFMG**.

LDF logiciel de comptabilité en ShareWare

Vous pouvez [Envoyer un mail au Dr LASSERRE](#)



Site **LDF** Cours **VB.net**



0.0	Qu'allons nous faire ?	Suivant →	← Sommaire
------------	-------------------------------	------------------	-------------------

Qu'allons nous étudier?

Quel plan de cours suivrons nous?

Quels logiciels utiliser?

Quelle configuration est nécessaire?

Qu'allons-nous étudier?

Ce cours est un cours de VisualBasic.Net

Nous étudierons principalement: LES APPLICATIONS WINDOWS.



Application Windows



Bibliothèque de classes



Bibliothèque de contrôles Wi...

Les applications Windows utilisent les **WindowsForms**.

Les applications Windows sont des **programmes directement**

exécutables qui utilisent des **fenêtres Windows**: des programmes de traitement de texte, d'image, de musique, des jeux, de petits utilitaires, des logiciels métiers (médicaux)...

Nous laisserons de côté **les applications 'Web' (en ASP qui utilisent les WebForms) et qui permettent de créer des sites Internet**, les applications 'console'..



PLAN DU COURS :

Nous étudierons donc comment créer une application Windows :

On étudiera la notion d'**objet**, d'**évènement**. (Section 0.)

On étudiera le langage **Visual Basic**.net. (Section 1.)

On utilisera les objets '**fenêtre**' et les '**divers contrôles**' pour créer l'**interface utilisateur** (Section 3.).

On découvrira la manière de **créer une application**. (Section 4.)

On apprendra à **créer une classe** (Section 5.)

On verra comment utiliser les bases de données. (Section 6.)

voir le [Sommaire](#) du cours. On peut s'aider de l'[Index](#) pour rechercher un mot clé.

Conseil de lecture : Lire la succession de pages en cliquant sur le **bouton Suivant** en bas de page. **Ne pas hésiter à relire plusieurs fois les chapitres.**

Quels logiciels utiliser?

Il y a plusieurs manières de faire du VB.NET:

q Acheter **Visual Studio.Net 2003**

de Microsoft, il contient une interface de développement (IDE) (programme permettant d'écrire un logiciel et de créer un programme exécutable) Il comporte: VB.Net mais aussi C#.Net. Il existe des versions d'initiation', 'professionnelle', 'entreprise', 'architect'.



Ce cours utilise **Visual Studio.Net 2003**

Nous allons apprendre à utiliser **Visual Basic .NET** version 7.1 2002 avec le Framework 1.1 de 2002.

C'est très cher. Il est possible de faire un [essai gratuit de 60 jours](#) de VisualStudio.Net 2003.

q Tester **Visual Studio Express**

2005 de Microsoft, **Béta gratuite** basée sur la bêta du Framework 2.0

Dans Visual Studio il y a **Visual Basic 2005** qui est la futur version VB. La version Express est une version allégée pour débutant. Gratuite?, La Bêta est gratuite elle. Nouveau Framework, avec nouvelle Class. Elle ne contient que le VB .(il existe VisualWeb Express

par ailleurs). Dans Visual Studio il y a aussi Visual C#..

[Téléchargement gratuit](#)

q L'alternative gratuite: **SharpEditor**

Installer un logiciel de développement gratuit:

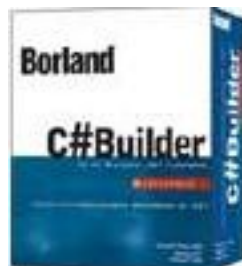


SharpEditor

par exemple, [SharpDevelop](#) le configurer pour qu'il marche en VB (il supporte VB et C#).

[voir l'"IDE #develop" ou la télécharger? comment l'installer et l'utiliser?](#)

q Utiliser **C#Builder** de Borland.



C# Builder est l'IDE .NET de Borland. L'édition personnelle de C# builder est entièrement gratuite mais limitée à des développements non commerciaux en VB ou C#. Il existe des versions payantes plus puissantes.

[Télécharger la version personnelle gratuite \(usage non professionnel\)](#)

Un produit à part:

WebMatrix de
Microsoft, il est
gratuit:

Cet outil
permet de
développer des
applications
WEB (des pages
Internet) en
ASP.NET, en C#
et **VB.NET**. Vous
trouverez des
informations
supplémentaires
[ici](#).

Si j'ai bien
compris: dans
une page HTML,
l'ASP gère les
contrôles, le
code des
procédures peut
être en VB.

**Il ne permet
pas d'utiliser
des WebForms et
d'écrire des
applications
Windows.**

ASP.NET Web
Matrix
nécessite le
Microsoft
.NET
Framework,
1.1 et

Site
tourne sur
Windows
Server 2003,
Windows 2000,
et Windows XP
operating
systems.

[Télécharger et
installer
WebMatrix](#)



Help : Avez-vous utilisé WebMatrix C#Builder ? Merci de votre expérience.

Quel logiciel choisir?

Lire le comparatif [C#Builder versus VisualStudio](#)

[Que faut-il pour développer avec Visual Basic.net?](#)

Pour développer avec Visual Studio 2003 il faut Windows XP ou 2000 avec au minimum **256 Mo de mémoire vive. Un grand écran (vu le nombre de fenêtres) est conseillé.**

Les exécutables fonctionnent sous Windows 98, XP, 2000.



Retour



Index



Sommaire



Suivant



Cours VB.NET



[Démarrer le cours](#)

[Home
Page](#)



[Qui est l'auteur ?](#)



[Lui envoyer
un mail](#)

Sommaire.

[Introduction , page de présentation.](#)

Introduction: débutez ici

0-0 [Qu'allons nous étudier? Avec quoi développer en VB.Net? Configuration nécessaire?](#)

0-1 [Les 'Objets.'](#)

Principe et structure des programmes.

1-1 [Programmation événementielle: le premier programme.](#)

1-2 [Les instructions, les procédures . Les 'Sub' , les 'Function'](#)

1-2-2 [Les modules](#)

1-d [Environnement de développement de Visual Studio](#)

1d bis [Environnement de développement de SharpDevelop
\(alternative gratuite\)](#)

Le langage Visual Basic

1-3 [Introduction](#)

1-4 [Les 'Algorithmes'.](#)

1-5 [L'affectation.](#)

1-6 [Les variables.](#)

1-6.1 [Variables 'String' et 'Char'](#)

1-6.2 [Variables Numériques](#)

1-6.3 [Conversion](#)

1-6.4 [Les 'Tableaux'](#)

1-6.5 [Les 'Collections'](#)

- 1-6.6 [Les 'Structures'](#)
- 1-6.7 [Attention au type valeur ou référence](#)
- 1-6.8 [Variable 'Object' et autre](#)

- 1-7 [Soyons strict et explicite.](#)
- 1-8 [Les constantes.](#)
- 1-9 [Surcharge.](#)
- 1-10 [Les opérateurs.](#)
- 1-11 [Les structures de contrôle.](#)
- 1-12 [Revenons sur les procédures et leurs paramètres.](#)
- 1-13 [Portée des variables.](#)
- 1-14 [Les nombres aléatoires](#)
- 1-15 [La 'Récursivité'](#)

- 1-19 [Faut-il oublier le GOTO](#)

- 1-20 **[Espace de noms, Classes, Objet](#)**

Exemple de petites routines.

- E 1-1 [Petites routines d'exemples très simple](#)
- E 1-2 [Petits programmes mathématiques](#)
- E 1-3 [Tri et recherche dichotomique](#)
- E 1-4 [Calculs financiers simples](#)

L'interface utilisateur.

- 3-1 [Introduction.](#)
- 3-2 [Les feuilles ou 'Formulaires'.](#)
- 3-3 [Les 'Boutons'.](#)
- 3-4 [Les 'TextBox'](#)
- 3-5 [Les 'Labels'](#)
- 3-6 [Les cases à cocher.](#)
- 3-7 [Les 'Listes'.](#)
- 3-8 [Boites toutes faites.](#)
- 3-9 [Regroupement de contrôles. \(comment créer des **groupes de contrôles**\).](#)
- 3-10 [Dimensions, position des contrôles.](#)
- 3-11 [Main Menu, ContextMenu](#)
- 3-12 [Avoir le Focus](#)
- 3-13 [Barre de bouton , barre de status.](#)
- 3-14 [Les images](#)

Résumons, révision pour y voir plus clair:

- 3-30 **[Structuration des programmes+++ Exemple détaillé: Calcul de l'IMC++](#)**

3-31 [Ordre des instructions dans un module: résumé.](#)

Exemple de petits programmes.

E 3-1 [Conversion F/](#) (Une fenêtre)

E 3-2 [Calcul mensualités d'un prêt.](#) (les fonctions financières de VB)

Pour faire un vrai programme il faut savoir:

4-1 [Démarrer ou arrêter un programme. Procédure Main\(\).](#)

4-2 [Ouvrir une autre fenêtre.](#)

4-3 [Traiter les erreurs.](#)

4-4 [Créer une fenêtre 'multi document'.](#)

4-5 [Travailler sur les dates, les heures, sur le temps.](#)

4-6 [Lire et écrire dans les fichiers \(séquentiels ou aléatoires\)](#)

4-7 [Travailler sur les répertoires](#)

4-8 [Afficher correctement du texte](#)

4-9 [Modifier le curseur](#)

4-10 [Lancer une autre application, afficher une page Web](#)

4-11 [Imprimer](#)

4-12 [Dessiner](#)

4-13 [Faire une aide pour l'utilisateur](#)

4-14 [Appeler un API](#)

4-15 [Faire du glisser-déplacer \(Drag&Drop\)](#)

4-20 [Débogage \(ou comment rechercher les 'Bugs'\)](#)

Pour diffuser le programme, il faut:

D-1 [Comprendre le FrameWork](#)

D-2 [Distribuer l' application](#)

Autres exemples

E 4-1 [Horloge numérique](#)

E 4-2

Création de Classes, composant, Programmation objet

5-1 [Programmation orientée objet](#)

5-2 [Créer une Classe](#)

5-3 [Créer un composant \(Bibliothèque de Classe et de Contrôles\)](#)

5-4 [Les interfaces](#)

5-10 [Programation 'fonctionnelle' ou 'objet'?](#)

Un gros morceau: les bases de données

- 6-1 [Notion sur les bases de données](#)
- 6-2 [Généralités sur ADO.NET](#)
- 6-3 [Syntaxe SQL](#)
- 6-4 [Lire rapidement en lecture seule: le DataReader](#)
- 6-5 [Travailler sur un groupe de données: le DataSet](#)
- 6-6 [Contrôles \(DataGrid, ListBox..\) et base de données.](#)

Migration VB6=>VB.NET

- 7-1 [Différences entre VB6 et VB.net; Migration.](#)

Règles de bonne programmation Optimisation.

- 7-2 [Règles de bonne programmation.](#)
- 7-3 [VB.net est-il rapide? Optimiser le code en vitesse.](#)
- 7-4 [Chronométrer du code](#)

Allons plus loin (Mais pas de problèmes, vous êtes des bons!!!)

- 8-1 [Portée des procédures](#)
- 8-2 [Comprendre le code généré par VB](#)
- 8-3 [Créer des contrôles par code.](#)



Developpez.com
des centaines de **tutoriels** sur tous les domaines
à consulter ou à télécharger **gratuitement**



[Démarrer le Cours.](#)

[Envoyer un mail à P. LASSERRE](#)



Site: **LDF**



Site du Dr Lasserre
Philippe:
Programmation pour
tous et informatique du
médecin.



.net

LDF Le Livre de Frais

	Recettes par	Recette
J	Solde Cpt Pro	Emprunts Cheque



Cours de
**Visual
Basic.Net**

Apprenez à
programmer.

Découvrir et télécharger le
**logiciel de
comptabilité**
LDF distribué en
Shareware.



La maladie d'Alzheimer
Avec ataxie et ptosys.

**Dictionnaire
médical** gratuit pour
correcteur d'orthographe
Word.

plus de 15 000 mots.



La page de
Médiwin: Trucs,
astuces pour les
utilisateurs de ce logiciel
de cabinet médical.



Qui est l'auteur de ce site ?

Site **LDF** Cours **VB.net**



0.1	Les objets		
		Suisvant	Sommaire

VB utilise la notion d' 'OBJET' .

Pour bien comprendre ce qu' est un objet, nous allons prendre des exemples dans la vie courante puis nous passerons à des exemples dans Visual Basic.

Ma voiture est un **objet**, cet objet existe, on peut l' utiliser.

Ma voiture fait partie des 'voitures', du type, du genre 'voiture'. Les 'voitures' c' est une classe d' objet (**Class**) qui a ses caractéristiques : c' est en métal, ça roule en transportant des passagers... mais je ne peux pas utiliser 'les voitures'.

De manière générale, une **classe est une représentation abstraite** de quelque chose, tandis qu' un **objet est un exemple utilisable** de ce que représente la classe.

Pour fabriquer ma voiture, je prends les caractéristiques de la **class** 'voitures' (c' est comme un moule) et je fabrique (j' **instance**) une voiture, je la nomme '**MaVoiture**'.

`Dim MaVoiture As New voitures`

Déclare `MaVoiture` comme un nouvel objet de type 'voitures'



Class --> Objet

Type 'voitures'--> Objet 'Mavoiture'

Propriétés (Attributs) :



Prenons `MaVoiture`.

Elle a des **propriétés** : une marque, une couleur, une puissance...

Pour indiquer la couleur de ma voiture on utilise la notation :

`MaVoiture.couleur`

Syntaxe : `Objet.Propriété` (Il y a **un point** entre les 2 mots)

Pour modifier la couleur et avoir une voiture verte on écrit :

`MaVoiture.couleur= "Vert"`

Et la voiture devient verte !!



`MaVoiture.Phares.Avant` indique les phares avant de la voiture.

`MaVoiture.Phares.Avant.Allumé` indique l'état des phares (Allumé ou non)

Si je fais :

`MaVoiture.Phares.Avant.Allumé=True` (Vrai) cela allume les phares.

Méthodes :

`MaVoiture` fait des choses : elle roule par exemple.

Pour faire rouler la voiture j'appelle la **méthode** 'Roule'

`MaVoiture.Roule`

Si c'est possible pour cette méthode je peux indiquer à quelle vitesse la voiture doit rouler :

`MaVoiture.Roule(100)` 'j'ai ajouté un **paramètre**.

Le paramètre est un renseignement envoyé avec la méthode.

Il est possible parfois d'indiquer en plus si la voiture doit rouler en marche avant ou en marche arrière.

`MaVoiture.Roule(10, Arriere)`

Il y a donc 2 manières d'appeler la méthode Roule : avec 1 ou 2 paramètres, on dit que la méthode est **surchargée**; chaque manière d'appeler la méthode s'appelle '**signature**'.

Evènement :

Des évènements peuvent survenir sur un objet.

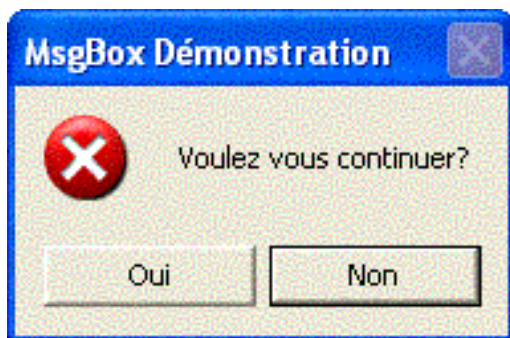
`MaVoiture_démarre` est un évènement, quand la voiture se met en route (si par exemple j'ai fait `MaVoiture.Roule(10, Arriere)`), cet évènement `démarre` se déclenche automatiquement.

Tous cela ce sont des images, mais voila, **vous avez compris ce qu'est un objet !**

Et dans Visual Basic.net:

Une application Windows se compose de **fenêtres** (nommées aussi **formulaires**) dans lesquelles se trouvent des **contrôles** (bouton, liste, texte...)

Exemple de fenêtre avec 2 boutons, une zone de texte (un label) et une icône:



Dans une application Windows, il y a aussi des lignes de code utilisant des variables pour faire des calculs.

En Visual Basic.net tout est objet :

les fenêtres (on dit les formulaires),

les variables,

les contrôles (les boutons, liste, image, case à cocher..)

...

Il faut un **moule** pour faire un objet. Le moule c'est une **classe**.

Le moule va servir à créer un objet, on dit **une instance**.

On peut créer, instancer une multitude d'objets avec le même moule.

Pour créer, démouler un objet, on utilise les mots clé **Dim** et **As New**.

Dim objet **As New** Classe

New est un **constructeur**.

Exemple : créer une fenêtre (un formulaire) :

Je dessine une fenêtre **FormDémarrage** (c'est la Classe, le moule)

puis

Dim F As New FormDémarrage

crée une fenêtre qui se nomme 'F' à partir du moule, du modèle (FormDémarrage) que j'ai dessiné.

Autre exemple :

```
Dim B as New Buttons
```

'Créer un bouton avec les attributs habituels des boutons (**Class** Button)

Troisième exemple:

Comment créer une variable nommée Mavariabale pouvant contenir un entier (Integer)

```
Dim MaVariable As New Integer
```

```
Dim MaVariable As Integer 'est correct aussi
```

Ici, pour une variable, on remarque que New peut être omis

Tout objet a des propriétés.

On utilise la syntaxe : `Objet.Propriété` (Il y a un **point** entre les 2 mots)

`F.BackColor` indique la couleur de fond de la fenêtre.

S'il y a un bouton, la couleur de fond du bouton sera :

```
Bouton.BackColor
```

Ou

```
F.Bouton.BackColor
```

Noter la syntaxe : La couleur du bouton qui est dans la fenêtre F

Comment modifier cette propriété?

`Bouton.BackColor=Red` 'modifie la couleur de fond du bouton



Autre exemple:

La propriété **Visible**: si elle a la valeur `True` (Vraie) l'objet est visible si elle est à `False` l'objet n'est pas visible.

`Bouton.Visible=False` 'fait disparaître le bouton

<==Ici il y a un bouton invisible!! oui, oui!!

Les objets ont des méthodes parfois.

Prenons un exemple simplifié.

Les **Listes** (liste déroulante) ont une méthode **Clear** qui permet de les vider.

Si je veux vider toutes les lignes d'une liste nommé `Liste1`, je fais:

`Liste1.Clear` 'Le concept est exact mais l'exemple est un peu simplifié!



Les propriétés et méthodes se nomment **les membres**

d'un objet.

Certains objets ont des évènements:

Reprenons notre bouton. Quand l'utilisateur click dessus, l'évènement `Bouton_Click` survient.

Ce sont les objets **contrôles** (bouton, case à cocher..)et les formulaires qui ont des évènements.

Interface et implémentation:

Ce que je vois de l'objet, c'est son **interface** (le nom des propriétés, méthodes..) exemple: la méthode `Clear` fait partie de l'interface d'une `ListBox`. Par contre le code qui effectue la méthode (celui qui efface physiquement toutes les lignes de la `ListBox`), ce code se nomme **implémentation**, lui n'est ni visible ni accessible.

Visibilité:

Quand un objet est créé, il est **visible et utilisable**, uniquement dans la partie du programme où il a été défini.

Par exemple habituellement, je peux voir et modifier la couleur d'un bouton uniquement dans le code de la fenêtre ou il est situé.

Pour les variables on parle de **portée**: la variable peut être locale (**Private**) ou de portée générale ('**Public**') visible partout.

En résumé :

En Visual Basic.net **tout est objet**.

Les **Classes** sont des types d'objet.

Pour créer un objet à partir d'une Classe, il faut utiliser les mots clé `Dim` `..As` `New`:

```
Dim Objet As New Class
```

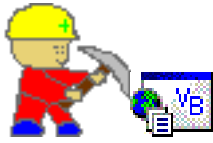
Un objet a :

Des **propriétés**.

Des **méthodes**.

Des **évènements**.





Cours VB.net

[Index.](#)  [Sommaire](#) [Home](#)
[Page](#)

LDF

A.

- [Affectation](#)
- [Abs](#)
- [Ado](#)
- [Add \(collections\)](#)
- [Addhadler](#)
- [Anchor](#)
- [AppActivate](#)
- [Arrêt programme](#)
- [Array](#)
- [ArrayList](#)
- [Asc](#)
- [AscW](#)
- [Assignment](#)
- [Attributes](#)

B.

- [BackColor](#)
- [BinarySearch](#)
- [Bottom](#)
- [Button_Click](#)
- [By Val By Ref](#)

C.

- [Catch](#)
- [CBool](#)

[CByte](#)
[CChar](#)
[CDate](#)
[CDBl](#)
[CDec](#)
[Check](#)
[CheckBox](#)
[CheckChanged](#)
[CheckedListBox](#)
[Ceiling](#)
[Chaîne de longueur fixe](#) [Chaîne de longueur fixe\(fichier\)](#)
[Char](#)
[ChDir](#)
[ChrW](#)
[CInt](#)
[Clear](#)
[Clone](#)
[Close](#)
[CLng](#)
[Code managé](#)
[CObj](#)
[Configuration nécessaire pour faire du VB](#)
[Combo](#)
[Compare](#)
[Concat](#)
[Configuration minimale pour faire du VB.Net](#)
[Const](#)
[Constructeur \(structures\)](#)
[Constructeur\(Classe\)](#)
[ContainsKey](#)
[ControleBox](#)
[Copy](#) [Copy\(fichier\)](#)
[Cos](#)
[Couleurs \(constantes\)](#)
[Collections](#)
[CultureInfo \(classe\)](#)
[Create](#)
[CreateDirectory](#)
[CreateInstance](#)
[CShort](#)
[CSng](#)
[CStr](#)
[CType](#)
[Cursor \(classe\)](#)
[CurrentDirectory](#)

D.[DataReader](#)[DataSet](#)[DateTime](#)[Debug \(classe\)](#)[Délégué](#)[Dequeue](#)[DialogResult](#)[Dir](#)[Directory Directory \(classe\)](#)[DirectoryInfo \(classe\)](#)[DirectoryName](#)[Dispose](#)[Dock](#)[Do Loop](#)**E.**[Enabled](#)[Encapsulation](#)[EndTrim](#)[EnQueue](#)[Enum](#)[EOF](#)[Espace de nom](#)[Evènement](#)[Events](#)[Environment \(classe\)](#)[Exceptions](#)[Exist](#)**F.**[Fenêtre de démarrage](#)[Fenêtre MDI](#)[Fenêtre Splash](#)[File](#)[FileClose](#)[FileCopy](#)[FileGet](#)[FileInfo](#)[FileLen](#)[FileOpen](#)[FilePut](#)[Finally](#)[Fix](#)[Flag](#)[FlatStyle](#)

[Floor](#)

[Flush](#)

[Focus](#)

[For_Each](#)

[ForeColor](#)

[Form_Activate](#)

[Format](#)

[Form_Load](#)

[FormBorderStyle](#)

[For_Next](#)

[Font](#)

[Fonctions financières](#)

[Framework](#)

[Friend](#)

[Function](#)

G.

[Get\(classe\)](#)

[GetChar](#)

[GetCurrentDirectory](#)

[GetDirectories](#)

[GetFile](#)

[GetParent](#)

[GetProcessByName](#)

[GetProcesses](#)

[GetType](#)

[GetValue](#)

[GetUpperBound](#)

[GroupBox](#)

[Groupe de contrôle \(comment remplacer le groupes de contrôle de VB6\)](#)

H.

[Handles](#)

[HashTable](#)

[Height](#)

[Héritage](#)

[Heure](#)

I.

[Icon](#)

[If Then](#)

[Iif](#)

[IndexOf](#) [IndexOf \(tableau\)](#)

[ImageList](#)

[Imports](#)

[Implementation](#)

[InputBox](#)

[Int](#)
[Interface](#)
[Instr](#)
[InstRev](#)
[Is](#)
[IsNumeric](#)
[IsReference](#)
[Item](#)
[Items](#)

L.

[Label](#)
[LCase](#)
[Len](#)
[Left](#) [Left\(contrôle\)](#)
[like](#)
[LineInput](#)
[LinkArea](#)
[LinkClicked](#)
[Linklabel](#)
[ListBox](#)
[ListView](#)
[Location](#)
[LOF](#)
[Ltrim](#)
[Length](#)
[LSet](#)

M.

[Main\(\) Sub](#)
[MainMenu](#)
[Math Espace de noms](#)
[Max](#)
[MaxLenght](#)
[MaximizeBox](#)
[MaxValue](#)
[MDI](#)
[MessageBox](#)
[Méthodes](#)
[Mid](#)
[Min](#)
[MinimizeBox](#)
[MinValue](#)
[MkDir](#)
[Mod](#)
[Modale \(fenêtre\)](#)

[Module Standard](#)

[Move\(fichier\)](#)

[MsgBox](#)

[Multiline](#)

N.

[Name](#)

[Now](#)

O.

[On_error](#)

[OpenFileDialog](#)

[Open](#)

[Option Compare](#)

[Option Explicit](#)

[Option Strict](#)

[Overloads](#)

[Overrides Overrides](#)

[Owner](#)

P.

[Panel](#)

[ParamArray](#)

[Par Valeur Par Référence By Val By Ref](#)

[Parse](#)

[Path \(classe\)](#)

[Peek](#)

[PictureBox PictureBox](#)

[Poke](#)

[Polymorphisme](#)

[Pow](#)

[Private](#)

[Print](#)

[Property](#)

[Propriétés](#)

[Process\(classe\)](#)

[Protected](#)

[Public](#)

[Push](#)

Q.

[Queue](#)

R.

[RadioButton](#)

[Read](#)

[Recherche dichotomique](#)

[Récursivité](#)

A vertical toolbar on the left side of the page contains various icons for Visual Basic 6.0, including symbols for zooming, undo, redo, and other standard editing functions.

[Références](#)

[Refresh](#)

[Rename](#)

[Replace](#)

[Return](#)

[Reverse](#)

[Round](#)

[Right Right\(contrôle\)](#)

[RSet](#)

[RichTextBox](#)

[RTF](#)

[Rtrim](#)

S

[SaveFileDialog](#)

[StartPosition](#)

[ScrollBars](#)

[Select](#)

[Select Case](#)

[Sender](#)

[Set\(classe\)](#)

[SetValue](#)

[Shell](#)

[Signature](#)

[Sin](#)

[Sign](#)

[Sort _Sorted_\(ListBox\)](#)

[SortedList](#)

[Space](#)

[Split](#)

[Splitter](#)

[Sqrt](#)

[Stack](#)

[StartTrim](#)

[Statique\(classes\)](#)

[StatusBar](#)

[StrCom](#)

[StrDup](#)

[Stream](#)

[StreamReader](#)

[StringCollection](#)

[StrRevers](#)

[Str](#)

[Structure](#)

[Sub](#)

[Sub Main\(\)](#)

[SubString](#)



[Surcharge](#)

[Switch](#)

T.

[TabControl](#)

[TabStop](#)

[Ticks](#)

[TimeSpan](#)

[ToDay](#)

[ToolBar](#)

[Tag](#)

[Text](#)

[TextAlign](#)

[ToLower](#)

[Top](#)

[TopMost](#)

[ToString](#) [ToString](#)

[ToUpper](#)

[Trace \(classe\)](#)

[TransparentColor](#)

[TreeState](#)

[Tri \(routine de\)](#)

[Try](#)

[Type primitif](#)

[Trim](#)

U.

[Ubound](#)

[Unicode](#)

[Ucase](#)

V.

[Val](#)

[View](#)

[Visible](#)

W.

[Width](#)

[WindowState](#)

[With](#)

[WithEvents](#)

[Write](#)

[WriteLine](#)

[Démarrer le Cours.](#)



Site **LDF** Cours **VB.net**



1.1	Les évènements		
------------	-----------------------	--	--

Nous allons comprendre la programmation événementielle: Comment fonctionne Visual Basic:

- Ce que voit l'utilisateur.
- Ce qu'a fait le développeur pour arriver à ce résultat.

Principes de la programmation VB

Le programmeur va dessiner l'interface utilisateur (fenêtre, bouton, liste..), il va ensuite uniquement écrire les actions à effectuer quand certains événements se produisent sur cette interface.

C'est **Visual Basic** qui va entièrement s'occuper de la gestion des événements.

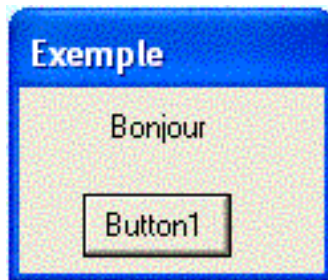
Exemple :le premier programme:

Il affiche 'Bonjour' quand on clique sur un bouton.

C'est pas original: le premier programme, dans tous les cours d'informatique, permet d'afficher 'Bonjour' (ou 'Hello Word').

q **Que voit l'utilisateur du programme?**

L'utilisateur final, celui qui utilise le logiciel, voit une **fenêtre** avec un **bouton**, Si il appuie sur ce bouton il voit s'afficher « **Bonjour** ».



q **Que se passe t-il dans le programme?**

Quand l'utilisateur clique sur le bouton cela déclenche automatiquement **un événement**. (**Button1_Click**), cet évènement contient du code qui affiche « Bonjour ».

q **Que doit faire le programmeur pour arriver à ce résultat?**

Pour atteindre ce résultat, **le programmeur va dessiner la fenêtre, le bouton, la zone d'affichage du texte (un label)** puis il va simplement indiquer dans l'évènement **Button_Click d'afficher « Bonjour »**.



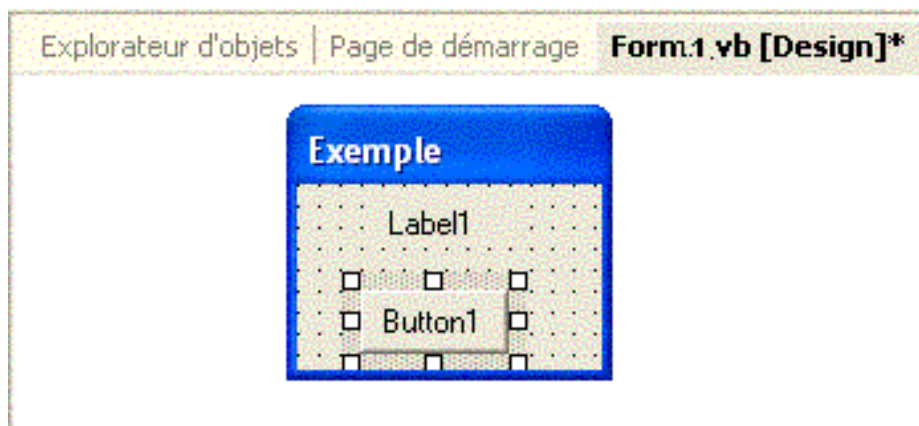
Le fait de déterminer la procédure à appeler ou de réaliser l'appel est entièrement pris en charge par VB.

En pratique, que fait le programmeur:

Le programmeur est en **mode 'conception'** (ou mode **Design**): Il écrit le programme :

A- Il dessine l'interface utilisateur

(Ce que verra l'utilisateur final, c'est **l'interface utilisateur** : une fenêtre avec des boutons, des listes, du texte..) :

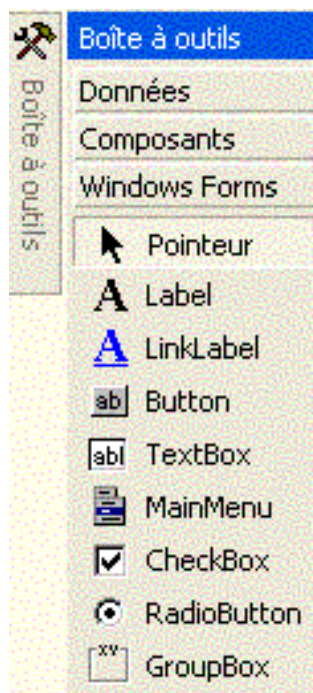


Il crée une fenêtre :

Menu **Projet**, **Ajouter un formulaire Windows**, cliquer sur **Windows Form**, une fenêtre 'Form1' apparaît.

Il ajoute un bouton :

Pour cela il utilise la Boite à outils:



Il clique sur 'Boite à Outils' à gauche , bouton Windows Forms, puis bouton '**Button**', il clique dans Form2, déplace le curseur sans lâcher le bouton, puis lâche le bouton de la souris : le dessin d'un bouton apparaît.

Pour l'exemple, Il **ajoute un label**.

Un label est un contrôle qui permet d'afficher un texte.

Comme pour le bouton il clique sur 'Boite à Outils' à gauche, bouton Windows Forms, bouton 'Label' et met un contrôle label sur la fenêtre.

B- Il va écrire le code correspondant aux événements :

Il double-clique sur le bouton qu'il a dessiné :

Une fenêtre de conception de code s'ouvre et il apparaît :

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

```
End Sub
```

Cela correspond à la **procédure** (entre Sub et End Sub) **évènement** qui, quand le programme fonctionne, est automatiquement déclenchée quand l'utilisateur du logiciel clique sur le bouton1.

Une procédure est un ensemble de lignes de code qui commence par Sub et se termine par End Sub (ou Function..End Function).

Comment indiquer dans cette procédure d'afficher "Bonjour"?

Le label possède une propriété nommée '.text' qui contient le texte à afficher.

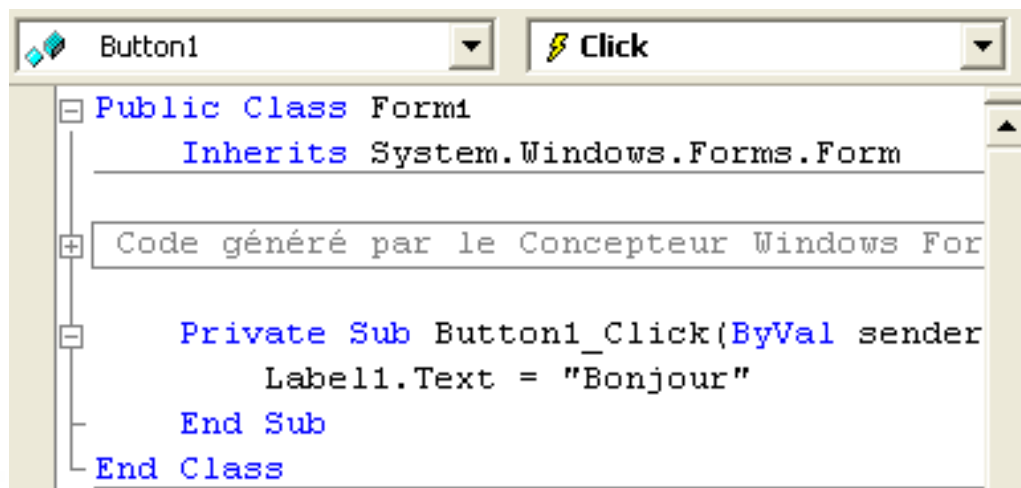
Il faut taper le code qui modifie cette propriété '.text' , qui y met la chaîne de caractère "Bonjour":

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

```
Label1.Text = "Bonjour"
```

```
End Sub
```

cela donne:



```
Button1 Click
Public Class Form1
    Inherits System.Windows.Forms.Form
    Code généré par le Concepteur Windows For
    Private Sub Button1_Click(ByVal sender
        Label1.Text = "Bonjour"
    End Sub
End Class
```

Voilà votre premier programme est écrit.

Comment exécuter ce programme?

Il est possible de tester immédiatement le programme en mode débogage, sans quitter l'environnement de développement:

Utiliser le menu 'Déboguer' puis 'Démarrer' qui lance l'exécution du programme.

On peut aussi taper sur **F5** pour lancer le programme.

Ou plus simplement cliquer sur la flèche:



C'est plus rapide, **lancer l'exécution avec le premier bouton**, le second servant à arrêter temporairement l'exécution, le troisième à terminer l'exécution.

Quand le programme est totalement écrit, terminé, testé, il est possible de le **compiler** et ainsi de créer un **fichier exécutable** (possédant une extension '.exe') qui fonctionne de manière autonome en dehors de l'environnement de développement.

C'est ce fichier exécutable qui est fourni à l'utilisateur.

Par opposition le code écrit par le programmeur, composé d'instructions Visual Basic, se nomme **le code source**.

En mode exécution :

L'utilisateur voit bien une fenêtre avec un bouton, s'il clique dessus, « Bonjour » s'affiche.

En résumé :

Le programmeur utilise des outils de dessin pour construire une **interface utilisateur** : des fenêtres avec des contrôles dessus : menus, boutons, case à cocher..

VB, pour chaque fenêtre ou pour chaque contrôle, génère une liste d'**événements**, (Evènement lié au chargement d'une fenêtre, évènement lié au fait de cliquer sur un bouton, évènement survenant quand on modifie un texte...)

Il suffit, dans la **procédure événement** qui nous intéresse, d'**écrire le code** qui doit être effectué lorsque cet événement survient.

Comme nous l'avons vu le code sert à agir sur l'interface (Afficher un texte, ouvrir une fenêtre, remplir une liste, un tableau), mais il peut aussi effectuer des calculs, évaluer des conditions et prendre des décisions, travailler en boucle de manière répétitive et ainsi effectuer les tâches nécessaires .



 Retour

 Index

 Sommaire

Suivant 

Site **LDF** Cours **VB.net**



	<h2>Les 1.2 instructions, les procédures</h2>		
--	---	--	--

Qu'est ce qu'une instruction , une procédure?

Quelle différences entre les procédures:

liées aux évènement?

Les procédures non liées?

Les 'Sub', les 'Fonctions'.

Les instructions:

Une instruction est le texte permettant d'effectuer une opération, une déclaration, une définition.

`Dim A As Integer` est une instruction (de déclaration)

`A=1` est aussi une instruction qui effectue une opération.

C'est habituellement une 'ligne de code' 'exécutable'.

Une instruction est exécutée lorsque le programme marche.

Plusieurs instructions peuvent se suivre sur une même ligne, séparées par ':'

`Dim B As String : B="Bonjour"`

Quand un programme tourne, les instructions sont effectuées ligne par ligne.

Pour mettre **des commentaires** dans un programme, on le fait précéder de ':

'Ceci est un commentaire, ce n'est pas une instruction.

Le commentaire ne sera pas exécuté.

Les procédures:

Une **procédure** est un ensemble d'instructions, de lignes de code, un groupement d'instructions bien définies effectuant une tâche précise.

Les procédures sont bien délimitées:

Il y en a de 2 sortes:

Les procédures Sub:

Elles débutent par le mot **Sub** et se terminent par **End Sub**.

Les procédures Function:

Elles débutent par **Function** et se terminent par **End Function**.

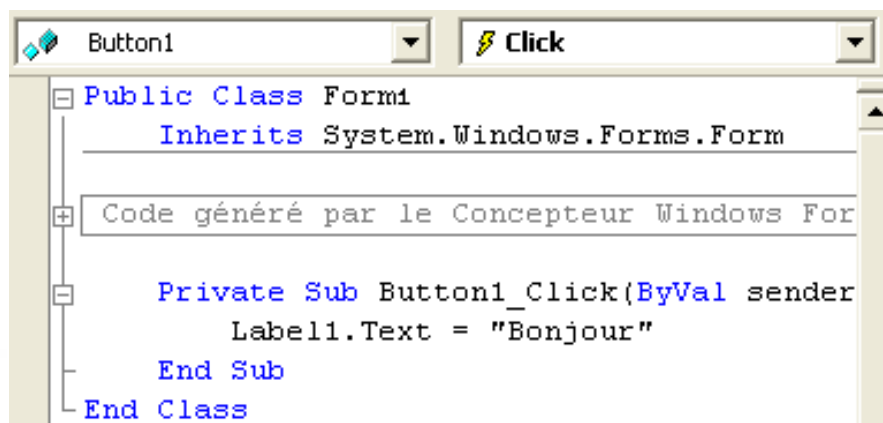
Exemple:

```
Sub Maprocédure
```

```
    A=1
```

```
End Sub
```

Exemple concret d'une procédure: la procédure Button_Click du premier programme. (Celui qui affiche 'Bonjour'; elle ne contient qu'une ligne de code. Le mot Sub est précédé de Private, on verra plus loin ce que cela signifie.



Vous avez vu que l'on peut dessiner l'interface, une fenêtre Form1 par exemple. En mode conception, après avoir dessiné l'interface, on doit avoir accès aux procédures.



Si on double-clique sur la fenêtre, on a accès aux procédures évènement liées à cette fenêtre, si on double-clique sur un objet (bouton, case à cocher... on voit apparaître les procédures évènement de ce contrôle.

```

Button1 Click
Public Class Form1
    Inherits System.Windows.Forms.Form
    Code généré par le Concepteur Windows For
    Private Sub Button1_Click(ByVal sender
        Label1.Text = "Bonjour"
    End Sub
End Class
  
```

Quand on voit ces procédures, on peut y inclure du code.

Nous allons voir qu'il y a 2 types de procédures: les procédures liées aux évènements et celles qui ne sont pas liées.

Procédures liées aux évènements.

Si on double clique sur le fond d'une fenêtre, (Celle du programme 'Bonjour') on voit apparaître les procédures liées à cette fenêtre et aux contrôles contenus dans cette fenêtre :

```

Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim a As String
#Region " Code généré par le Concepteur Windows Form "
Private Sub Form1_Load(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles MyBase.Load
    Label1.Text = « «
End Sub
Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
    Label1.Text = "Bonjour"
End Sub
End Class
  
```

Détaillons:

On voit 3 parties:

Public Class Form1

Ce n'est pas une procédure mais la définition de la fenêtre.

La fenêtre fait partie des `Windows.Forms.Form..`

Ces lignes sont générées automatiquement par VB.

Si vous déroulez cette partie, en cliquant sur le petit +, vous pouvez lire le code permettant de créer la fenêtre, les contrôles.. C'est généré automatiquement par VB. (le chapitre [8-2](#) explique en détails le code généré par VB, mais c'est complexe pour les débutants pour le moment!!)

▫ `Private Sub Form1_Load`

Chaque fenêtre a une procédure `Form_Load` qui est exécutée lorsque la fenêtre est chargée, on y met généralement le code initialisant la feuille.

Il y a bien d'autres procédures liées à la fenêtre :

Dérouler la liste box en haut à gauche de la fenêtre de code, cliquer sur (**Form1 events**), si vous déroulez maintenant la liste à droite vous aurez **tous les événements qui génèrent une procédure** :

<code>Load</code>	Lors du chargement de la fenêtre
<code>Unload</code>	Lors du déchargement de la fenêtre
<code>Activated</code>	Lorsque la fenêtre devient active
<code>GotFocus</code>	Lorsque la fenêtre prend le focus
<code>Resize</code>	Lorsque la fenêtre est redimensionnée

...

4 Private Sub Button1_Click

C'est la **procédure liée au bouton** et qui contient le code à effectuer quand l'utilisateur clique sur le bouton.

C'est là que l'on écrit le code qui doit s'effectuer lorsque l'utilisateur clique sur le bouton.

De la même manière que pour la fenêtre, vous pouvez voir dans la liste en haut, tous les événements liés aux boutons qui génèrent une procédure :

Click Lorsque l'utilisateur clique sur le bouton.

DoubleClick Lorsque l'utilisateur double-clique sur le bouton.

MouseDown 'se déclenche si appui du bouton de la souris

MouseUp 'se déclenche si relâchement du bouton de la souris

...



On voit donc que le formulaire (la fenêtre) et tous les contrôles d'une application ont chacun des procédures pour chaque événement qui peut survenir.

Procédures non liées:

Parfois on a besoin de code qui fait une tâche particulière, qui est utilisé à plusieurs endroits et qui **n'est pas liée à un événement**.

On crée dans ce cas **une procédure indépendante des événements**.

Le système des procédures permet aussi de découper un problème complexe en quelques fonctions moins complexes et indépendantes les unes des autres.

Ces procédures sont en fait des **sous-programmes** : si une ligne appelle une procédure, le programme 'saute' au début de la procédure, il effectue le code de la procédure puis revient juste après la ligne qui avait appelé la procédure et continue les lignes suivantes.

Exemple : plusieurs fois dans le programme j'ai besoin de calculer la surface d'un cercle à partir de son rayon et de l'afficher sur un label.

Plutôt que de retaper dans chaque procédure le code, je peux créer une procédure 'Sub' nommée `AfficheSurfaceCercle`.

Il suffit ensuite si nécessaire d'appeler la procédure qui effectue le calcul et affiche le résultat puis revient effectuer le code situé après l'appel.

Comment appeler une procédure?

Par `Call NomdeProcedure()` ou par

```
NomdeProcedure()
```

Call est facultatif.

Noter les parenthèses après le nom de la procédure.

Procédures Sub :

Comment créer cette procédure Sub?

Dans la fenêtre de code, tapez :

Sub `AfficheSurfaceCercle` puis validez. Vous obtenez :

```
Sub AfficheSurfaceCercle()
```

```
End sub
```

Le code de la procédure est compris entre le `Sub` et le `End Sub`.

Pour que le calcul se fasse, il faut fournir, (transmettre de la procédure qui appelle à la procédure Sub) la valeur du rayon.

Pour indiquer que la Sub doit recevoir un **paramètre** (un **argument** en VB) ajouter entre les parenthèses :

```
Sub AfficheSurfaceCercle( Rayon as Single)
```

Cela signifie qu'il existe une procédure qui reçoit comme paramètre une variable de type Single (Réel simple précision) contenant le Rayon.

Ajouter le code :

```
Label.text =(3.14*Rayon*Rayon).ToString
```

Que fait cette ligne ?

Elle fait le calcul: '3.14*Rayon*Rayon' ('*' signifie multiplier), on transforme le résultat en chaîne de caractères (grâce à '.ToString') que l'on met dans la propriété .text du label : Cela affiche le résultat. (On verra toute cette syntaxe en détail ultérieurement)

On obtient:

```
Sub AfficheSurfaceCercle( Rayon as Single)

    Label.text =(3.14*Rayon*Rayon).ToString

End sub
```

Comment appeler cette Sub?

N'importe quelle procédure pourra **appeler la Sub** AfficheSurfaceCercle en envoyant la valeur du rayon afin d'afficher la surface du cercle dans un label.

Exemple d'appel pour un rayon de 12 :

```
AfficheSurfaceCercle(12)
```

Affiche dans le label: 452.16

Procédures 'Function' :

Parfois on a besoin que la procédure **retourne un résultat, qu'elle donne en retour un résultat à la procédure appelante.** . Dans ce cas on utilise une **Fonction.**

Exemple: je veux créer une fonction à qui je fournis un rayon et avoir en retour la surface d'un cercle.

Comment créer cette Fonction?

Tapez `Function SurfaceCercle` puis validez, ajouter (`Rayon As Single`)

Tapez `Return 3.14*Rayon*Rayon`

Ce que la fonction doit retourner est après `Return` (ce que la procédure doit renvoyer à la procédure appelante.)

On obtient la fonction complète:

```
Function SurfaceCercle( Rayon as Single)
```

```
    Return 3.14*Rayon*Rayon
```

```
End Function
```

Comment appeler cette Fonction?

Dans la procédure qui appelle, il faut une variable pour récupérer la valeur retourner par la Fonction:

```
S= NomdeLaFonction()
```

N'importe quelle procédure pourra appeler la fonction et obtenir le résultat dans la variable S par exemple pour un rayon de 12 :

```
Dim S As Single
```

```
S=SurfaceCercle(12)
```

On appelle la fonction `SurfaceCercle` en envoyant le paramètre '12', ce qui fait que à l'entrée de la fonction, `Rayon=12`, le calcul est effectué et le résultat du calcul (452.16) est retourné grâce à `Return`. S récupère ce résultat.

Après l'appel de cette fonction, S est égal à 452.16

Il est possible de **spécifier le type retourné par la fonction**:

```
Function SurfaceCercle( Rayon as Single) As Single
```

`As Single` en fin de ligne après () indique que la fonction retourne un Single. Il faut donc que la variable qui reçoit la valeur retournée (S dans notre exemple) soit aussi un Single.

Il existe une **autre manière de retourner le résultat d'une fonction**, reprenons l'exemple précédent, on peut écrire:

```
Function SurfaceCercle( Rayon as Single)
```

```
    SurfaceCercle= 3.14*Rayon*Rayon
```

```
    Exit Function
```

```
End Function
```

Ici on utilise le nom de la fonction pour retourner le résultat , avec un signe '='.

Utilisez plutôt la méthode Return.

`Exit Function` permet aussi de sortir de la fonction, cela a le même effet que Return sauf que Return peut être suivi d'un argument de retour (et pas Exit Function).

Module standard :

La sub **AfficheSurfaceCercle** affiche le résultat dans le formulaire où elle est située.

Par contre la fonction **SurfaceCercle** est d'intérêt général, n'importe quelle procédure doit pouvoir l'appeler, de plus elle n'intervient pas sur les contrôles des formulaires et n'est donc pas liée aux formulaires.

On la placera donc dans un **module standard** qui est un module du programme qui ne contient que du code. (Pas d'interface utilisateur)

Pour créer un module standard Menu Projet>Ajouter un module.

Y mettre les procédures.

Private Public :

Avant le mot **Sub** ou **Function** on peut ajouter :

Private indiquant que la procédure est accessible uniquement dans le module.

C'est donc une procédure **privée**.

Les procédures liées aux évènements d'une feuille sont privées par défaut.

Public indiquant que la procédure est accessible à partir de toute l'application.

S'il n'y a rien devant sub la procédure est publique

Remarques :

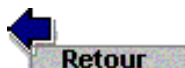
1- Pour sortir d'une procédure Sub avant la fin utiliser **Exit Sub** (**Exit Function** pour une fonction).

2- Quand vous appelez une procédure, il faut toujours mettre des parenthèses même s'il n'y a pas de paramètres.

```
FrmSplash.ShowDialog ()
```

Eventuellement on peut faire précéder l'appel du mot clé **Call**, mais ce n'est pas obligatoire.

```
Call FrmSplash.ShowDialog ()
```



site **LDF** Cours : **VB.net**

1.2.2	Les modules	 	Suivant	Sommaire
--------------	------------------------	---	-------------------------	--------------------------

Qu'est ce qu'un module?

On a vu qu'un programme est décomposé en **modules**, chaque module contenant **des procédures**.

Chaque module correspond physiquement à un fichier '.vb'.

Il existe

- les modules des formulaires.
- les modules standards.
- les modules de 'Classe'.

Un programme Visual Basic comporte donc :

« Les 'Modules de Formulaires':

contenant:

Le dessin des fenêtres de l'interface utilisateur (ou formulaire) contenant les **contrôles (boutons, listes, zones de texte, cases à cocher...)**

Le code qui comprend :

Les procédures liées aux événements de la feuille (Button_Click..)

Les procédures indépendantes des événements mais qui interviennent sur l'interface. Ce sont des Sub() ou des Function().

¶ **Les modules standards.**

Ils servent de stockage de procédures. Procédures "d'intérêt général".

Ces procédures sont des Sub() ou des Function() qui peuvent être appelées à partir de n'importe quel endroit (pourvu qu'elles soient 'Public').

Ils peuvent aussi servir à déclarer les objets ou déclarer les variables 'Public' qui seront utilisées donc accessibles par la totalité du programme.

¶ **les modules de Classe**

Ils ont vocation à fabriquer des objets, on verra cela plus loin (chapitre 5).

Il y a deux manières de travailler en VB.NET:

En programmation 'Fonctionnelle':

Chaque problème est décomposé en '**Fonctions**' (Les Subs et Fonctions).

Ces fonctions sont stockées **dans des modules standards** (ou dans les modules de formulaire).

C'est cette approche qui est privilégiée jusqu'aux chapitres 4-..

Voir l'exemple ci-dessous.

En programmation 'Objet':

On le verra dans les chapitres 5-...: on crée ses propres objets dans des **modules de Classe**, on utilise les membres de ces objets pour programmer.

La mode est à la programmation Objet!!

Voir le chapitre 5-10 pour approfondir ces notions.

Comment créer un module standard:

Faire Menu Projet>Ajouter un module. Donner un nom au module. C'est Module1.vb par défaut.

On remarque que le module est bien enregistré dans un fichier .vb

Un module standard ne contient que du code.

Comment ajouter une Sub dans un module Standard?

Taper Sub Calcul puis valider, cela donne:

```
Sub Calcul
```

```
End Sub
```

Exemple d'utilisation de procédures et de modules:

Créons un petit programme exemple:

L'utilisateur saisit un nombre puis il clique sur un bouton; cela affiche **le carré** de ce nombre:

Il faut créer l'interface utilisateur: créer une fenêtre (Form1), y mettre un bouton (nommé Button1), une zone de texte (Text1) permettant de saisir un nombre, un label (label1) permettant l'affichage du résultat.

Créer un module standard (Module1) pour y mettre les procédures communes.

On observera uniquement l'agencement des procédures et non leur contenu. Pour un programme d'une telle complexité, la structure aurait pu être plus simple, mais l'intérêt de ce qui suit est didactique.

On décompose le programme en tâches plus simples: En particulier une procédure sert au calcul, une sert à l'affichage.

La procédure `CalculCarré` calcule le carré.

La procédure `AfficheCarre` affiche le résultat dans le label.

La procédure `Button1_Click` (qui est déclenchée par le Click de l'utilisateur):

 Lit le chiffre tapé par l'utilisateur dans la zone texte.

 Appelle la procédure `CalculCarré` pour calculer le carré.

 Appelle la procédure `AfficheCarré` pour afficher le résultat.

Où sont placées les procédures?

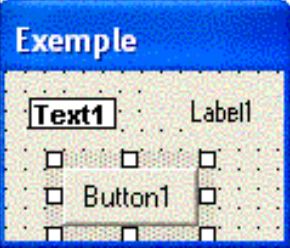
La procédure `Button1_Click` est automatiquement dans le module du formulaire, Form1 (elle est liée au contrôle Bouton1) elle est créée automatiquement quand on crée le bouton.

La procédure `AfficheCarré` est créée **dans le module du formulaire** (Form1) car elle agit sur le contrôle Label1 de ce formulaire.

La procédure `CalculCarré` est créée **dans le module Standard** (Module1) car elle doit être appellable de n'importe où; elle est d'ailleurs 'Public' pour cette raison. Elle n'agit sur aucune fenêtre, aucun contrôle, elle est 'd'intérêt général', c'est pour cela qu'on la met dans un module standard.

Module de Formulaire:Form1.vb

Exemple



```

Sub Button1_Click
    Dim Carre As Single
    Carre=CalculCarre(Val(Text1.Text))
    AfficheCarre(Carre)
End Sub

Sub AfficheCarre(C As Single)
    Label1.Text=C.ToString
End Sub

```

Module Standard: Module1.vb

```

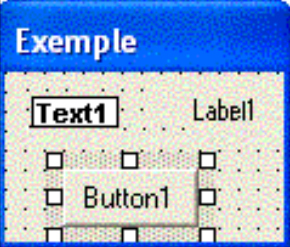
Public FunctionCalculCarre(R As Single)
    Return R*R
End Function

```

Voyons le cheminement du programme:

Module de Formulaire:Form1.vb

Exemple



```

Sub Button1_Click
    Dim Carre As Single
    Carre=CalculCarre(Val(Text1.Text))
    AfficheCarre(Carre)
End Sub

Sub AfficheCarre(C As Single)
    Label1.Text=C.ToString
End Sub

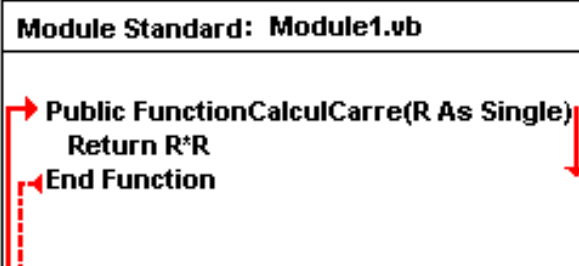
```

Module Standard: Module1.vb

```

Public FunctionCalculCarre(R As Single)
    Return R*R
End Function

```



Quand l'utilisateur clique sur le bouton la **Sub Button1_Click** démarre.

Elle appelle **CalculCarre**.

CalculCarre calcule le carré et renvoie la valeur de ce carré.

La **Sub Button1_Click** appelle ensuite **AfficheCarre** qui affiche le résultat.

On remarque:

On appelle la **Function CalculCarre** par

`Carre= CalculCarre(Valeur)`

On envoie un paramètre Single , la fonction retourne dans la variable Carre, la valeur du carré.

Par contre la Sub AfficheCarre reçoit un paramètre, et ne retourne rien puisque c'est une Sub!!



Site **LDF** :

**Cours
VB.net**



1.D

**L'environnement
de développement.
de Visual
Basic.net**

C'est l'IDE: Environnement de développement intégré de Visual Studio de M

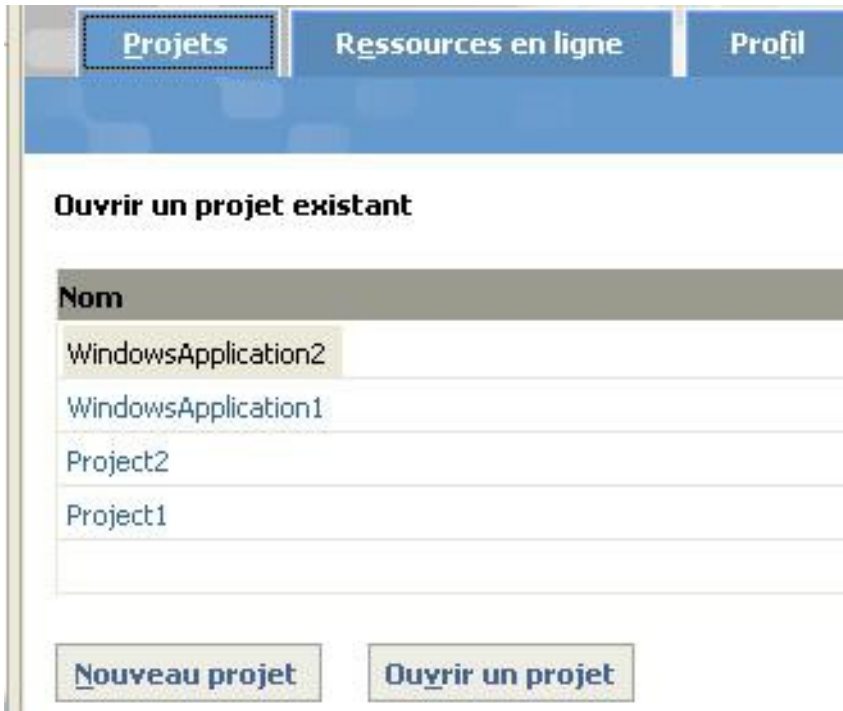
Fenêtre Projet.

Quand on lance VB.net, le logiciel présente une **fenêtre Projets** qui permet

d'**ouvrir un projet existant**

ou

de **créer un nouveau projet**:



Pour un projet Visual Basic normal, il faudra choisir dans les projets **Windows** '.



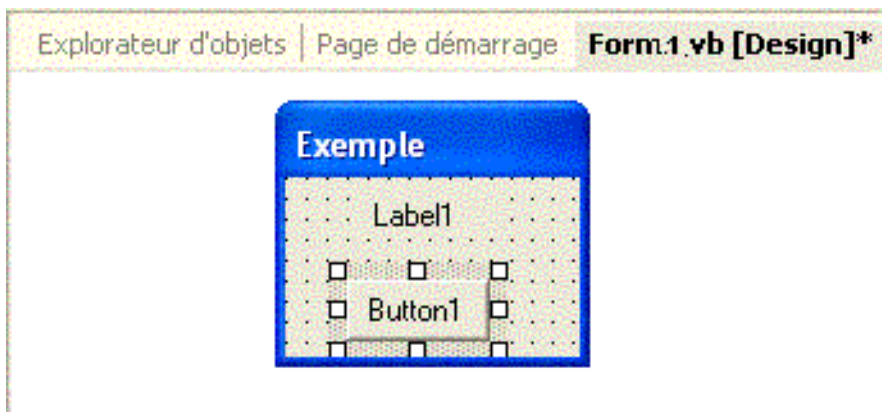
Puis il faut donner un **nom** au projet, modifier si nécessaire le **chemin de projet** qui est par défaut ' C:\Documents and Settings\Nom Utilisateur\Mes Studio Projects' enfin valider sur 'Ok'.

Dans un nouveau projet, créer une fenêtre :

Pour ajouter un fenêtre (un formulaire) Menu **Projet, Ajoute Windows,**

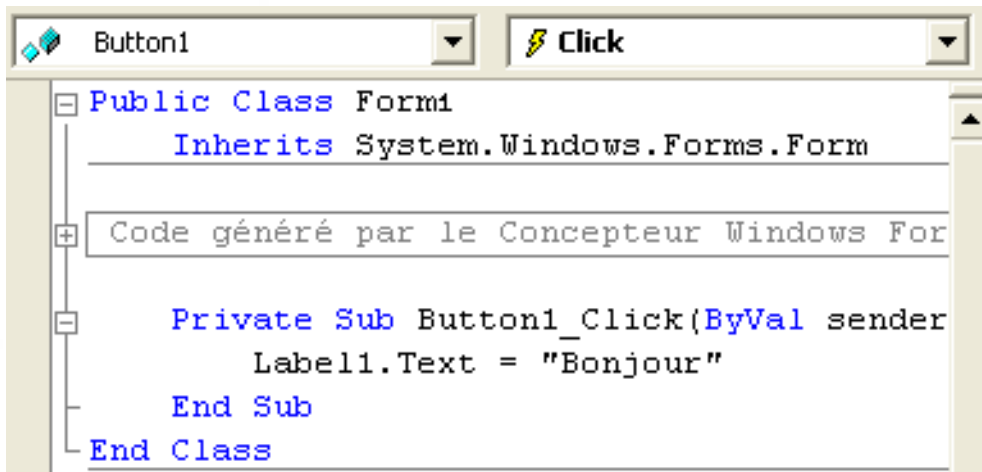
cliquer sur **Windows Form,** une fenêtre 'Form1' apparaît ('Form1' seconde feuille).

La zone de travail se trouve au centre de l'écran: C'est l'onglet **Form1.vb** qui donne donc accès au dessin de la feuille (du formulaire); on peut ajouter et modifier la taille de ces contrôles..



Voir les procédures :

L'onglet **Form1.vb** donne accès aux procédures liées à Form1.



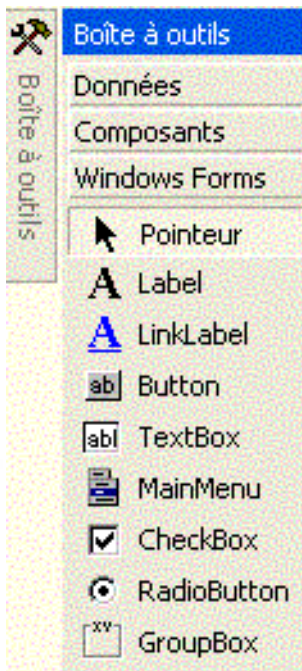
```
Button1 Click
Public Class Form1
    Inherits System.Windows.Forms.Form
    Code généré par le Concepteur Windows For
    Private Sub Button1_Click(ByVal sender
        Label1.Text = "Bonjour"
    End Sub
End Class
```

La liste déroulante de gauche donne la **liste des objets**, celle de droite, **correspondants**.

Il est possible en double-cliquant dans le formulaire ou un contrôle de s dans le code de la procédure correspondant à cet objet.

Ajouter des contrôles à la feuille

Ajouter un bouton par exemple:



Cliquer sur '**Boîte à Outils**' à gauche de l'interface, puis sur '**Button**', cliquer sur le bouton dans la boîte à outils, déplacer le curseur sans lâcher le bouton :



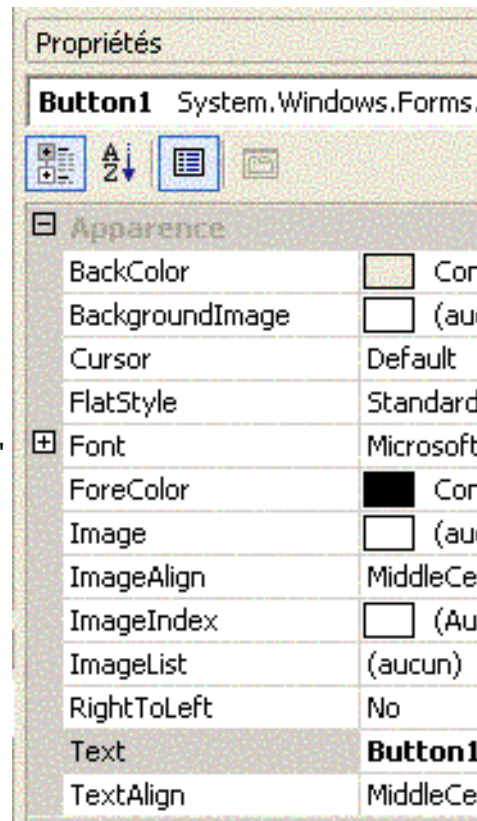
Un bouton apparaît.

Modifier les propriétés d'un contrôle ou du formulaire

Quand un formulaire ou un contrôle est sélectionné dans la fenêtre Design, **ses propriétés sont accessibles dans la fenêtre de propriétés à droite en bas:**

Ici ce sont les propriétés du contrôle 'Button1' (BackColor, Image, Texte..)

(on peut les modifier directement.)



Voir tous les composants d'un projet:

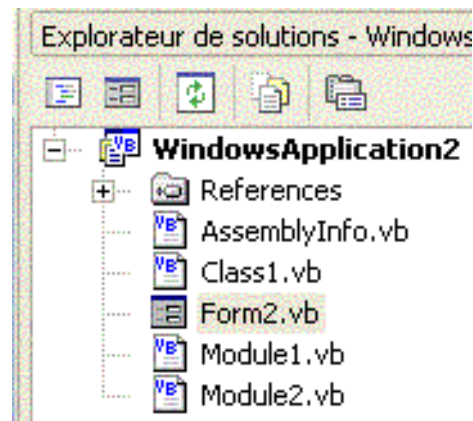
Pour cela il faut utiliser **L'explorateur de solution** en haut à droite, il permet de voir et d'avoir accès au contenu du projet:

Form2.vb qui est un formulaire (une fenêtre).

Module1.vb qui est un module standard.

Références qui contient les espaces de nom.


AssemblyInfo: info nécessaire pour créer un installateur.



Il suffit de cliquer sur la ligne module1.vb dans l'explorateur de solution pour que le module et son code apparaissent dans la fenêtre principale.

Si on clique sur un espace de noms dans la liste **Références**, cela ouvre la fenêtre **d'objet** qui montre **l'arborescence des Classes** et une description sommaire.

Tester son logiciel:

On peut tester le projet grâce à : **lancer l'exécution avec le premier bouton**, le **second** servant à arrêter temporairement l'exécution (mode 'Debug'), le **troisième** pour arrêter définitivement l'exécution (Retour au mode 'Design' ou 'Conception').

Quand on est en arrêt temporaire en mode 'Debug', la ligne courante, celle qui est en cours d'exécution, est en jaune:

```
For i=0 To 100
  Label1.Text=i.ToString
Next i
```

Si on tape la touche **F10** (exécution pas à pas), la ligne `Label1.Text=i.ToString` la position courante passe à la ligne en dessous.

```
For i=0 To 100
  Label1.Text=i.ToString
Next i
```

La **sauvegarde** du projet se fait comme dans tous les logiciels en cliquant de disquettes.

Projet et solutions.

Dans la terminologie VB, un **projet** est une application en cours de développement.

Une **solution** regroupe un ou plusieurs projets (C'est un groupe de projets).

Fichiers, Chemins des sources.

Si vous regardez les fichiers correspondant à un projet VB, les extensions

.vbproj est le fichier de projet.

.sln est le fichier solution.

.vb sont tous les fichiers Visual Basic (Feuille module...)

Les sources sont par défaut dans ' C:\Documents and Settings\NomUtilisateur\Studio Projects\nom projet'

Si on compile le projet l'exécutable est dans un sous répertoire \Bin

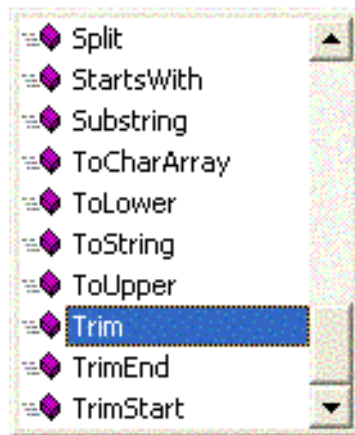
VB propose des AIDES.

Quand on tape du code, VB affiche, quand il le peut, des aides :

- -VB permet de choisir dans une liste une des propriétés d'un objet.

Exemple: Si je crée une variable chaîne de caractères (`Dim Chaine As String` loin), et que je tape le nom de la variable suivi d'un point: 'Chaine.' les possibilités s'affichent.

Chaine.



Quand je pointe dans la liste un des membres (propriété ou méthode) un carré jaune affiche la définition de la fonction avec ses paramètres et une explication.

```
Public Function Trim(ParamArray trimChars() As Char) As String
Public Function Trim() As String
Supprime toutes les occurrences d'un jeu de caractères spécifié dans un tableau à p
instance.
```

q **-VB aide à retrouver les paramètres d'une fonction:**

Si on tape le **nom d'une fonction** et (, VB affiche les paramètres possible

```
R=MessageBox.Show(|
Di ▲ 6 sur 12 ▼ Show (text As String) As System.Windows.Forms.DialogResult
Di text: Texte à afficher dans le message.
```

En plus il affiche les différentes manières d'utiliser les paramètres (le signatures), on peut les faire défiler avec les petites flèches du cadre

q **-VB aide à compléter des mots.**

Si je tape App puis sur le bouton 'A->', Vb affiche la liste commençant pas App

```
Anchor
AnchorStyle
AppActivate
AppDomain
```

Il existe une abondante documentation:

q **-VB donne accès à l'aide sur un mot Clé.** Si le curseur passe sur un affiche la définition de la fonction. Si je clique sur un mot et que s'ouvre et un long texte donne toutes les explications.

q **-VB donne accès à l'aide sur les contrôles.** Si le curseur est sur un tape **F1** l'aide s'ouvre pour donner accès à la description des différ

objet.

- **-L'aide dynamique est constamment mise à jour.** Pour la voir, il faut 'Aide Dynamique' en bas à droite (même fenêtre que la fenêtre propriété) liste de liens en rapport avec le contexte en cours.
- **Enfin il est toujours possible de rechercher des informations** en passant de la fenêtre en haut à droite.

1. Sommaire (plan , arbre de l'aide)
2. Index (liste des mots)
3. Recherche (rechercher un mot)

(Ici on vient de décrire **l'aide interne**; on peut paramétrer le logiciel **l'aide externe** c'est à dire que l'aide s'affiche dans une fenêtre externe cela allège les fenêtres et onglets de l'IDE.)

Erreur.

S'il existe une erreur dans le code au cours de la conception, celle-ci est indiquée par un carré ondulé. Un carré donne la cause de l'erreur si le curseur passe sur la zone de l'erreur.

```
Label1.Texte() = "12"
```

```
'Texte' n'est pas un membre de 'System.Windows.Forms.Label'.
```

Ici la propriété 'Text' a été mal orthographiée.

Si je lance le programme en mode 'Run' et qu'il y a des erreurs, Vb me les indique dans **la liste des tâches** en bas.

Mode débogage (mode BREAK) :

Une fois lancée l'exécution (**F5**), puis stoppée (**Ctrl +Alt +Pause**), on peut

Voir la valeur d'une propriété d'un objet en la sélectionnant avec la souris

```
'Dim MyNumber As Integer
Label1.Text = IsReference(MyArray) '
Label2.Text = Label1.Text = "1" & " (myString) '
```

Il s'affiche un petit cadre donnant la valeur de la propriété d'un objet.

Voir la valeur d'une variable, simplement en positionnant le curseur sur

Par défaut on ne peut pas **modifier le code en mode Break.**

On peut l'autoriser en passant par les menus
Outils/Options/Débogage/Modifier&continuer, activer 'M'autoriser à
fichiers vb durant le débogage' mais la modification n'est pas prise
si on relance le programme!!!

F11 permet l'exécution pas à pas (y compris des procédures appelées)

F10 permet le pas à pas (sans détailler les procédures appelées)

Maj+F11 exécute jusqu'à la fin de la procédure en cours.

En cliquant sur le bouton droit de la souris, on peut **afficher ou définir**
suivante, exécuter jusqu'au curseur, voir la définition de ce qui est sous
définition, c'est l'instruction ou une variable à été déclarée par exemple



On peut grâce au menu débogage puis Fenêtre ouvrir les fenêtres:

Automatique, qui affiche les valeurs des variables de l'instruction en cours et des instructions voisines.

Immédiat où il est possible de taper des instructions ou expressions pour voir leurs valeurs.

Espions permettant d'afficher le contenu de variables ou d'expressions.

Espions Express permet d'afficher la valeur de l'expression sélectionnée.

Points d'arrêts permet de modifier les propriétés des points d'arrêts. on peut ajouter un point d'arrêt en cliquant dans la marge grise à gauche: l'instruction correspondante est mise en marron et l'exécution s'arrêtera sur cette ligne.

Me affiche les données du module en cours.

Variables locales affiche les variables locales.

Modules affiche les dll ou .exe utilisés.

Mémoire, Pile d'appels, Thread, Registres, Code Machine permettent d'étudier du programme à un niveau plus spécialisé et technique.

Il est possible de mettre **des points d'arrêt, des espions** pour arrêter l'exécution et voir la valeur de certaines expressions. (voir traiter les erreurs)

voir [débugage](#) pour plus d'information.



SiteVB **LDF** :

Cours VB.net



1.D Bis

**L' environnement
de développement :
SHARPDEVELOP**

C'est l'IDE: Environnement de développement intégré **GRATUIT**, alternative à la version



Oui, vous pouvez faire du Visual Basic.Net (ou du C#) gratuitement

C'est un **logiciel libre** en **Open Source** (GPL), fonctionne officiellement sous Windows
officiellement sous ME et 98)

Il paraît que SharpDevelop fonctionne sous Windows 98 (non testé, si vous avez essayé, m'en faire
NT 4, Windows 2000 (testé) , XP (testé). Alors que *Visual Studio ne fonctionne pas sur un PC non
Millenium*).

Où le trouver? Comment l'installer?

Respecter l'ordre d'installation.

1/ Télécharger et installer le Framework. (impérativement en premier)

Installer **Microsoft .NET version 1.1 Redistributable package** .

C'est le Framework (la couche logiciel entre l'application et le système), il est télé

[Télécharger le Framework 1.1 français](#) puis l'installer. (**Gratuit**)

2/ Télécharger et installez le SDK .

C'est le Kit de Développement Microsoft .NET Framework: SDK version 1.1 en français.

[Télécharger le SDK](#) (**Gratuit**) Attention, changer la langue et charger la version franç

3/ Télécharger et installez SharpDevelop 1.0.

[Télécharger SharpDevelop 1.0](#) (**Gratuit**)

L'installer en exécutant le fichier '*SharpDevelop_1.0.0.1550_Setup.exe*'.

4/ Configurer SharpDevelop

Au premier démarrage, **créer une nouvelle base de complétion de code**
(option par défaut)

Aller dans le menu 'Outils' - 'Options'

Choisir :

Style visuel : Choisir VBNET dans la liste.

Type de Fichier : cocher 'Fichier source VB.NET' (.vb) en plus.

Le Framework, le SDK et #develop suffisent pour faire des programmes.

On peut aussi installer:

Framework .NET v2.0 Beta (gratuit)

SDK .NET v2.0 Beta (gratuit)

MSDE, SQL SERVER version light (gratuit)

SDK Direct X, pour faire du graphisme ou du multimédia (gratuit)

Quelques liens

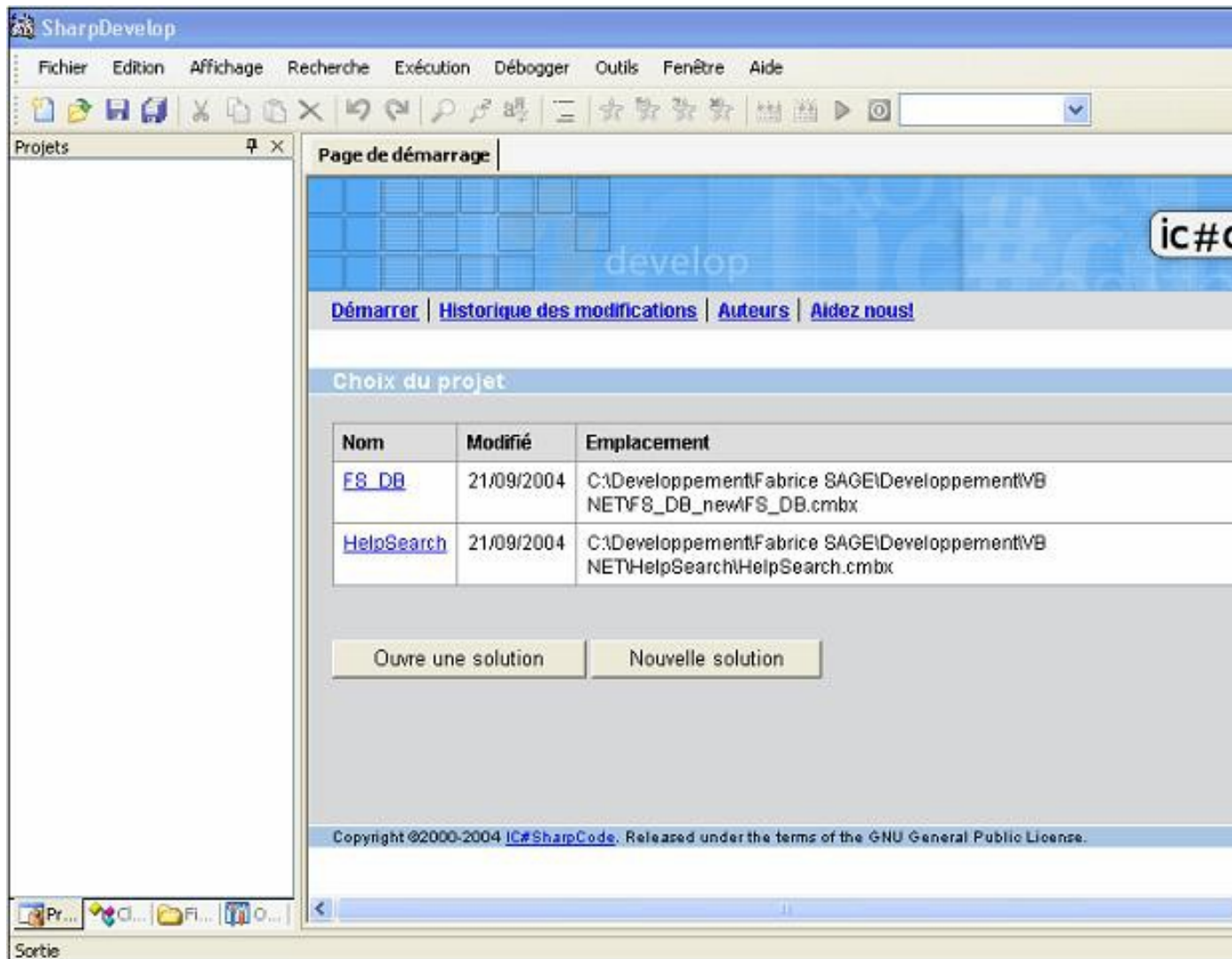
[Petit didacticiel en anglais](#)

[WikiSharpDevelop](#) (en anglais)

[WikiDeboguage](#) (en anglais)

Fenêtre Projet.

Lancer SharpDevelop:

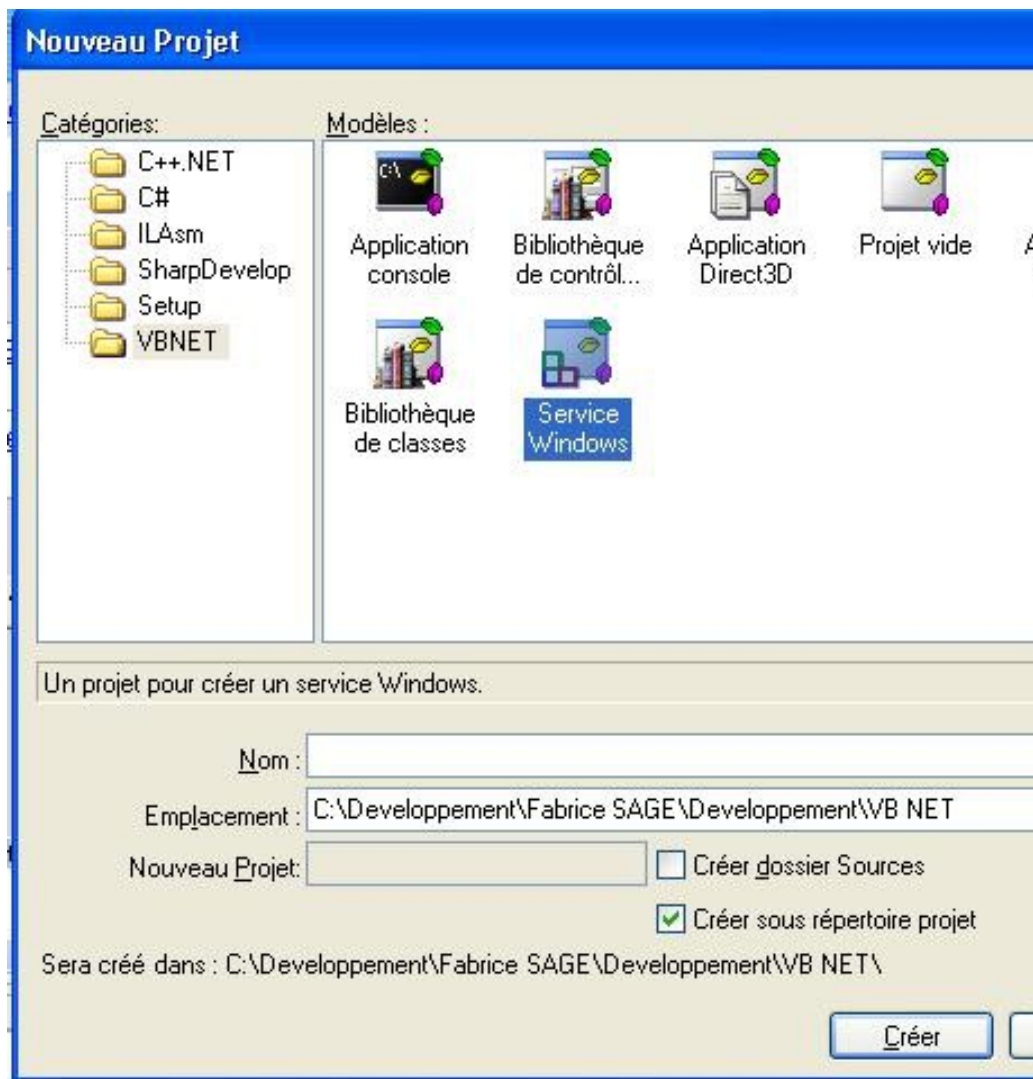


Au lancement de l'application, on peut :

- o ouvrir une solution existante: Bouton 'Ouvrir une solution'
- o créer un nouveau projet (une nouvelle solution) :

Bouton 'Nouvelle solution' ou

Menu 'fichier'-'Nouveau'-'Solution'



Sélectionner la catégorie 'VBNET' et choisir le type d'application à créer. (Dans le cas d'une création d'un projet Visual Basic, il faut choisir dans les 'Modèles': **Application Windows**.) On remarque que #Develop permet aussi d'écrire du C#, du C++.

Puis il faut donner un **nom** au projet (il n'y a pas de nom par défaut), modifier si nécessaire le **chemin de l'emplacement du projet** qui est par défaut 'C:\Documents and Settings\NomUtilisateur\My Documents\SharpDevelop Projects' (cocher si nécessaire 'Créer le répertoire source') enfin valider sur le bouton 'Créer'. Une fenêtre

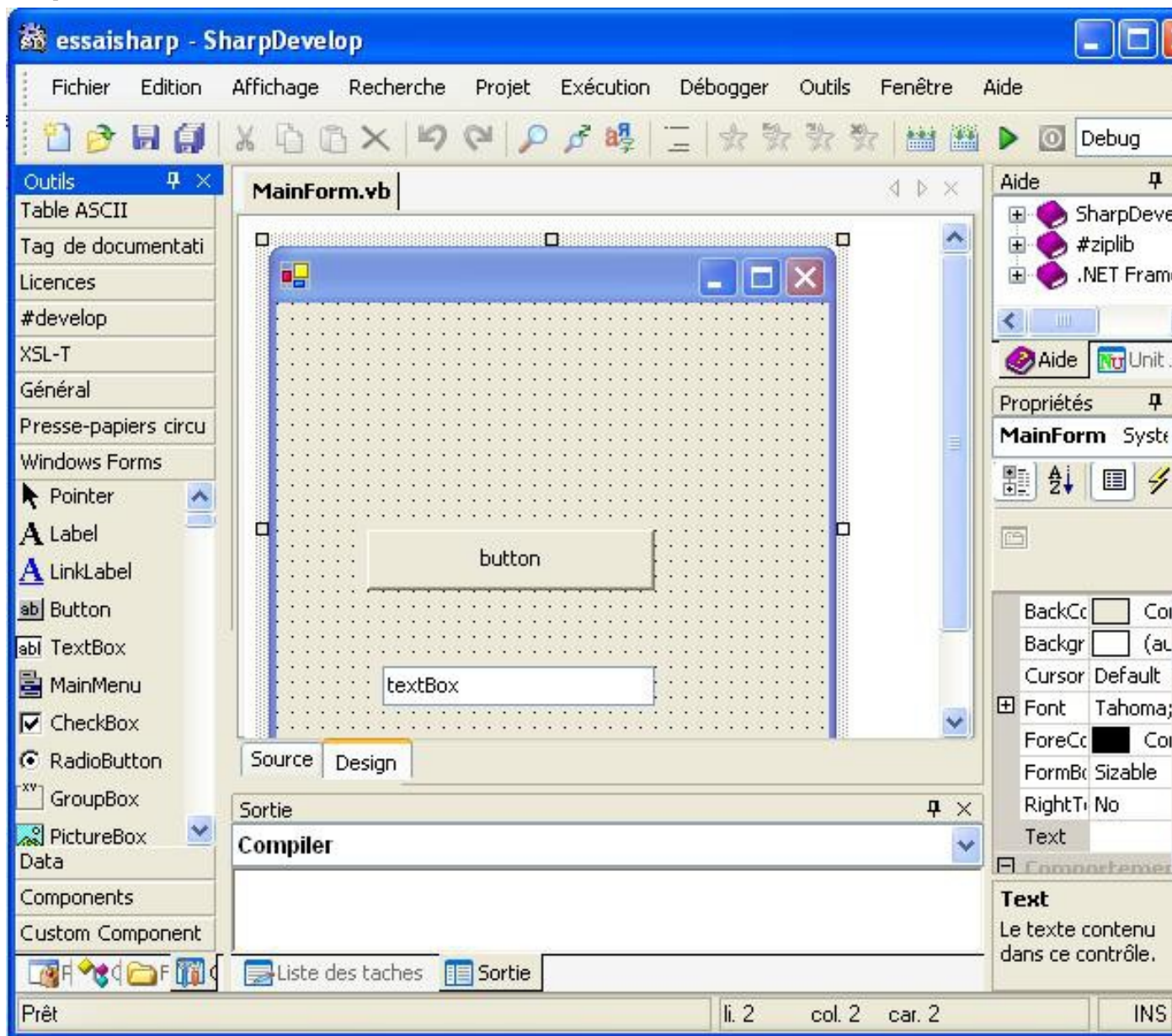
'MainForm' apparait.

- importer / exporter un projet de Visual studio.

En passant par le menu 'Fichier'.

Si l'on veut rajouter des fichiers à notre projet faire :
'Fichier'-'Nouveau'-'Fichier' et catégorie VB.

Quand on ouvre une solution (un projet), le logiciel se présente ainsi:



Noter que le logiciel est **francisé** (Si cela n'a pas été fait, le faire par le menu **Tools**).

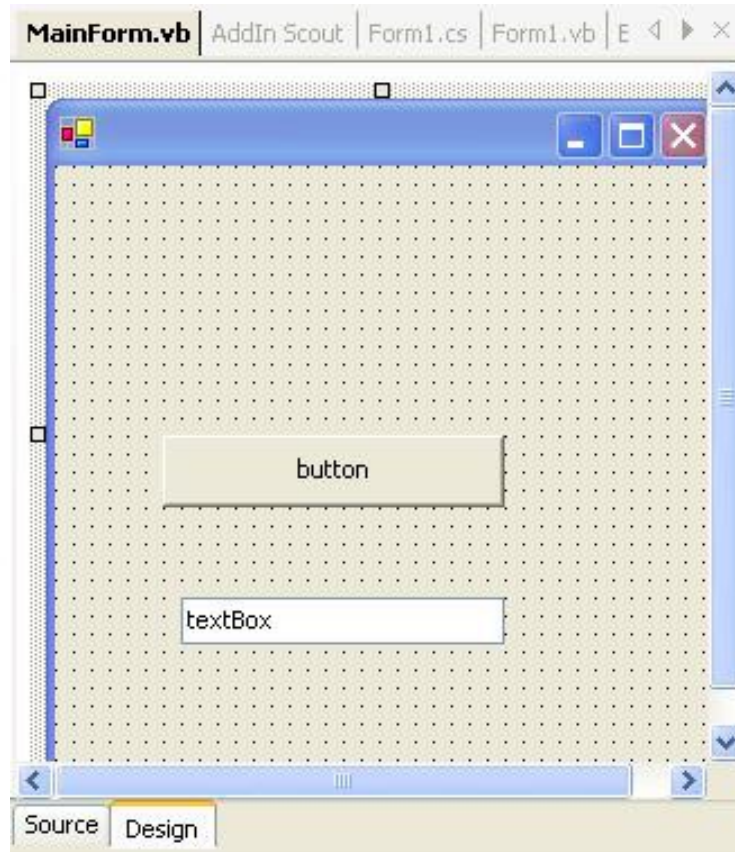
Dans un nouveau projet, créer une fenêtre :

<http://plasserre.developpez.com/v1-dbis.htm> (7 sur 21) [01/03/2005 14:46:32]

Pour ajouter un fenêtre (un formulaire) Cliquer sur le premier bouton **Fichier** (le tooltip dit 'Ouvrir un nouveau Buffer', ou utiliser le menu 'Nouveau', 'Fichier'.

Dans la fenêtre qui s'ouvre , à gauche, choisir 'VB.NET' , à droite 'Form1', une fenêtre 'Form1' apparaît. La première fenêtre qui s'ouvre quand on crée un projet se nomme 'MainForm', si on en ajoute une, elle se nomme 'Form1'.

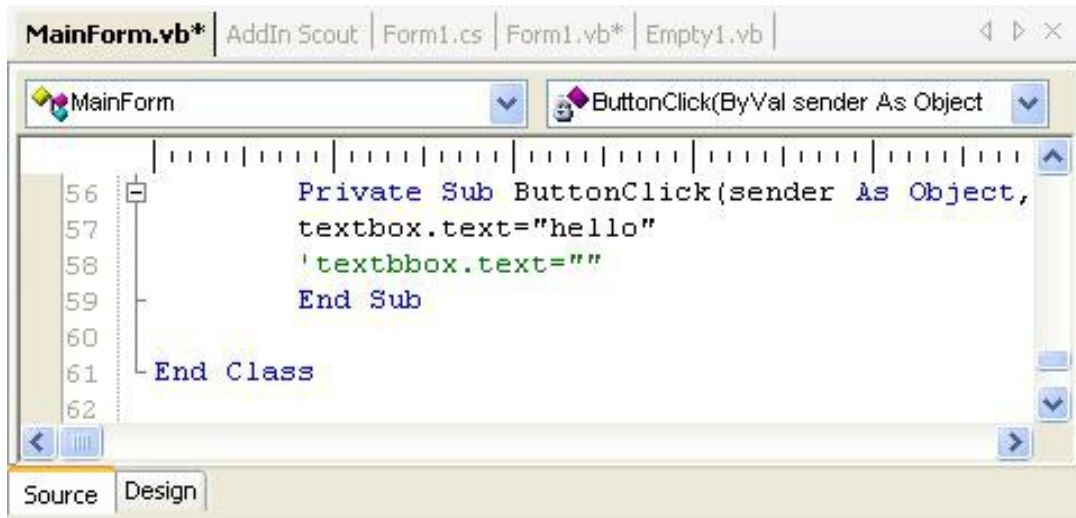
La zone de travail se trouve au centre de l'écran: On voit les onglets **MainForm, Form1** (fenêtre)



En bas les onglets 'Source' et 'Design' permettent de passer de l'affichage du code ('Source') à l'interface utilisateur ('Design'): l'affichage de la fenêtre et de ses contrôles permet de visualiser l'interface.

Voir les procédures:


L'onglet '**Source**' en bas donne accès aux procédures (au code) liées à Form1.



La liste déroulante de droite donne la **liste des objets**. Si on en choisit un, le pointeur de la souris se place sur l'objet lié à cet objet.

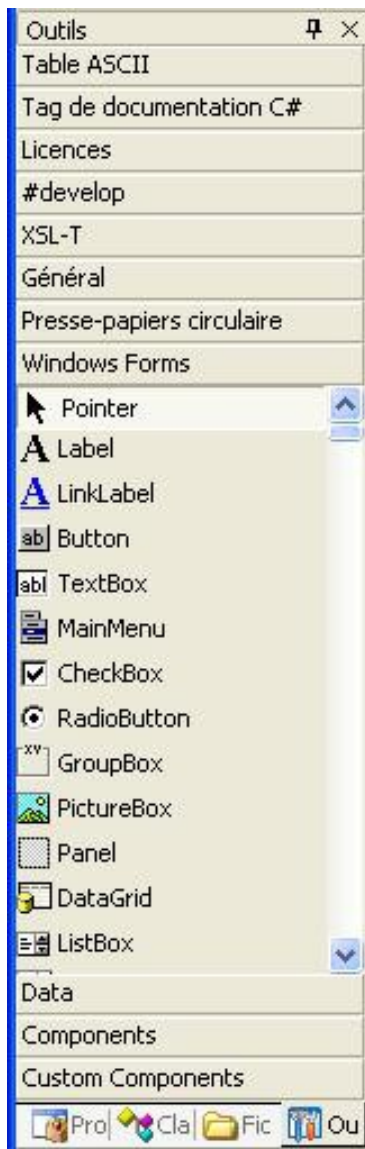
Il est possible en double-cliquant dans le formulaire ou sur un contrôle de se retrouver dans la procédure correspondant à cet objet.

Si la procédure n'existe pas (ButtonClick par exemple), le fait de double-cliquer sur le formulaire ou sur un contrôle crée la procédure.

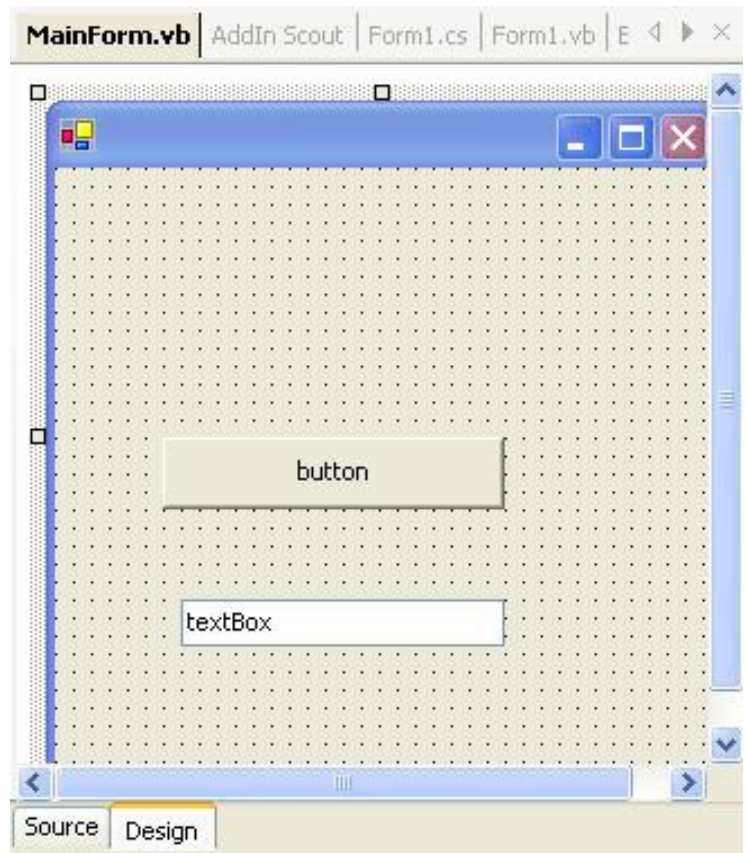
Pour créer les autres procédures événements, utiliser le bouton  qui est sur la barre d'outils à droite, il fait apparaître la liste des événements et permet de créer les procédures.

Ajouter des contrôles à la feuille

Ajoutons un bouton par exemple:



Cliquer sur l'onglet '**Outils**' à gauche et sur '**Windows Forms**', puis sur '**Button**', cliquer sur le bouton, puis déplacer le curseur sans lâcher le bouton :



Un bouton apparaît.

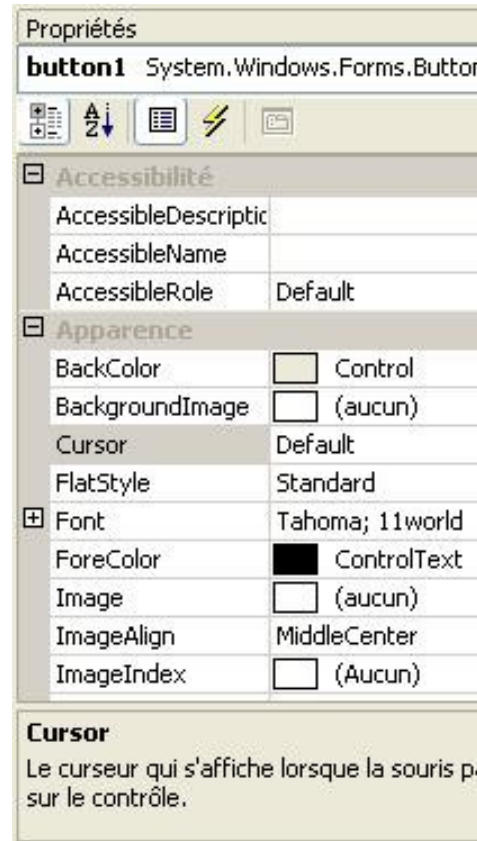
Modifier les propriétés d'un contrôle ou du formulaire.

Quand une feuille ou un contrôle est sélectionné dans la fenêtre Design, **ses propriétés sont accessibles** dans la **fenêtre de propriétés à droite en bas** :

Ici ce sont les propriétés du contrôle 'Button1' (BackColor, Image, Texte..)

Un petit texte d'aide concernant la propriété en cours apparaît en bas.

(on peut modifier les propriétés directement.)



Voir tous les composants d'un projet:

Pour cela il faut utiliser **La fenêtre Projet** et **la fenêtre Class** à gauche, elles permettent de voir et d'avoir accès au contenu du projet:

Au

[Références](#) qui contient les espaces de nom.

[Assembly](#): info nécessaire pour créer un installateur..

Remarque relative aux fenêtres de l'IDE:

Pour **faire apparaître une fenêtre** qui a disparu (fenêtre projet par exemple) utiliser l'attribut `Visible = True` dans le code de la fenêtre 'projet'.

Quand la fenêtre est ancrée (accrochée aux bords), le fait de la déplacer avec sa barre de titre et elle devient autonome.

Pour la 'ré ancrer', il faut double-cliquer dans sa barre de titre.

Tester son logiciel:

On peut compiler le projet avec le premier bouton ci-dessous. **Lancer l'exécution avec le bouton à droite (qui devient rouge pendant l'exécution)** sert à terminer l'exécution.



La **sauvegarde** du projet se fait comme dans tous les logiciels en cliquant sur l'icône de sauvegarde.

Projet et solutions.

Dans la terminologie VB, un **projet** est une application en cours de développement.

Une **solution** regroupe un ou plusieurs projets (C'est un groupe de projets).

#Sharp permet de créer, ouvrir:

des fichiers

des projets/solutions.

Fichiers, Chemins des sources.

Si vous regardez les fichiers correspondant à un projet VB, les extensions sont:

.prjx est le fichier de projet.

.cmbw est le fichier solution.

.vb sont tous les fichiers Visual Basic (Feuille module...)

.ressources pour les ressources.

Les sources sont par défaut dans ' C:\Documents and Settings\NomUtilisateur\Mes documents

Si on compile le projet l'exécutable est dans un sous répertoire `\Bin\Debug` ou `\Bin\Release`

Si vous avez plusieurs versions du framework sur votre machine (version 1.0, version 2.0 Beta), il vous est possible de choisir le compilateur dans les options du projet.

Visual Studio 2003 à version 1.1 du framework.

Visual Studio 2005 à version 2.0 du framework

Il est possible d'importer ou d'exporter au format VisualBasic de Microsoft.

#Develop propose des AIDES.

La fenêtre d'aide à droite donne accès à des aides:

De #develop en anglais, non à jour!!

Du Framework

De zipLib

Si vous avez installé le SDK (SDK Framework .Net et/ou SDK Direct X) , vous avez accès à la fenêtre d'aide (à droite de l'écran) , et donc également à l'intellisense, qui affiche les propriétés, les paramètres des fonctions, des types, ... , des différents objets.

```
Dim R As ResultDialog
R=MessageBox.sho
```

```
End Sub
```

- ◆ Equals
- ◆ ReferenceEquals
- ◆ Show

Shared Function Show(ByVal text As String, ByVal caption As String, ByVal buttons As System.Windows.Forms.MessageBoxButtons, ByVal icon As System.Windows.Forms.MessageBoxIcon, ByVal defaultButton As System.Windows.Forms.MessageBoxDefaultButton, ByVal options As System.Windows.Forms.MessageBoxOptions) As System.Windows.Forms.DialogResult (+11 overloads)

Ici par exemple on a tapé `MessageBox.` , la liste des membres (`Equals`, `Show...`) est aff

Débugage avec le débogueur du SDK de Microsoft:

Ce qui est nommé point d'arrêt dans #develop est en fait **un signet** dans le texte.

Pour déboguer il faut installer le débogueur fourni gratuitement avec le SDK que vous a

Le débogueur est un programme autonome, indépendant de #develop.

Si vous avez installé le Framework, le SDK et #develop , Il est dans le répertoire:

```
C:\Program Files\Microsoft.NET\SDK\v1.1\GuiDebug
```

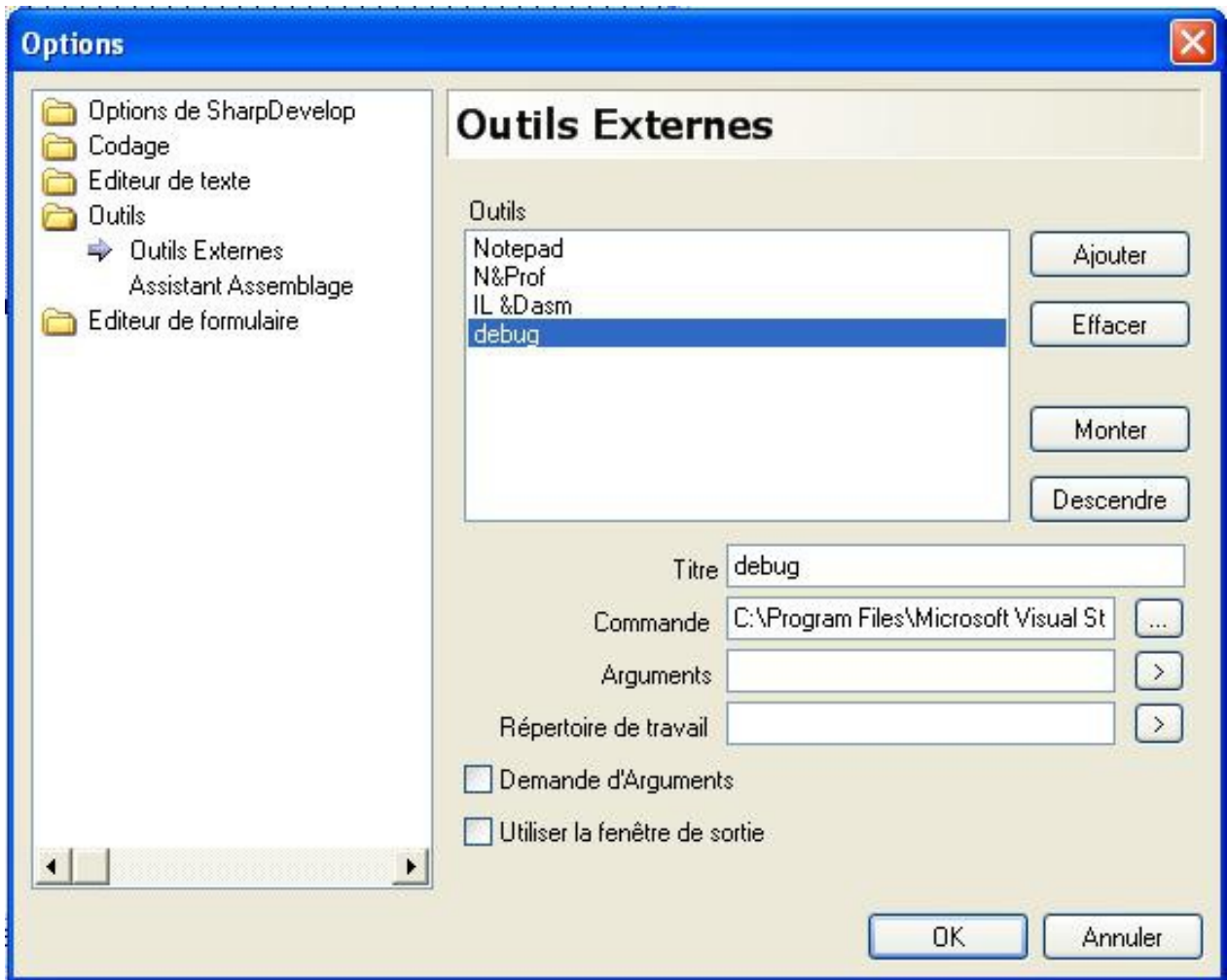
Si vous avez installé Visual Studio.Net, Il est dans le répertoire:

```
C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\GuiDebug
```

et le débogueur en lui même se nomme : [DbgCLR.exe](#)

Pour lancer l'ouvrir rapidement à partir de #develop, il faut ajouter une ligne 'Debug'

Menu 'Outils', sous menu 'Option', et cliquer la ligne outils (outils externes) de l



Cliquer sur le bouton 'Ajouter'

Titre:debug

Commande:

C:\Program Files\Microsoft.NET\SDK\v1.1\GuiDebug\DbgCLR.exe
ou

C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\GuiDebug\DbgCLR.exe

puis cliquer sur "OK"

Maintenant vous avez une ligne 'Debug' dans le menu 'Outils':

Cliquer dessus, cela ouvre le Débogueur 'Microsoft'

Pour charger le programme à déboguer:

Il faut 'charger' le fichier exécutable .EXE

Menu "Débuguer", sous menu "Programmes à déboguer", sélectionner le programme ".F

Il faut 'charger' les sources:

Menu "Fichier", charger tous les fichiers .VB du projet

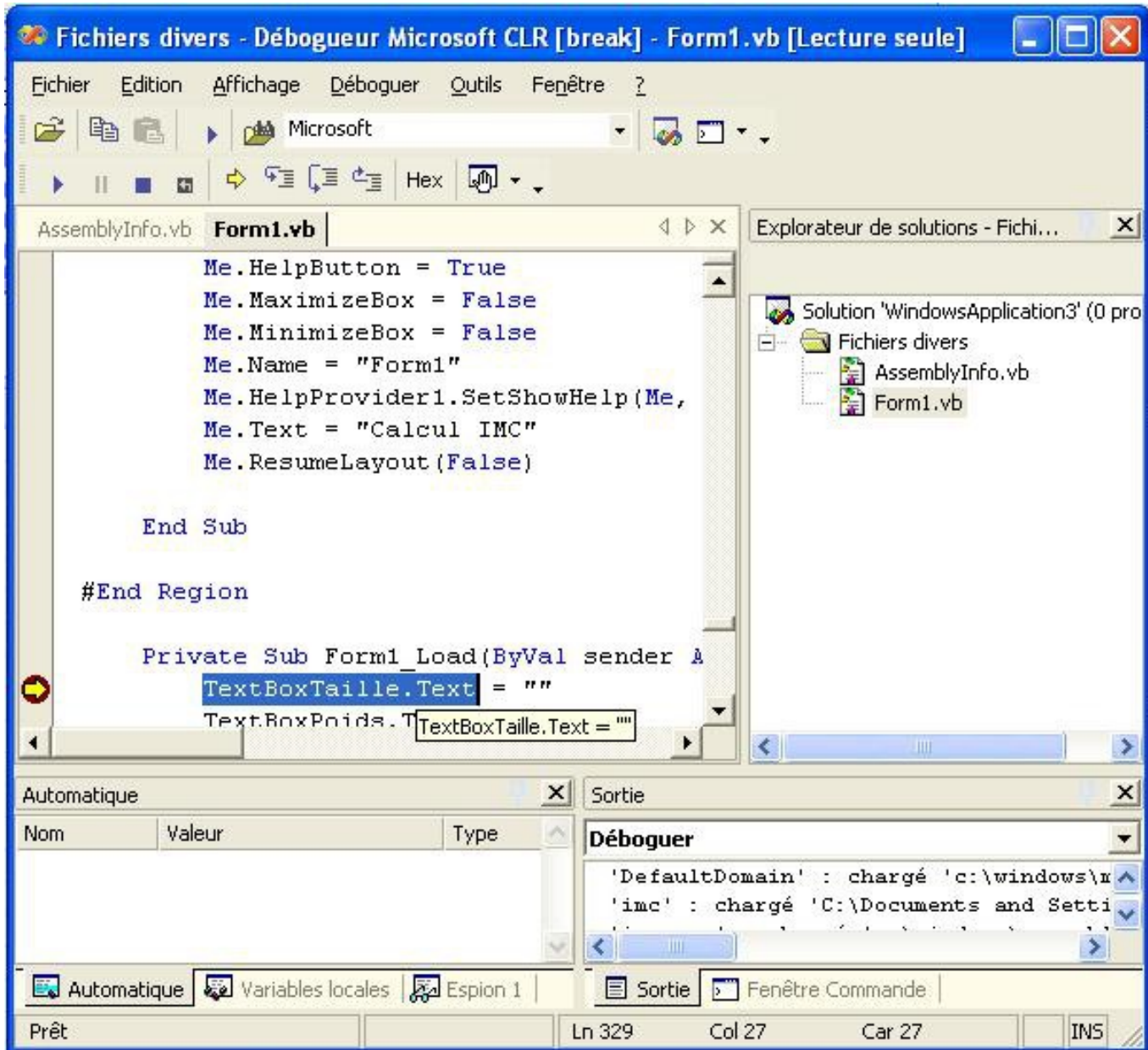
Enregistrer le fichier debugger solution (.dln) en passant par le menu 'fichiers' puis

Cela permet, la fois suivante, d'ouvrir le fichier solution (.dln)

(et non plus la totalité des fichiers sources et exe , ce qui est plus long).

Pour lancer le débogage:

touche F5 ou la flèche droite.



Une fois l'exécution lancée:

On peut la stopper (Ctrl +Alt +Pause).

<http://plasserre.developpez.com/v1-dbis.htm> (17 sur 21) [01/03/2005 14:46:32]

Ajouter des points d'arrêt.

grâce à **des points d'arrêt** (pour définir un point d'arrêt en mode de conception, cliquer sur la marge grise, cela fait apparaître un rond marron. Quand le code est exécuté, il s'arrête).

```

For i= 1 To 6
    Tableau(i)=i*i
Next i

```

Voir la valeur d'une propriété d'un objet en la sélectionnant avec la souris:

```

'Dim MyNumber As Integer
Label1.Text = IsReference(MyArray) '
Label2.Text = Label1.Text = "1" & " " & IsReference(myString) '

```

Il s'affiche un petit cadre donnant la valeur de la propriété d'un objet.

Voir la valeur d'une variable, simplement en positionnant le curseur sur cette variable.

F11 permet l'exécution pas à pas (y compris des procédures appelées)

F10 permet le pas à pas (sans détailler les procédures appelées)

Maj+F11 exécute jusqu'à la fin de la procédure en cours.

En cliquant sur le bouton droit de la souris, on peut

Exécuter jusqu'au curseur

Insérer un point d'arrêt.

On peut grâce au menu 'Débogage' puis 'Fenêtre' ouvrir les fenêtres:

Automatique, qui affiche les valeurs des variables de l'instruction en cours et des instructions suivantes.

Immédiat où il est possible de taper des instructions ou expressions pour les exécuter.

Taper "?I" (c'est l'équivalent de "Print I" qui veut dire: écrire la valeur de la variable I).
valider, cela affiche la valeur de la variable I.

Autre exemple, pour voir le contenu d'un tableau A(), taper sur une seule ligne:
Next i"

Enfin, il est possible de modifier la valeur d'une variable: Taper " I=10" puis valider.
la valeur de la variable.

En bas à gauche on a aussi les fenêtres:

Espions permettant d'afficher le contenu de variables ou d'expressions.

Espions Express permet d'afficher la valeur de l'expression sélectionnée.

Variables locales affiche les variables locales.

Attention, comme dans Visual Studio, il n'est pas possible de modifier les fichiers sources à partir du moment où vous avez démarré le débogage.

Conclusion:

Programme permettant de faire du VB.net gratuitement (rapport qualité/prix infiniment supérieur à Visual Studio)

CONCLUSION D'UN UTILISATEUR:

SharpDevelop est un IDE agréable à utiliser, pour le développement des programmes en VB.net

Il est possible d'atteindre un niveau de qualité équivalent à Visual Studio ou Borland Delphi, sans faire une installation complète. Très ouvert, on peut lui rajouter des plugins. Des plugins externes peuvent être utilisés également avec Visual Studio ou Borland C# Builder.

SharpDevelop est en perpétuelle évolution.

Un forum permet de déposer le descriptif des erreurs rencontrées mais également des modifications, si vous pensez à une évolution qu'il serait bien que SharpDevelop apporte, vous pouvez récupérer le code source et pouvez donc modifier à loisir l'IDE.

Ancien utilisateur de Visual basic 5.0 Pro, Visual Studio 6.0 enterprise (Visual Studio .NET) et de Visual Studio 2003 .Net architecte, je ne me sens pas à l'étroit.

Bien sur, pour les débutants, il manque les assistants de Visual Studio (Crystal Reports). Un problème avec les assistants est qu'une fois qu'on pratique un peu, ils deviennent souvent, il faut repasser derrière eux, pour enlever le superflu de code qu'ils génèrent (ils n'optimisent pas le code).

Il manque également la partie UML de Visual Studio Architecte, mais là on attaque les développeurs.

Par contre SharpDevelop apporte en plus:

Aide à la génération automatique des MessageBox

Aide à la conversion C# vers VB.NET et de VB.NET vers C#

Aide à la génération d'expression régulière.

Importer / exporter un projet de Visual studio.

Support du multi langage.

Il fournit les logiciels:

NDoc : permet de faire des fichiers d'aide compilée au format MSDN, à partir de commentaires dans le code.

NUnits : permet de faire des tests unitaires (!).

SharpQuery : Permet de se connecter aux bases de données .

Installateur:

Comment créer un installateur avec #develop? Merci de votre expérience.

Merci à Fabrice SAGE pour son aide.

Merci à Hubert WENNEKES, CNRS Institut de Biologie de Lille pour son aide.



1.3	Introduction au langage.		
-----	-------------------------------------	--	---

**Nous allons
étudier:**

Le **langage** Visual Basic.Net qui est utilisé dans les **procédures**

Comme nous l'avons vu, le **langage Visual Basic** sert à

- **agir sur l'interface** (Afficher un texte, ouvrir une fenêtre, remplir une liste, un tableau, poser une question).

```
Button1.Text="Bonjour"
```

- **effectuer des calculs, des affectations** en utilisant des **variables**.

```
B=A+1
```

- Faire des **tests** avec des structures de décision: évaluer des conditions des comparaisons et prendre des décisions.

```
If A=1 Then...End If
```

- travailler en **boucle** pour effectuer une tâche répétitive.

```
For I=0 To 100... Next I
```

Tout le travail du programmeur est là.

Dans VB.Net nous avons à notre disposition:

- Les **Classes venant du Framework**, on travaille sur des objets en utilisant leurs propriétés, leurs méthodes.

Il existe des centaines de 'Classes': les plus utilisées sont les classes **String** (permettant de travailler sur des chaînes de caractères), **Maths** (permettant d'utiliser des fonctions mathématiques), **Forms** (permettant l'usage de formulaire, de fenêtre), **Controls** (donnant accès aux contrôles: bouton, case à cocher, liste..)

Elles sont communes à tous les langages utilisant le Framework.

Exemple d'utilisation de la Class TextBox (contrôle contenant du texte):

```
TextBox1.Text="Hello" 'Affiche "Hello" dans le textbox.
```

- Les **instructions du Common Langage Runtime** (Un autre constituant de Visual Studio).

Il s'agit d'instruction, de mot clé.

Exemple:

```
A = Mid(MaString, 1, 3) 'Mid retourne une partie de la chaîne de caractères.
```

- Les **Classes de compatibilité VB6**. Elles ne dépayseront pas ceux qui viennent des versions antérieures de VB car elles reprennent la syntaxe utilisée dans VB6.

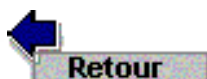
Ajoute à VB.Net des fonctions VB6

L'outils d'import automatique de VB6

Site

vers VB.Net en met beaucoup dans le code. Il faut à mon avis éviter de les utiliser. Ce cours 'pur' VB.Net n'en contient pas.

Pour le moment cela peut paraître un peu compliqué, mais ne vous inquiétez pas, cela va devenir clair.



Site LDF Cours VB.net



Pour écrire un programme, aller du problème à résoudre à un programme exécutable, il faut passer par les phases suivantes:

▫ **Analyse du cahier des charges.**

Il doit être clair, exhaustif, structuré.

▫ **Analyse générale du problème.**

Il existe des méthodes pour professionnels (MERISE, JACKSON..), nous utiliserons plutôt **l'analyse fonctionnelle**: Le problème global est découpé en sous problèmes nommés fonctions. Chaque fonction ne contient plus qu'une partie du problème. Si une fonction est encore trop complexe, on itère le processus par de nouvelles fonctions à un niveau plus bas.

▫ **Analyse détaillée.**

Chaque fonction est mise en forme, la logique de la fonction est écrite dans un pseudo langage (ou **pseudo code**) détaillant le fonctionnement de la fonction. Ce pseudo code est universel, il comporte des mots du langage courant ainsi que des mots relatifs aux structures de contrôle retrouvées dans tous les langages de programmation.

▫ **Codage.**

Traduction du pseudo code dans le langage que vous utilisez.

q **Test**

Car il faut que le programme soit valide.

Exemple simpliste:

q **Analyse du cahier des charges.**

Création d'un programme affichant les tables de multiplication, d'addition, de soustraction.

q **Analyse générale du problème.**

Découpons le programme en diverses fonctions:

Il faut créer une fonction 'Choix de l'opération', une fonction 'Choix de la table', une fonction 'TabledeMultiplication', une fonction 'TabledeAddition', une fonction 'Affiche'...

q **Analyse détaillée.**

Détaillons la fonction 'TabledeMultiplication'

Elle devra traiter successivement (pour la table des 7 par exemple)

1X7
2X7
3X7..

Voici l'algorithme en **pseudo code**.

Début

 Pour i allant de 1 à 10

 Ecrire (i*7)

 Fin Pour

Fin

q **Codage.**

Traduction du pseudo code en Visual Basic.

```
Sub MultiplicationPar7
Dim i As Integer
For i=1 to 10
    Call Affiche(i*7)
next i.
End Sub
```

q **Test**

Ici il suffit de lancer le programme pour voir si il marche bien..



L'algorithme détaille , en pseudo code, le fonctionnement d'une fonction et en décrit la logique.

Il faut avouer que dans la pratique, la phase d'analyse détaillée avec écriture du pseudo code se passe dans la tête et que bien souvent le code VB est écrit directement, mais ce qui suit est DIDACTIQUE et comme on l'a dit, commun à tous les langages.

Etudions cette logique valable pour tous les langages de programmation:

Structure séquentielle d'un programme:

Au sein d'une procédure, la structure d'un programme est généralement séquentielle.

Le code d'une procédure est fait d'une succession de lignes (ou **instructions**) qui seront lues et traitées les unes après les autres.

Instruction 1

Instruction 2

Instruction 3

..

En VB on peut mettre plusieurs instructions sur la même ligne séparées par ":"

Instruction1 : Instruction2

Les variables:

Elle contiennent les informations nécessaires au déroulement du programme (C'est le même sens qu'en mathématique)

Chaque variable a un **nom** (identifiant) et un **type**. Ce dernier indique la nature de l'information que l'on souhaite mettre dans la variable: Entier, réel, booléen, caractère, chaîne de caractères, objet.

Exemple: la variable 'Total' contiendra un réel dans un programme de comptabilité.

Affectation (ou Assignment):

C'est une instruction consistant à donner une valeur à une variable.

En langage algorithmique on l'indique par '<-'

X <- 2 veut dire: donner à la valeur X la valeur 2

Z <- X+1 veut dire: donner à la variable Z la valeur de la variable X à laquelle on ajoute 1 (Z prendra la valeur 2+1 =3).

Cela revient à évaluer l'expression de droite et à en mettre la valeur dans la variable de gauche.

En VB le signe d'affectation est '=' on écrit donc:

Z=X+1

Attention ce n'est pas une égalité mais une affectation.

Les choix:

Le programme doit pouvoir choisir parmi deux ou plusieurs possibilités en fonction d'une condition :

Si Condition Alors

 Action 1

Sinon

 Action 2

Fin Si

Si Condition est vraie Action 1 est effectuée, sinon Action 2 est effectué.

Parfois il n'y a pas de seconde branche :

Si Condition Alors

 Action 1

Fin Si

ou sur une seule ligne:

Si Condition Alors Action 1

Il peut y avoir plusieurs conditions imbriquées :

Si Condition 1 Alors

 Si Condition 2 Alors

 Action 1

Sinon

 Action 2

Fin Si

Sinon

 Action 3

Fin Si

Noter bien le retrait des lignes de la seconde condition afin de bien visualiser la logique du programme :

Action 2 est effectuée si la Condition 1 est remplie et la Condition 2 n'est pas remplie.

En VB cela correspond à l'instruction **IF THEN**

```
If Condition 1 Then
```

```
    Action 1
```

```
Else
```

```
    Action 2
```

```
End If
```

Décider entre :

Il est parfois nécessaire d'effectuer **un choix** parmi plusieurs solutions :

Décider Entre

```
    Quand Condition 1 Alors
```

```
        Action 1
```

```
    FinQuand
```

```
    
```

```
    Quand Condition 2 Alors
```

```
        Action 2
```

```
    FinQuand
```

```
    ..
```

```
    ..
```

```
    Autrement
```

```
        Action 4
```

```
    FinAutrement
```

FinDécider

Si la condition 1 est remplie Action 1 est effectuée puis le programme saute après FinDécider.

Si la condition 1 n'est pas remplie, on teste la condition 2..

Si aucune condition n'est remplie on saute à **Autrement**, on effectue Action 4.

On pourrait aussi parler de sélection :

Sélectionner.

Le cas : condition 1

Action 1

Le cas : condition 2

Action 2

..

Les autres cas

FinSélectionner

En VB cela correspond à

Select Case Valeur

Case condition 1

Action 1

Case condition 2

Action 2

..

Case Else

Action 4

End Select

Si Valeur=Condition 1 Action 1 est effectuée, si Valeur=Condition 2 Action 2 est effectuée...

Pour :

Permet de répéter une séquence un nombre de fois déterminé :

Le cas le plus classique est :

Pour I variant de 0 à N Répéter

Action

FinRépéter

I prend la valeur 0, 'Action' est effectuée,

puis I prend la valeur 1, Action est effectuée,

puis I prend la valeur 2..

cela jusqu'à N

La boucle tourne N+1 fois (car ici on commence à 0)

Cela se nomme une **itération**.

Intérêts?

Au lieu de faire

Afficher (1*7)

Afficher (2*7)

Afficher (3*7)

Afficher (4*7)

...

on remarque qu'un élément prend successivement la valeur 1, 2, 3, ..

Une boucle peut faire l'itération:

Pour i allant de 1 à 10 Répéter

Affiche (i*7)

Fin répéter

La variable dite 'de boucle' prend bien les valeurs 1 puis 2 puis 3.. ; elle est utilisée dans le corps de la boucle.

En VB

```
For i=0 To N
```

..

```
Next i
```

On peut aussi **boucler en parcourant tous les éléments d'une collection.**

(Une collection est une liste d'objets, liste de taille variable en fonction de ce qu'on ajoute ou enlève.)

Pour Chaque élément de la liste

Action

Fin Pour

En VB :

```
For each... In
```

```
  
```

```
Next
```

Tant que :

Permet de faire une **boucle sans connaître le nombre d'itérations à l'avance.**

Tant Que Condition

Action

Fin Tant Que

L'action qui est dans la boucle doit modifier la condition afin qu'à un moment 'Tant que' ne soit pas vérifié et que l'on sorte de la boucle. Sinon la boucle tourne sans fin.

Pour plus cadrer avec la réalité :

Faire tant que condition

Action

Boucler

En VB :

Do while Condition

Action

Loop

Il existe une boucle équivalente :

Répéter

Action

Jusqu'à Condition

En VB :

Do

Action

Loop Until Condition

'Sous-programme' ou 'procédure' :

On a déjà vu cette notion.

Quand on appelle une procédure, le logiciel 'saute' à la procédure, il effectue celle-ci puis revient effectuer ce qui suit l'appel.

Et VB les sous-programmes (ou procédures) sont des **Sub** ou des **Function**. Pour appeler une procédure on utilise **Call NomProcédure()** ou **NomProcédure()**

```
Instruction 1
Instruction 2
Call
MaProcédure()3
Instruction 4
Instruction 5

Sub MaProcédure
Instruction 10
.
Instruction 11
End Sub
```

Le programme effectuera les instructions 1, 2, 3, 10, 11, 4, 5.

Une opération complexe peut être découpée en plusieurs procédures ou sous-programmes plus petits et plus simples qui seront appelés.

On peut fournir aux procédures des **paramètres**, ce sont des variables qui seront transmises à la procédure.

Exemple:

Création d'une Procédure 'MaProcédure' recevant 2 paramètres:

```
Sub MaProcédure(paramètre1, paramètre2)
```

```
..
```

```
End Sub
```

Exemple d'appel de la procédure 'Maprocédure' en envoyant 2 paramètres:

Call MaProcédure(premierparamètre, secondparamètre)

Exemple : si j'écris Call MaProcédure(2,3)

dans la procédure MaProcédure paramètre1=2 et paramètre2=3.

Il est nécessaire de définir le type des paramètres:

Sub MaProcédure(paramètre1 **As Integer**, paramètre2 **As Integer**) indique que la procédure attend 2 entiers.

Il y a 2 manières d'envoyer des paramètres :

Par valeur : (By Val) c'est la valeur, le contenu de la variable qui est envoyé.

Par référence : (By Ref) c'est l'adresse (le lieu physique où se trouve la variable) qui est envoyée. Si la Sub modifie la variable, cette modification sera visible dans la procédure appelante après le retour.

Parfois on a besoin que la procédure appelée retourne une valeur dans ce cas il faut créer une **fonction**:

```
Function MaFonction As Integer  
.  
End Function
```

Pour appeler la fonction:

```
ValeurRetournée=MaFonction()
```

Les tableaux:

Un tableau de variables permet de stocker plusieurs variables de même type sous un même nom de variable, chaque élément étant repéré par un *index* ou *indice*.

C'est une suite finie d'éléments.

Soit un tableau A de 4 éléments

3
12
4
0

Pour accéder à un élément il faut utiliser l'indice de cet élément.

L'élément d'index 0 se nomme A[0] et contient la valeur 3.

On remarque que le premier élément est l'élément d'index 0 (ZERO).

L'élément d'index 1 se nomme A[1] et contient la valeur 12.

Quand on crée un tableau, **il a un nombre d'éléments bien défini**: 4 dans notre exemple d'index 0 à 3.

Pour donner une valeur à un des éléments , on affecte la valeur à l'élément.

```
A[2] <- 4
```

Pour lire une valeur dans un tableau et l'affecter à une variable x:

```
x <- A[2]
```

Traduction en VB

```
A(2)=4
```

```
x = A(2)
```

Pour parcourir tous les éléments d'un tableau on utilise une boucle:

Exemple: afficher tous les éléments du tableau tab qui contient n éléments.

début

```
Pour i de 0 à n-1 Répéter  
    écrire(tab[i])  
Fin Répéter
```

fin

Il existe des **tableaux multidimensionnels avec plusieurs index**:

Voyons les index de chaque élément:

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

B[1,0] désigne l'élément rouge (ligne 2, colonne 1).

Voyons par exemple, le contenu de chaque élément:

3	12	0
18	4	5
2	2	8

Ici B[1,0] = 18

Notion de Collections:

Une collection permet de stocker plusieurs variables ou objets, chaque élément étant repéré par un *index* ou *indice*.

Soit la collection **Col**, au départ elle est vide.

J'ajoute des éléments (ou items) à cette collection.

```
Col.Ajouter ("Toto")
```

Voici la collection:

Toto

La collection a maintenant 1 élément.

```
Col.Ajouter("Lulu")
```

```
Col.Ajouter("Titi")
```

Toto
Lulu
Titi

La collection a 3 éléments maintenant.

```
Col.Retirer(2) enlève l'élément numéro 2
```

Toto
Titi

La collection n'a plus que 2 éléments maintenant.

On voit que le nombre d'éléments n'est pas connu à l'avance, il varie en fonction des éléments ajoutés (ou retirés)

Un élément est repéré par un indice.

En VB

```
Col.Add 'Ajoute un élément
```

```
Col.Remove 'Enlève une élément
```

Utilisation de Flag:

Un Flag (ou **drapeau**) est une variable utilisée pour enregistrer un état, la valeur de cet état servant à déclencher ou non des actions. C'est une manière de retenir qu'un évènement s'est produit.

Si le drapeau est abaissé, les voitures roulent..

Exemple: Utiliser un Flag pour sortir d'une boucle:

On utilise FlagSortir.

vbintro

```
FlagSortir=Faux
```

```
Tant que FlagSortir =Faux
```

Si on doit sortir de la boucle, on met la valeur de FlagSortir à Vrai

Boucler

En VB:

```
FlagSortir=Faux
```

```
Do while FlagSortir =Vrai
```

Si on doit sortir de la boucle, on met la valeur de FlagSortir à Vrai

Loop

Tant que FlagSortir =Faux la boucle tourne.



Site **LDF** Cours **VB.net**



1.5L'Affectation.

Suivant

Sommaire

C'est l'instruction la plus utilisée en programmation.

On peut aussi utiliser le terme '**Assignment**' à la place de l'affectation.

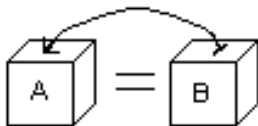
`NomdeVariable=Expression` est une affectation.

Elle transfère le résultat de l'expression située à droite du signe 'égal' dans la variable (ou la propriété) à gauche du signe égal.

Exemple:

`A=B` est une affectation (ou assignation.)

A=B affecte la valeur de la variable B à la variable A, la valeur de B est mise dans A.



Si

`A=0`

`B=12`

`A=B` entraîne que `A=12`

Si nécessaire l'expression, (l'élément) à droite du signe = est évaluée, calculée avant d'être affectée à la première variable.

Si

A=0

B=12

A=B+2 entraîne que A=14

L'affectation permet donc de **faire des calculs** :

Si NombredHeure=100 et TauxHoraire=8

Paye= NombredHeure * Tauxhoraire

Paye prend la valeur 800 (notez que '*', l'étoile veut dire : multiplication.)



Attention dans le cas de l'affectation "=" ne veut donc pas dire égal.

A=A+1 est possible

Si A=1

A=A+1 entraîne que A=2

On verra qu'il existe des variables **numériques** ('Integer', 'Single') et **alphanumériques**(chaîne de caractères ou 'String'), l'affectation peut être utilisée sur tous les types de variables.

Le second membre de l'affectation peut contenir des constantes, des variables, des calculs dans le cas de variables numériques.

`A=B+2+C+D`

On ne peut pas affecter une variable d'un type à une variable d'un autre type:

si A est numérique et B est alphanumérique (chaîne de caractères)
`A=B` n'est pas accepté.

Ecriture compacte:

`A=A+1` peut s'écrire de manière plus compacte : `A += 1`

`A=A*2` peut s'écrire de manière plus compacte : `A *= 2`

`A=A&"Lulu"` pour une variable chaîne de caractère peut s'écrire de manière plus compacte : `A &= "Lulu"`

L'affectation marche pour les objets, leurs propriétés..

`Bouton1.BackColor= Bouton2.BackColor`

Signifie que l'on donne au Bouton1 la même couleur de fond que celle du bouton2: on affecte la valeur BackColor du Bouton2 au Bouton1.

Attention le signe '=' signifie par contre 'égal' quand il s'agit d'évaluer une condition, par exemple dans une instruction If Then (Si Alors):

`If A=B then` 'signifie: Si A égal B alors...

Permutation de variables:

C'est un petit exercice:

J'ai 2 variables A et B contenant chacune une valeur.

Je voudrais mettre dans A ce qui est dans B et dans B ce qui est dans A.

Si je fais

`A=B`

B=A

Les 2 variables prennent la valeur de B!!

Comment faire pour permuter?

Et bien il faut utiliser une variable intermédiaire C qui servira temporairement à conserver le contenu de la variable A:

C=A

A=B

B=C

Voilà, on a bien permuté.



Site **LDF** Cours **VB.net**



1.6	Les Variables		
		Suivant	Sommaire

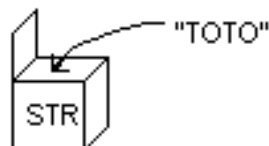
C'est le même sens qu'en mathématique. Sauf qu'en VB la variable est un **objet**.

Une variable sert à stocker un nombre, du texte (chaîne de caractères), une date, un objet..

Une variable est un objet qui contient donc un nombre, du texte..

Une variable a un **nom** et un **type** qui indique ce que l'on peut y mettre.

si **STR** est une variable de type chaîne de caractère (ou **String**) : je peux y mettre une chaîne de caractère (« TOTO » par exemple)



STR="TOTO"

Si **Age** est une variable de type numérique ; je peux y stocker un nombre (45,2 par exemple).

Age=45,2

Nom des variables :

On peut utiliser dans les noms de variable des **majuscules ou des minuscules** mais pour VB il n'y a pas de différence.

Exemple :

On ne peut pas déclarer une variable VB et une variable vb.

Si on déclare une variable nommée 'VB' et si ultérieurement on tape 'Vb', le logiciel le transforme automatiquement en 'VB'.

- **On peut mettre des chiffres et des lettres** dans les noms de variable mais pas de chiffres en premier caractère.

`2A` n'est pas un nom valide

`Nom2` l'est.

- **Certains caractères de ponctuation** (' »..) ne peuvent pas être utilisés, d'autres "_" et "-" sont permis :

`Nom_Utilisateur` est valide.

L'espace n'est pas permis.

- Bien sur, **les mots clé de VB ne peuvent pas être utilisés** : On ne peut pas nommer une variable Form ou BackColor

Il est conseillé de donner des **noms explicites** qui rappelle ce que contient la variable:

`Nom_Utilisateur` est explicite, pas `NU`.

Parfois on indique en début de nom, par une lettre, le **type de variable** ou sa **portée** (la zone de code où la variable existe).

`s_Nom` 'Le s indique qu'il s'agit d'une variable String (Chaîne de caractères)

`iIndex` 'Le i indique qu'il s'agit d'une variable Integer (Entier)

`gNomUtilisateur` 'g indique que la variable est globale

Il est possible de '**forcer**' le **type de la variable** en ajoutant un caractère spécial à la fin du nom de la variable.

`Dim c$ = "aa"` ' \$ force c a être une variable String.

De même % force les Integer, & les long..

Cette notation est moins utilisée.

Voir en annexe, en bas de page des recommandations supplémentaires.

Déclaration:

Avant d'utiliser une variable, il faut **la déclarer**, la créer, pour cela on utilise l'instruction `Dim`:

`Dim A As Integer` 'Déclare une variable nommée 'A' de type Entier

Avant la ligne, A n'existait pas, après le DIM, A existe et contient 0.

Il faut aussi parfois l'initialiser, c'est à dire lui donner une valeur de départ : `A=3`

On peut déclarer et initialiser en même temps:

vbintro

```
Dim A As Integer =3
```

En pratique:

Les variables après avoir été déclarées, vont servir à stocker des valeurs, à effectuer des calculs, on peut afficher ensuite leur valeur.

Exemple simpliste d'utilisation de variables:

```
Dim A As Integer      'Création d'une variable A
```

```
Dim B As Integer      'Création d'une variable B
```

```
Dim Total As Integer  'Création d'une variable total
```

```
A=3                   'Mettre '3' dans A
```

```
B=2                   'Mettre '2' dans B
```

```
Total=A+B            'Mettre dans la variable 'Total' la  
valeur de A et de B
```

```
LabelTotal.Text= Total.ToString  'Afficher le total dans  
un label (un label est une zone permettant d'afficher du  
texte), pour cela transformer 'Total' qui est un entier en  
chaîne de caractères puis le mettre dans un label.
```

Noter bien la différence entre :

```
Dim Total As Integer
```

```
Total= 123           'Total est une variable numérique (Integer  
ou Entier) contenant le nombre 123
```

et

```
Dim Total2 As String
```

```
Total2= "123"        'Total2 est une variable string  
contenant la chaîne de caractère "123" c'est à dire les  
caractères "1" , "2" et "3"
```



On peut afficher les chaînes de caractères (ou String), pas les variables numériques. On fait des calculs avec les variables numériques.

Il faudra donc parfois **convertir** le contenu d'une variable d'un type dans un autre type. (convertir une variable numérique en String pour l'afficher par exemple ou convertir une variable String en numérique pour faire un calcul) on apprendra cela plus loin.

L'exemple simpliste juste au dessus en montre un exemple: il faut convertir Total qui est un entier en string pour l'afficher.

Concernant les variables numériques:

En mathématique, les entiers ou les réels peuvent être infiniment grands, en informatique les entiers ou les réels sont stockés sur un certains nombre d'octets. Ce qui fait qu'il existe une **limite supérieure** dans les nombres que l'on peut stocker.

Exemple un Entier 'Integer' est stocké sur 32 bits soit 4 octets, on conçoit donc qu'il y a une valeur maximum (c'est 2147483647)

Si on dépasse cette valeur VB le signale en déclenche une erreur.

Les variables numériques peuvent être **signées** (accepter les valeurs négatives ou positives) ou **non signées** (Comme le Type 'Byte' qui ne contient que des valeurs positives)

Il existe différents types de variable:

Nom : **Contient :**

Boolean Contient une valeur Booléenne :
True ou False.

Byte Contient les nombres entiers de 0 à 255 (sans signe)

Short Entier sur 16 bits (-32768 à 32768)

Integer Entier sur 32 bits (-2147483648 à 2147483647)

Long Entier sur 64 bits (-9223372036854775808 à 9223372036854775807)

Single Nombre réel en virgule flottante (-1,401298 *10⁻⁴⁵ à 1,401298 10⁴⁵)

Double Nombre réel en virgule flottante double précision. (..puissance 324)

Decimal Nombre réel en virgule fixe grande précision

Char 1 caractère alphanumérique

String Chaîne de caractère de longueur variable (jusqu'a 2 milliard de caractères)

DateTime Date plus heure

Object Peut contenir tous les types de variables mais aussi des contrôles, des fenêtres..

Structure Ensemble de différentes variables définies par l'utilisateur.

Détaillons :

Variable Booléenne.

Contient une valeur Booléenne : True ou False

Exemple:

Dim B As Boolean

B=True

Variable entière:

Byte, Short, Integer, Long

Byte Contient les nombres entiers de 0 à 255 (sans signe)

Short Entier sur 16 bits (-32768 à 32768)

Integer Entier sur 32 bits (-2147483648 à 2147483647)

Long Entier sur 64 bits (-9223372036854775808 à 9223372036854775807)

Pour une variable entière il n'y a pas de possibilité de virgule!! attention, une division de 2 entiers donne un entier.

Attention ,les variables numériques ne peuvent pas contenir de nombre infiniment grand: Il y a une limite maximum : un Integer est par exemple codé sur 32 bits ce qui fait qu'il peut varier de -2147483648 à 2147483647. Si on utilise une valeur trop grande, une erreur se produit. Les entiers peuvent être positifs ou négatifs (Sauf les Bytes).

Plus on augmente le type (Long au lieu de Integer) plus on peut y mettre des grands nombres. Mais cela prend aussi plus de place et le traitement des opérations est plus long.

Les processeurs travaillant sur '32 bits', travailler plutôt sur les **Integer** (qui sont codés sur 32 bits aussi), c'est plus rapide,

que sur les short.

On utilise largement les 'Integer' comme variable de boucle, Flag, là où il y a besoin d'entier...(Les calculs sur les réels en virgule flottante sont beaucoup plus lent.)

Exemple:

```
Dim I As Integer
```

```
I=12
```

Le type de données **Byte** est utilisé pour contenir des données binaires (octet codant de 0 à 255) non signé.

Variable réel avec virgule:

Un réel peut avoir une partie fractionnaire: 1,454 est un réel.

Single, Double, Decimal.

Single Nombre réel en virgule flottante (-1,401298 *10⁻⁴⁵ à 1,401298 10⁴⁵)

Double Nombre réel en virgule flottante double précision. (..puissance 324)

Decimal Nombre réel en virgule fixe grande précision

Les variables en virgule flottante ou notation scientifique:

(**Single, Double**)

La variable peut être positive ou négative.

Le 'Double' est, bien sûr, plus précis et peut atteindre des nombres plus grand que le 'Single'.

Le 'Single' comporte 7 chiffres significatifs maximum.

Le nombre est **codé en interne** sous forme scientifique, exemple: `1,234568E+008`.

Mais en pratique, on travaille et on les affiche de manière habituelle, en notation normale avec **un point** comme séparateur décimal:

`Dim Poids As Single`

`Poids=45.45`

Format Scientifique, Mantisse et exposant:

Voici 3 nombres:

14500000

0,145

0,0000145

Ils comportent tous les 3, deux informations:

-le nombre entier 145

-la localisation du premier chiffre par rapport à la virgule

8

-1

-5 dans nos exemples.

Donc un réel peut être stocké sous la forme d'un couple:

-Partie entière

-Localisation de la virgule

Il est codé en interne avec une **mantisse (la partie entière)** et un **exposant** (position de la virgule), sous la forme `mmmEeee`, dans laquelle `mmm` correspond à la mantisse (chiffres significatifs: partie entières) et `eee` à l'exposant (puissance de 10).

En fait, en notation scientifique (en codage interne) un chiffre précède toujours la virgule: `1,234568E+008`.

Attention ,les variables numériques réelles ne peuvent pas contenir de nombre infiniment grand: Il y a une limite maximum comme pour les entiers. La valeur positive la plus élevée d'un type de données **Single** est `3,4028235E+38` et celle d'un type de données **Double** est `1,79769313486231570E+308`.

Quand on travaille avec des nombres ayant beaucoup de chiffres significatifs, il peut y avoir des erreurs d'arrondi. Le type 'Single' comporte par exemple une mantisse de 7 chiffres significatifs seulement.

Si on utilise des nombres (même petit: avec un exposant négatif par exemple) avec 8 chiffres significatifs il peut y avoir des erreurs arrondi.

Le type de données en Virgule fixe (Decimal) prend en charge jusqu'à 29 chiffres significatifs et peut représenter des valeurs jusqu'à $7,9228 \times 10^{28}$. Ce type de données est particulièrement adapté aux calculs (par exemple, financiers) qui exigent un grand nombre de chiffres, mais qui ne peuvent pas tolérer les erreurs d'arrondi.

Pour les calculs financiers on utilisera les 'Decimal' ou les 'Double'

Pour les petits calculs du genre résultats d'examen biologique on utilisera les 'Single' ou les 'Double' qui sont les plus rapides.

String, Char:

Le type 'String' peut contenir une 'chaîne de caractères' (alphanumérique) comme du texte. La longueur de la chaîne n'est pas fixe et une String peut avoir un nombre de caractère allant jusqu'à environ 2 milliards de caractères.

Les chaînes de longueur fixe n'existent pas (plus).

Le Type 'Char' contient un seul caractère. On utilise souvent des tableaux de 'Char'.

Pour information Char et String contiennent des caractères au format Unicode (dans la variable, chaque caractère est codé sur 2 octets) et pas de l'ASCII ou de l'ANSI..(ancien codage ou chaque caractère était codé sur un octet).

Les premiers caractères ont le même code Unicode et Ascii.

Exemple:

Caractère	Code
-----------	------

"a" 65

"b" 66

" " 32

Il y a aussi des caractères non affichable:

RC 13 retour chariot

LF 10 Line Feed

09 Tabulation

Pour passer à la ligne , on utilise les codes 13 puis 10.
Il y a une constante toute faites pour cela:

`ControlChars.CrLf` `Chr$(13)+Chr$(10)`

Annexe 1:Place occupée en mémoire:

Exemple de place occupée par une variable (et le nom de sa Classe dans le Common Langage Runtime).

Boolean 2 octets de **System.Boolean**

Byte 1 octet de **System.Byte**

Short 2 octets de **System.Int16**

Integer 4 octets de **System.Int32**

Long 8 octets de **System.Int64**

Single 4 octets de **System.Single**

Double 8 octets de **System.Double**

Decimal 16 octets de **System.Decimal**

Date 8 octets de **System.DateTime**

Char 2 octets de **System.Char**

Objet 4 octets de **System.Objet**

String dépend de la chaîne , de **System.String**

La méthode **GetType** permet de savoir de quel type, de quelle Classe est une variable.

`Dim x As String`

```
MessageBox.Show(x.GetType.ToString) 'Affiche:  
Systeme.String
```

Prend le type de x, le transforme en String, l'affiche dans une MessageBox.

Annexe 2: Type primitif

Mise à part **Objet**, **Structure**, **Class** tous les autres types sont dit '**Primitif**' (**Byte**, **Boolean**, **Shot**, **Integer**, **Long**, **Single**, **Double**, **Decimal**, **Date**, **Char**, **String**)

- Tous les types primitifs permettent la création de valeurs par l'écriture de **littéraux**. Par exemple, `i=123` ou `i=123I` (le I force 123 a être entier) est un littéral de type **Integer**.
- Il est possible de déclarer des **constantes** des types primitifs.
- Lorsque qu'une expression est constituée de constantes de type primitif, le compilateur évalue l'expression au moment de la compilation. C'est plus rapide.

Annexe 3: Choix des noms de variable

- La plupart des noms sont une concaténation de plusieurs mots, **utilisez des minuscules et des majuscules pour en faciliter la lecture**. Pour distinguer les variables et les routines (procédures), utilisez la casse Pascal (`CalculTotal`) pour les noms de routine (la première lettre de chaque mot est une majuscule). Pour les variables, la première lettre des mots est une majuscule, sauf pour le premier mot (`documentFormatType`).
- Les noms de variables booléenne doivent contenir **Is** qui implique les valeurs Yes/No ou True/False,

Exemple `fileIsFound`.

- Même pour une variable à courte durée de vie qui peut apparaître uniquement dans quelques lignes de code, utilisez un nom significatif. Utilisez des noms courts d'une seule lettre, par exemple `i` ou `j`, pour les index de petite boucle uniquement.
- N'utilisez pas des nombres ou des chaînes littérales telles que `For i = 1 To 7`. Utilisez plutôt des constantes nommées, par exemple `For i = 1 To Nombre_jour_dans_semaine`, pour simplifier la maintenance et la compréhension.
- Utilisez des paires complémentaires dans les noms de variables telles que `min/max`, `begin/end` et `open/close` ou des expressions `min max` si nécessaire en fin de nom.





1.6.1	Les string et Char	Suivant 	 Sommaire
--------------	---------------------------	---	--

Une variable **String** peut contenir une chaîne de caractères, des données alphanumériques.

Une variable **Char** peut contenir un caractère.

Pour travailler avec les 'String', on peut:

Utiliser les méthodes de la Classe **String**.

Utiliser les méthodes du **Basic** (les anciens reconnaîtrons)

STRING.

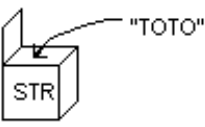
Il faut **déclarer** une variable avant de l'utiliser, pour cela on utilise l'instruction **DIM**

```
DIM Str As String
```

'Déclare une variable nommée **Str** et qui peut contenir une chaîne de caractère.

Cette variable peut être utilisée pour conserver une chaîne de caractère.

```
Str= "TOTO"
```



'On met la chaîne de caractères "TOTO" dans la variable **Str**

On peut afficher le contenu de la chaîne dans un label (zone présente dans une fenêtre et où on peut afficher du texte) par exemple :

```
Label.text = Str
```


Cela affiche 'TOTO'

Remarquons que pour définir une chaîne de caractères il faut utiliser des " " : Ce qui est entre " et " est la chaîne de caractères. On parle de **chaîne littérale**: (une représentation textuelle d'une valeur particulière)



Après avoir été créée une String contient NOTHING (pas une chaîne vide: ""), il faudra l'initialiser pour qu'elle contienne quelque chose.

```
DIM Str As String      'Str contient Nothing (pas le texte  
"Nothing" mais 'Rien')
```

```
Str= ""                'Str contient "" : chaîne vide
```

```
Str= "TOTO"           'Str contient "TOTO"
```

Notez bien l'importance des guillemets:

A est la variable A

"A" est une chaîne de caractères contenant le caractère "A"

Exemple:

```
Dim A As string= "Visual"
```

```
Dim B As string= "Basic"
```

```
Label.text = "A+B"    affiche bêtement la chaîne « A+B »
```

```
Label.text = A+B      affiche « Visual Basic » 'on affiche les  
variables.
```

Notez enfin que " ", l'**espace** est un caractère à part entière.

Si je veux inclure un caractère " dans la chaîne il faut le doubler pour qu'il ne soit pas considéré comme caractère de fin de chaîne:

```
A=" Bonjour ""Monsieur"" "    'Afficher cela donne : Bonjour "Monsieur"
```

On peut **initialiser** la variable en même temps qu'on la déclare.

```
Dim Chaîne as string = "Toto"
```

On peut déclarer plusieurs variables d'un même type sur une même ligne.

```
Dim x, y, z As String 'Déclare 3 variables 'String'
```

```
ListBox1.Items.Add(x.GetType.ToString)  
ListBox1.Items.Add(y.GetType.ToString)  
ListBox1.Items.Add(z.GetType.ToString)  
Affiche dans une liste box  
System.String  
System.String  
System.String  
Ce qui prouve que les 3 variables sont bien des Strings. (ce qui n'était pas le cas  
en VB6)
```

Les Variables 'chaîne de caractères' sont des objets 'STRING':

Le type **String** (Chaîne de caractères) est une Classe qui a des méthodes.



Pas besoin de connaître toutes les méthodes, il suffit (Après déclaration de la String par DIM Str AS String) de taper **Str.** .Et vous voyez apparaître toutes les propriétés et méthodes :

.Split
.StarsWith
..
.ToUpper
.Trim

Par exemple la méthode **.ToUpper**

Mettre en majuscules une chaîne de caractère

```
Str=Str.ToUpper()
```

Si Str contenait "abc" il contiendra "ABC"

.ToLower transforme par contre la chaîne en minuscule.

Quel intérêt ? Exemple

Je dois comparer 2 string pour savoir si elle sont égales, la première a été saisie par l'utilisateur et je ne sais pas si l'utilisateur a tapé en majuscule ou en minuscules.

Si je compare A = "Vb" et B= "vb" elles sont différentes.

Si je compare A.ToLower et B.ToLower elles sont égales.

.Trim

Permet de supprimer des caractères en début et fin de chaîne.

```
Dim A As String = "#@Informatique@#"
```

```
Dim B As Char() = {"#", "@"}
```

```
A=A.Trim(B)  Donne A= "Informatique"
```

Attention : Bien utiliser Char() qui est un tableau de caractères pour définir les caractères à supprimer.

Dim B As string= "#@" est déconseillé car produisant des résultats curieux. On peut à la rigueur utiliser les String pour un seul caractère.

Pour enlever les espaces avant et après la chaîne (Cas le plus fréquent) :

```
A=A.Trim(" ")
```

Il existe aussi **StartTrim** et **EndTrim** pour agir seulement sur le début ou la fin de la chaîne.

Length : Taille d'une chaîne en nombre de caractère.

Afficher la taille de la chaîne « VB »

```
Dim S as String= "VB"
```

```
MsgBox(S.Length.ToString) 'Affiche 2
```

Concat :

Concaténation de plusieurs chaînes : mise bout à bout :

```
S=String.Concat(A,B)
```

Il est plus rapide de faire : **S=A&B**

(S=A+B marche mais est déconseillé)

Insert :

Insère une chaîne dans une autre.

```
Dim S as string= "VisualBasic"
```

```
S= A.Insert(6,"  ") 'Donne S= "Visual Basic"
```

Replace :

Remplace partout dans une chaîne de départ, une chaîne par une autre.

```
Resultat=ChaineDépart.Replace(ChaîneAREmplacer,ChaîneQuiRemplace)
```

```
Dim S as string= "Visual_Basic"
```

```
S= S.Replace("_"," ") 'Donne S= "Visual Basic"
```

Autre exemple:

L'utilisateur a tapé une date, mais avec comme séparateur des ".", comme on le verra plus loin, il est nécessaire d'utiliser plutôt les "/", pour cela on utilise Replace

```
Dim LaDate as string= "12.02.1990"
```

```
LaDate= LaDate.Replace(".","/") 'Donne S= "12/02/1990"
```

Split :

Découpe en plusieurs sous Chaînes une chaîne de départ, cela par rapport à un séparateur.

Exemple :

Je récupère dans un fichier une chaîne de mots ayant pour séparateur « ; », je veux mettre chaque mot dans un tableau.

'Chaîne contenant les mots séparés par « ; »

```
Dim S as string= "Philippe;Jean ;Toto"
```

```
Dim Separateur as Char = ";"
```

```
Dim Nom() as String
```

```
Nom=S.Split(Separateur)
```

Donne :

Nom(0)= "Philippe"

Nom(1)= "Jean"

Nom(2)= "Toto"

Remarque: Quand on déclare le tableau Nom(), on ne donne pas le nombre d'élément, c'est Split qui crée autant d'élément qu'il faut.

.IndexOf .LastIndexOf

Indique le numéro du caractère, la position (la première occurrence) ou une chaîne à chercher est trouvée dans une autre.

```
Dim A as String= "LDF.EXE"
```

```
Dim R as Char()={"."}
```

```
A.IndexOf(R)  retourne 3
```



Se souvenir : le premier caractère est en position 0.

.LastIndexOf retourne la dernière occurrence.

.Compare

Compare 2 chaînes :

```
String.Compare(a,b)
```

Retourne un entier

Négatif si a<b

0 si a=b

Positif si a>b

.Substring

Extrait une partie d'une chaîne

```
Dim A as string= "Informatique"
```

```
MessageBox.show(A.Substring(2,3)) 'Affiche for
```

Le premier paramètre indique la position du caractère où doit commencer la sous-chaîne, en commençant à la position 0. (les caractères sont comptés 0, 1, 2,

3...; on verra qu'avec mid la sous-chaîne, commence à la position 1.

Le second paramètre la longueur de la sous-chaîne.

.Chars

Une chaîne peut être perçue comme un **tableau de caractères** (instances Char) ; vous pouvez extraire un caractère particulier en faisant référence à l'index de ce caractère par l'intermédiaire de la propriété Chars. Par exemple :

```
Dim maString As String = "ABCDE"  
Dim monChar As Char  
monChar = maString.Chars(3) ' monChar = "D"
```

On peut créer des chaînes avec la Classe String:

```
MyString = New String(" ", 15) 'Créer une chaîne de 15 espaces
```

.PadRight

Aligne les caractères de cette chaîne à gauche et remplit à droite en ajoutant un caractère Unicode spécifié pour une longueur totale spécifiée.

```
Dim str As String  
Dim pad As Char  
str = "Nom"  
pad = Convert.ToChar(".")  
Console.WriteLine(str.PadRight(15, pad)) ' Affiche  
Nom.....
```

PadLeft fait l'inverse.

On peut aussi utiliser les méthodes Visual Basic:

Elles sont bien connues des 'anciens' et font partie intégrante du langage vb (Common Language Runtime) et sont parfois plus simples.

Elle font partie de l'espace de nom Microsoft.VisualBasic mais il est 'chargé' par défaut et il n'y a pas lieu de l'importer. Par contre quand certaines propriétés sont communes à plusieurs classes, il peut y avoir ambiguïté et il faut utiliser dans ce cas la syntaxe complète. Cela semble le cas pour left qui est un mot clé Vb mais aussi une propriété des contrôles. Pour lever l'ambiguïté il faut écrire `Microsoft.VisualBasic.Left(C,i)` par exemple.

Ces méthodes font souvent double emploi avec les méthodes de la classe String:

Mid:

Permet de récupérer une sous-chaîne.

```
MaString = "Mid Demonstration"  
A = Mid(MaString, 1, 3) ' Retourne "Mid".
```

Retourne 3 caractères à partir du premier

Le premier paramètre indique la position du caractère où doit commencer la sous-chaîne, en commençant à la position 1. (les caractères sont comptés 1, 2, 3...; on rappelle qu'avec SubString la sous-chaîne, commence à la position 0.

```
A = Mid(MaString, 14) ' Retourne "tion": du 14ème à la fin (pas de 3ème argument)
```

Mid permet aussi de remplacer une string dans une string

```
Mid(MaString, 1, 3) = "Fin" => MaString="Fin Demonstration"
```

Left, Right

Retourne x caractères de gauche ou de droite:

```
A=Right(MaString,2) 'A="on"
```

```
A=Microsoft.VisualBasic.Left(MaString,2) 'A="Mi"
```

Notez bien que, pour lever toute ambiguïté avec les méthodes left d'autres classes, il faut indiquer `Microsoft.VisualBasic.Left`.

Len.

Retourne la longueur de la chaîne:

```
MyLen = Len(MaString) ' Retourne 17.
```

LTrim, RTrim

Enlève les espaces à gauche ou à droite d'une chaîne.

```
A=LTrim(" RRRR") ' A="RRR"
```

InStr

Retourne un entier spécifiant la position de début de la première chaîne à l'intérieur d'une autre.

```
N=InStr(1,"aaaaRaa","R") 'retourne 5
```

Rechercher à partir du premier caractère, à quelque position se trouve 'R' dans la chaîne "aaaaRaa"

Si la chaîne n'est pas trouvée , retourne 0

InStrRev

Recherche aussi une chaîne mais de droite à gauche. la position de départ est le 3ème argument.

```
InStrRev (Ch1, Ch2 , PosDépart)
```

StrComp Compare 2 chaînes.

Space

Retourne une chaîne d'espace: `Space(10)` retourne " "

StrDup

Retourne une chaîne de caractères par duplication d'un caractère dont on a spécifié le nombre.

```
maString = StrDup(5, "P") ' Retourne "PPPPP"
```

Asc

Retourne le code de caractère du caractère entré. Il peut être compris entre 0 et 255 pour les valeurs du jeu de caractères codé sur un octet (SBCS) et entre -32 768 et 32 767 pour les valeurs du jeu de caractères codé sur deux octets (DBCS). La valeur retournée dépend de la page de codes

AscW retourne le code Unicode du caractère entré. Il peut être compris entre 0 et 65 535.

```
x=Asc("A") 'retourne 65
```

```
x=Asc("ABCD") 'retourne 65 : Seul le premier caractère est pris en compte
```

Chr et ChrW

Retourne le caractère associé au code de caractère.

```
Chr(65) retourne "A" 'cela dépend de la page de code.
```

ChrW retourne le caractère correspondant à l'Unicode

GetChar

Retourne le caractère d'une chaîne à une position donnée.

```
Dim maString As String = "AIDE"  
Dim monChar As Char  
monChar = GetChar(maString, 3) ' monChar = "D"
```

LCase Ucase

Retourne la chaîne en minuscule ou majuscule:

```
Lowercase = LCase(UpperCase)
```

Lset Rset

Retourne une chaîne alignée à gauche avec un nombre de caractère.

```
Dim maString As String = "gauche"  
Dim R As String  
R = LSet(maString, 2) ' Retourne "ga"
```

Si la chaîne de départ est plus courte que la longueur spécifiée, des espaces sont ajoutés.

```
R = LSet(maString, 8) ' Retourne "gauche "
```

StrRevers

Retourne une chaîne ou les caractères ont été inversés:

```
Dim maString As String = "STRESSED"
```



```
Dim revString As String
```

```
revString = StrReverse(myString) ' Retourne "DESSERTS"
```

Marrant l'exemple!!

Combinaison de chaînes de caractères, des variables..

Souvent, on a besoin d'afficher une combinaison de chaînes littérales, le contenu de variables, des résultats de calcul; c'est possible

Exemple :

Pour afficher dans un label le carré de X est X², avec x variable ayant une valeur :

```
Label1.text= "Le carré de " & X & " est " & X * X
```

Ce qui est entre guillemets est affiché tel quel. C'est le cas de "Le carré de " et de " est "

Ce qui n'est pas entre guillemets est évalué, le résultat est affiché. C'est le cas de X et X*X

Pour ne faire qu'une chaîne on ajoute les bouts de chaînes avec l'opérateur '&'.

Notez l'usage d'espace en fin de chaîne pour ne pas que les mots et les chiffres se touchent.

```
Dim X As Integer
```

```
X=2
```

```
Label1.text= "Le carré de " & X & " est " & X * X
```

affiche dans le label : « Le carré de 2 est 4 »

Exemple simple:

Une string 'Nom' contient un nom , mettre si nécessaire la première lettre en majuscule.

En utilisant les Classes String et Char:

```
Dim Nom As String = "philippe"
```

```
' Si le premier caractère est minuscule
```

```
If Char.IsLower(Nom.Chars(0)) Then
```

```
    ' Le transformer en majuscule et afficher
```

```
MessageBox.Show(Char.ToUpper(Nom.Chars(0)) + x.Substring(1, Nom.Length - 1))
```

End If

On regarde si le premier caractère de la chaîne Nom.Chars(0) est minuscule (.IsLower).

Si oui on transforme ce premier caractère en majuscule (.ToUpper) et on ajoute la sous chaîne allant du second au dernier caractère.

En utilisant les instructions VB:

```
Dim Nom As String = "philippe"
```

```
Nom = UCase(Microsoft.VisualBasic.Left(Nom, 1)) & Mid(Nom, 2)
```

On prend le premier caractère de gauche : Left(Nom,1), on le transforme en majuscule (Ucase) , on ajoute la chaîne commençant par le second caractère et allant jusqu'à la fin.

Like:

Instruction **hyper puissante**: Like, elle **compare** une chaîne String avec un modèle (Pattern), elle permet de voir si la chaîne contient ou ne contient pas un ou des caractères, ou une plage de caractères.

```
result = String Like Pattern
```

Si *string* correspond à *pattern*, la valeur de *result* est **True** ; s'il n'y a aucune correspondance, la valeur de *result* est **False**. Si *string* et *pattern* sont une chaîne vide, le résultat est **True**. Sinon, si *string* ou *pattern* est une chaîne vide, le résultat est **False**.

L'intérêt de **Like** est que l'on peut y mettre des **caractères génériques**:

? veut dire tout caractère unique

* veut dire * ou plusieurs caractères.

veut dire tout chiffre.

[caractères] veut dire tout caractères présent dans la liste.

[!caractères] veut dire tout caractères NON présent dans la liste.

- **trait d'union** permet de spécifier un début et une fin de plage.

Exemple:

```
Dim R As Boolean  
R = "D" Like "D" ' Est ce que "D" est égal à "D"? => True.
```

```
R = "F" Like "f" ' Est ce que "F" est égal à "f"? => False.
```

```
R = "F" Like "FFF" ' Est ce que "F" est égal à "FFF"? => False.
```

`R = "cBBBc" Like "c*c"` ' Est ce que "cBBBc" répond au pattern (avoir un "c" au début, un "c" à la fin, et des caractères au milieu? Retourne True.

`R = "J" Like "[A-Z]"` ' Est ce que "J" est contenu dans les caractères allant de ' A à Z? Retourne True.

`R = "I" Like "[!A-Z]"` ' Est ce que "I" n'est PAS dans les caractères allant de ' A à Z? Retourne False.

`R = "a4a" Like "a#a"` ' Est ce que "a4a" commence et finie par un ' "a" et à un nombre entre les 2? Retourne True.

`R = "bM6f" Like "b[L-P]#[!c-e]"` ' Est ce que "bM6f"
'commence par "b",
'a des caractères entre L et P
'un nombre
'se termine par un caractère non compris entre c et e
'retourne True

Option Compare

L'ordre des caractères est défini par **Option Compare** et les paramètres régionaux du système sur lequel s'exécute le code. En utilisant **Option Compare Binary**, la plage [A–E] correspond à A, B, C, D et E. Avec **Option Compare Text**, [A–E] correspond à A, a, À, à, B, b, C, c, D, d, E et e. La plage ne correspond pas à Ê ou ê parce que les caractères accentués viennent après les caractères non accentués dans l'ordre de tri.

Unicode:

Les variables **string** sont stockées sous la forme de séquences de 16 bits (2 octets) non signés dont les valeurs sont comprises entre 0 et 65 535. Chaque nombre représente un caractère **Unicode**. Une chaîne peut contenir jusqu'à 2 milliards de caractères.

Les premiers 128 codes (0–127) Unicode correspondent aux lettres et aux symboles du clavier américain standard. Ce sont les mêmes que ceux définis par le jeu de caractères ASCII. **Les 128 codes suivants** (128–255) représentent les caractères spéciaux, tels que les lettres de l'alphabet latin, les accents, les symboles monétaires et les fractions. Les codes restants sont utilisés pour des symboles, y compris les caractères textuels mondiaux, les signes diacritiques, ainsi que les symboles mathématiques et techniques.

CHAR.

Les variables **Char** sont stockées sous la forme de nombres 16 bits (2 octets) non signés dont les valeurs sont comprises entre 0 et 65 535. Chaque nombre représente un seul caractère **Unicode**. Les conversions entre le type **Char** et les types numériques sont impossibles, mais il y a les fonctions **AscW** et **ChrW** peuvent être utilisées..

L'ajout du caractère de type littéral **C** à un littéral de chaîne force ce dernier à être un type **Char**. A utiliser surtout si **Option Strict (qui force à être strict..)** est activé.

Exemple:

```
Option Strict On
```

```
' ...
```

```
Dim C As Char
```

```
C = "A" C
```

ToCharArray:

Permet de passer une string dans un tableau de Char:

```
Dim maString As String = "abcdefghijklmnop"
```

```
Dim maArray As Char() = maString.ToCharArray
```

La variable maArray contient à présent un tableau composé de **Char**, chacune représentant un caractère issu de maString.

Chars:

vous pouvez extraire un caractère particulier en faisant référence à l'index de ce caractère par l'intermédiaire de la propriété Chars. Par exemple :

```
Dim maString As String = "ABCDE"
```

```
Dim monChar As Char
```

```
monChar = maString.Chars(3) ' monChar = "D"
```

Un caractère est-il numérique? un chiffre? une lettre? un séparateur? un espace?....

```
Dim chA As Char
chA = "A" c
Dim ch1 As Char
ch1 = "1" c
Dim str As String
str = "test string"

Console.WriteLine(chA.CompareTo("B" c)) '
Output: "-1" ' A est plus petit que B
Console.WriteLine(chA.Equals("A" c)) '
Output: "True" ' Egal?
Console.WriteLine(Char.GetNumericValue(ch1)) '
Output: "1" ' Convertir en valeur numérique (double)
Console.WriteLine(Char.IsControl(Chr(9))) '
Output: "True" ' Est une caractère de contrôle?
Console.WriteLine(Char.IsDigit(ch1)) '
Output: "True" ' Est un chiffre
Console.WriteLine(Char.IsLetter(", " c)) '
Output: "False" ' Est une lettre
Console.WriteLine(Char.IsLower("u" c)) '
Output: "True" ' Est en minuscule
Console.WriteLine(Char.IsNumber(ch1)) '
Output: "True" ' Est un nombre
Console.WriteLine(Char.IsPunctuation(".", " c)) '
Output: "True" ' Est un caractère de ponctuation
Console.WriteLine(Char.IsSeparator(str, 4)) '
Output: "True" ' Est un séparateur
Console.WriteLine(Char.IsSymbol("+ " c)) '
Output: "True" ' Est un symbole
Console.WriteLine(Char.IsWhiteSpace(str, 4)) '
```

Output: "True" ' Est un espace
Console.WriteLine(Char.ToLower("M"c))

Output: "m" ' Passe en minuscule

Existe aussi `IsLetterOrDigit`, `IsUpper`.

Bien sur si on est en 'Option Strict' il faut ajouter
.ToString à chaque ligne.

On note que l'on peut tester un caractère dans une chaîne:
`Char.IsWhiteSpace(str, 4)`

Allons plus loin avec les Chaînes de longueur fixe

On a vu que les chaînes de longueur fixe n'existent pas en VB.NET (compatibilité avec les autres langages oblige), mais il y a moyen de contourner le problème:

On peut utiliser la **Classe de compatibilité VB6**:

(Il faut charger dans les références du projet
Microsoft.VisualBasic.Compatibility et CompatibilityData)

```
Dim MaChaineFixe As New VB6.FixedLengthString(100)
```

Pour afficher la chaîne fixe utilisez `MaChaineFixe.ToString`

Mais pour mettre une chaîne dans cette chaîne de longueur fixe!! galère!!!

`MaChaineFixe="ghg"` n'est pas accepté: on ne peut pas mettre une String dans une chaîne fixe

```
MaChaineFixe = CType("hg",  
Microsoft.VisualBasic.Compatibility.VB6.FixedLengthString) 'pas accepté non plus!!
```

Enfin ce type de chaîne fixe ne peut pas être utilisée dans les structures, mais il y a un autre moyen pour les structures. On verra cela plus loin.

Donc à priori, les chaînes fixes sont à éviter.



1.6.2 Variables numériques.

Suivant

Sommaire

Une variable numérique peut contenir des données numériques.

On a vu qu'une variable numérique peut être entière:

- q Integer (entier)
- q Short (entier court)
- q Long (Entier long)
- q Byte (entier de 0 à 255)

mais aussi

- q Single (virgule flottante simple précision)
- q Double (virgule flottante double précision)
- q Decimal (virgule fixe haute précision)

On déclare une variable numérique avec **DIM**, on peut l'initialiser en même temps :

```
Dim i As Integer= 3
```

Si la variable est numérique, il faut la transformer en String avant de l'afficher :

Dim I As Integer=12

Label.text = I.ToString

.ToString fait partie des méthodes. Il y en a d'autres:

.GetType retourne le type de la variable

```
Dim i As Integer
```

```
Label1.Text = i.GetType.ToString 'Affiche: System.Int32
```

.MaxValue **.MinValue** donne le plus grand et le plus petit nombre possible dans le type de la variable.

On verra qu'on peut utiliser des opérateurs + - * / .

```
Dim I As Integer=2
```

```
Dim J As Integer
```

```
J=I+3
```

' J est égal à 5 car on affecte à J la valeur I+3

On rappelle que **le séparateur est le point :**

```
J=1.2
```

 veut dire J=1,2 en bon français !!

Il existe des **fonctions mathématiques**.

Pour qu'elles soient disponibles il faut d'abord importer l'**espace de nom 'Math'** :

Pour cela il faut taper en haut de la fenêtre (au dessus de public Class)

```
Imports System.Math
```

On verra plus loin ce que cela signifie.

```
Dim N As Single
```

```
Dim R As Single
```

```
R=Abs(N) 'retourne la valeur absolu
```

```
'Si N=-1.2 R=1.2
```

```
R=Sign(N) 'retourne le signe
```

```
'Si N=-1.2 R=-1 (négatif) ; retourne  
1 si nombre positif
```

```
R=Round(N) 'retourne le nombre entier le plus  
proche
```

```
' N=1.7 R=2
```

```
' N=1.2 R=1
```

```
' N=1.5 R=2
```

On peut donner en second paramètre le nombre de digit:
`Math.Round(Valeur, 2)` donne 2 décimales après la virgule.

```
R=Floor(N) 'retourne le plus grand entier égal ou  
inférieur.
```

```
' N=1.7 R=1
```

```
R=Ceiling(N) 'retourne le plus petit entier égal  
ou supérieur
```

```
' N=1.2 R=2
```

```
R=Max(2,3) 'retourne le plus grand des 2 nombres.
```

```
'retourne 3
```

```
R=Min(2,3) 'retourne le plus petit des 2 nombres.
```


'retourne 2

`R=Pow(2,3)` 'retourne 2 puissance 3.

'retourne 8

`R=Sqrt(9)` 'retourne la racine carré.

'retourne 3

Il existe aussi les instructions du langage VB comme:

Int et **Fix** qui suppriment toutes deux la partie fractionnelle et retournent l'entier. Si *le nombre* est négatif, **Int** retourne le premier entier négatif inférieur ou égal au nombre, alors que **Fix** retourne le premier entier négatif supérieur ou égal au nombre. Par exemple, Int convertit -8,4 en -9 et Fix convertit -8,4 en -8.

`R=Int(8.4)` R=8

Bien sur il y a aussi **Sin Cos Tan, Sinh Cosh Tanh** (pour hyperbolique) **Asin Acos Atan Atan2**.

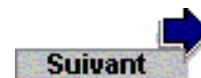
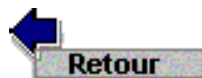
Prenons un exemple:

```
Imports System.Math
Dim MonAngle, MaSecant As Double
MonAngle = 1.3 ' angle en radians.
MaSecant = 1 / Cos(MonAngle) ' Calcul la sécante.

```

On remarque que les angles sont en **radians**.

Rappel: $2\pi=360^\circ$; Angle en radians= $(2\pi/360)*\text{Angle en degrés}$



1.6.3 Conversion.

Suivant

Sommaire

On a vu qu'on peut afficher les chaînes de caractères, par ailleurs, on fait des calculs avec les variables numériques.

Est-il possible de convertir une variable d'un type à un autre?

Conversion numérique=>String

Quel intérêt de convertir?

On veut afficher un résultat numérique.

On ne peut afficher que des **String** (chaîne de caractères) dans un label ou un TextBox par exemple.

Aussi, il faut transformer cette valeur numérique en chaîne avant de l'afficher, on le fait avec la méthode **ToString**:

```
Dim I As Integer=12           'On déclare une  
variable I qu'on initialise à 12
```

```
Label.text = I.ToString
```

La valeur de I est transformée en String puis affectée à la propriété text du label, ce qui affiche '12'

Conversion String=>numérique

A l'inverse une chaîne de caractère peut être transformée en numérique :

Par exemple, l'utilisateur saisie un nombre dans une boîte de saisie (InputBox), mais il tape des caractères au clavier et c'est cette chaîne de caractères qui est retournée, il faut la transformer en numérique.

```
Dim S as String
```

```
Dim i as Integer
```

```
S= InputBox ("Test", "Taper un nombre") 'Saisie dans une  
InputBox d'un nombre par l'utilisateur.
```

'S contient maintenant une chaîne de caractères,
"45" par exemple

```
I=Integer.Parse(S) 'on transforme la chaîne S  
en Integer
```

Bizarre cette syntaxe!! en fait c'est le type
Integer qui a une méthode (Parse) qui transforme
une chaîne en entier.

On peut aussi utiliser, et c'est plus simple,
CType pour convertir n'importe quel type en
n'importe quel type :

Il suffit de donner à cette fonction la variable à
modifier et le type à obtenir.

```
I=Ctype(S,Integer)
```

CType pour toutes les conversions

Ctype peut aussi servir à convertir de la même manière un
single en double, un Short en Integer....

Il est donc possible de convertir un type de variable en un
autre.

Il suffit de donner à cette fonction la variable à

modifier et le type à obtenir.

```
I=Ctype(S,Integer) 'conversion en entier
```

CType fait toutes les conversions, mais on peut aussi utiliser des fonctions qui sont **spécifiques au type de la variable de retour**: Le nom de ces fonctions contient le nom du type de la variable de retour.

```
CBool()  
CByte()  
CChar()  
CDate()  
Cdbl()  
CDec()  
CInt()  
CLng()  
CObj()  
CShort()  
CSng() Pour convertir en Single  
CStr()
```

Exemple **Cdbl** retourne un 'Double'.

```
Dim I As Integer=123
```

```
Dim D As Double
```

```
D=Cdbl(I) 'donnera D=123 D est un Double (réel double  
précision)
```

Ces fonctions sont plus **rapides** car elles sont spécifiques.

Remarque:

Les fonctions **CInt** et **CLng** arrondissent les parties décimales égales à 0,5 au **nombre pair** le plus proche. Par exemple, 0,5 s'arrondit à 0 et 1,5 s'arrondit à 2. Bizarre!!

Val et Str existe aussi:

Ouf pour les anciens!!

Ces fonctions permettent aussi la **conversion String=>Numérique et Numérique=>String**

Val donne la valeur numérique d'une expression String.

```
Dim i As Integer
```

`i=Val("5")` ' i=5

Val s'arrête au premier caractère non numérique.

`Val("12er")` retourne 12

Val reconnaît le point (et pas la virgule)

`Dim i As Double`

`i=Val("5.45")` ' donnera i=5,45

`i=Val("5,45")` ' donnera i=5

Str transforme une valeur numérique en String:

`Dim s As String`

`s=Str(1999)` ' s=" 1999"

Noter bien: Str ajoute un espace à gauche ou le signe '-' si le nombre est négatif.

Str ne reconnaît que le point comme séparateur décimal. (Pour utiliser les autres séparateurs internationaux, il faut utiliser la fonction CStr()).

Autre:

La Classe **System.Convert** permet la conversion d'un type de base vers un autre:

`.ToString` en fait partie

Exemple

Pour convertir **un Single en Byte** (entier 8 bits non signé)

`.ToByte`

Pour convertir **un Byte en Single**:

`.ToSingle`

`singleVal = System.Convert.ToSingle(byteVal)`

en **Decimal**

`.ToDecimal`

On a des méthodes pour pratiquement convertir tous les types en tous les types. Chercher!!

On verra plus loin, la fonction **Format** utilisée pour convertir les valeurs numériques que vous voulez mettre aux formats dates, heures ou monnaie ou dans d'autres formats définis par l'utilisateur.

IsNumeric

On utilise la fonction **IsNumeric** pour déterminer si le contenu d'une variable peut être évalué comme un nombre.

Exemples:

```
Dim MyVar As Object
Dim R As Boolean

MyVar = "45"
R = IsNumeric(MyVar) ' R= True.
' ...
MyVar = "678.92"
R = IsNumeric(MyVar) ' R= True.
' ...
MyVar = "45 Kg"
R = IsNumeric(MyVar) ' R= False.
```

'Attention le dernier exemple indique que "45 Kg" n'est pas purement numérique, mais `Val("45 Kg")` retourne 45 sans déclencher d'erreur car `Val` transforme les caractères numérique à partir de la gauche, en s'arrêtant dès qu'il y a un caractère non numérique.



Retour



Index



Sommaire



Suivant

Site **LDF** Cours **VB.net**



1.6.4	Les tableaux		
		Suivant	Sommaire

Ils permettent de regrouper des données de même type.

Les tableaux vous permettent de faire référence à un ensemble de variables par le même nom et d'utiliser un numéro, appelé *index* ou *indice*, pour les distinguer.

Comment déclarer un tableau :

```
Dim Tableau(3) As Integer      'déclare un
tableau de 4 entiers
```

On remarque que, dès la déclaration du tableau, **le nombre d'éléments est bien défini et restera toujours le même.**

`Dim Tableau(3) As Integer` entraîne la **création** des variables 'Integer' suivante:

Tableau (0)

Tableau (1)

Tableau (2)

Tableau (3)

0
0
0
0

soit 4 éléments

Noter que comme c'est un tableau d'entier, juste après la création du tableau les éléments sont initialisés à 0.



Le tableau commence toujours par l'indice

0

Le nombre d'éléments dans le tableau est toujours égale à l'indice de dimension + 1 (ou l'indice du dernier élément+1)

`Dim a(150)` comporte 151 éléments (éléments d'index 0 à 150).

`Tableau(1)= 12` permet d'affecter le nombre 12 au 2eme élément du tableau.

0
12
0
0

`S=Tableau(1)` permet d'affecter à la variable S le 2eme élément du tableau.

Un tableau peut avoir **plusieurs dimensions** :

`Dim T(2,2)` 3 X 3 éléments

Pour un tableau à 2 dimensions le premier argument représente les lignes, le second les colonnes.

0,0	,1	0,2
1,0	,1	1,2
2,0	,1	2,2

Exemple:

La première ligne comporte les 3 éléments:
T(0,0) T(0,1) et T(0,2)

Il est possible de créer des tableaux à 3, 4 ..dimensions:

```
Dim T(2,2,2) 3X3X3 éléments.
```

On peut créer des tableaux de tableaux:

```
Dim T(2),(2) Il a autant d'élément que le tableau T(2,2)
```

Il est possible de **créer des tableaux avec tous les types de variable** (y compris les structures)

```
Dim Mois(12) As String 'tableau de String
```

Notez que dans ce cas les éléments contiennent Nothing car le tableau contient des String et quand on déclare une String, elle contient Nothing au départ.

On peut **initialiser un tableau** (Donner une valeur aux éléments):

```
Dim Mois(12) As String
```

```
Mois(0)="Janvier"
```

Mois(1)="Février"

Mois(2)="Mars"

ou lors de sa déclaration :

```
Dim Mois() As String = {Janvier, Février, Mars.....}
```

On verra dans un des exemples que l'on peut créer un tableau avec la méthode **CreateInstance**.

Redim permet de redimensionner un tableau (modifier le nombre d'éléments d'un tableau existant), si on ajoute **Preserve** les anciennes valeurs seront conservées.

Attention , on ne peut pas modifier le nombre de dimension, ni le type des données. Un tableau à 2 dimensions de 20 fois 20 string pourra être redimensionné en tableau de String 30 fois 30, mais pas en tableau d'entiers ou à 3 dimensions.

```
Dim T(20,20) As String
```

...

```
Redim Preserve T(30,30)
```

Il est possible d'écrire Dim T(,) As String

`Dim T(,) As String` 'Sans donner les dimensions du tableau: il est déclaré mais n'existe pas car `T(1,1)="toto"` déclenche une erreur. Il faut avant de l'utiliser écrire `Redim T(30,30)`, (sans remettre As String).

Certaines instructions comme Split redimensionne elle-même le tableau au nombre d'élément nécessaire.

```
Dim Nom() as String
```

Nom=S.Split(Separateur)

Erase efface le tableau et récupère l'espace.

Comment parcourir un tableau?

Pour parcourir un à un tous les éléments d'un tableau, on utilise **une boucle**:

Exemple: créer un tableau de 11 éléments et mettre 0 dans le premier élément, 1 dans le second, 2 dans le troisième...

```
Dim T(10) As Integer
Dim i As Integer
```

```
For i = 0 To 10
    T(i)=i
Next i
```

La variable de boucle *i* est utilisée pour parcourir le tableau: on utilise l'élément *T(i)* donc successivement *T(1)* puis *T(2)*...et on affecte *i* donc 1 puis 2 puis 3..

On peut aussi utiliser **For Each**:(un tableau étant ici considéré comme une collection)

```
Dim amis() As String = {"pierre", "jean", "jacques",
    "toto"}
For Each nom As String In amis
    Console.Out.WriteLine(nom)
Next
```

L'exemple affiche sur la console (menu Affichage->Fenêtre->Sortie) les noms qui sont dans le tableau.

Un tableau est un objet !!

Créons 2 tableaux et examinons les principales méthodes.

Dim a(3) As String

Dim b(3) As String

b=a 'Copie le tableau a dans b

b=a.copy 'Est équivalent

Attention: il copie les références (l'adresse, l'endroit où se trouve la variable) et non pas la valeur de cette variable, ce qui fait que si vous modifiez b(3), a(3) sera aussi modifié.

Car lorsque vous assignez une variable tableau à une autre, seul le pointeur (l'adresse en mémoire) est copié.

Pour obtenir une copie 'indépendante' faire :

b=a.clone

Dans ce cas si vous modifiez a(2), b(2) ne sera pas modifié.

Par contre a(1)=b(1) n'affecte que l'élément a(1)

La Classe Array.

Tous les tableaux viennent de la classe **Array**; vous pouvez accéder aux méthodes et propriétés de **System.Array** de chaque tableau. Par exemple, la propriété **Rank** retourne le nombre de dimension du tableau et la méthode **Sort** trie ses éléments.

Exemple :

Soit un tableau Mois()

Clear

`Array.Clear(Mois,0,2)` 'Efface 2 éléments du tableau Mois à partir de l'élément 0

Reverse

`Array.Reverse(Mois, 1, 3)` inverse les 3 éléments à partir de l'élément 1

Copy

`Array.Copy(Mois,1,Mois2,1,20)` copie 20 éléments de Mois vers Mois2 à partir du 2eme élément.

Sort

`Array.sort(Mois)` Trie le tableau Mois

Malheureusement cette méthode marche sur des tableaux unidimensionnels uniquement.

Au lieu d'utiliser un tableau à 2 dimensions (sur lequel la méthode sort ne marche pas, on peut ruser et créer 2 tableaux et surchargé la méthode sort pour trier les 2 tableaux (un servant de clé, le second d'items):

`Array.Sort(myKeys, myValues)` (Voir un exemple plus bas)

Equals compare 2 tableaux.

Binarysearch recherche un élément dans un tableau **trié** unidimensionnel.(algorithme de comparaison binaire performant sur tableau **trié**)

Exemple :

`I=Array.BinarySearch(Mois, "Février")`

'retourne I=1 se souvenir le premier élément est Mois(0)

IndexOf

Recherche un objet spécifié dans un tableau unidimensionnel (trié ou non), retourne l'index de la première occurrence.

```
Dim myIndex As Integer = Array.IndexOf(myArray, myString)
```

Retourne -1 si l'élément n'est pas trouvé.

LastIndexOf fait une recherche à partir de la fin.

Ubound

Retourne le plus grand indice disponible pour la dimension indiquée d'un tableau

```
Dim Indice, MonTableau(10, 15, 20)
Indice = UBound(MonTableau, 1) ' Retourne 10. (1 indique la première dimension du tableau)
```

GetUpperBound même fonction

```
Indice = MonTableau.GetUpperBound(0) '( 0 pour première dimension!!) Retourne 10.
```

Lbound existe (plus petit indice) mais est inutile car toujours égal à 0.

Length retourne un entier qui représente le nombre d'éléments dans le tableau.

GetValue et **SetValue** permettent de connaître ou de modifier la valeur d'un élément du tableau:

```
Mois.GetValue(0) est équivalent à Mois(0)
```

Dans un tableau à 2 dimensions comment modifier l'élément (0,3):

```
myArray.SetValue("fox", 0, 3)
```

Exemple:

Exemple détaillé:

Créer un tableau de 6 éléments, mettre dans chaque élément du tableau le carré de son indice, afficher le contenu du tableau.

Cela montre l'intérêt d'utiliser une boucle pour balayer tous les éléments d'un tableau. Première boucle pour remplir le tableau, seconde boucle pour afficher. (Une boucle For ..Next est ici utilisée, on verra cela plus loin.)

```
Dim arr(5) As Integer
Dim i As Integer
For i = 0 To arr.GetUpperBound(0)
    arr(i) = i * i
Next i

For i = 0 To arr.GetUpperBound(0)
    Console.WriteLine("arr(" & i & ") = " & arr(i))
Next i
```

Faire une boucle allant de 0 au dernier élément du tableau (For i=0 to ..)

Dans chaque élément du tableau mettre le carré de son indice (arr(i)=i*i)

Nouvelle boucle pour afficher les noms des différents éléments et leur contenu. (Console.WriteLine() affiche sur la console le nom de l'élément et son contenu)

Le programme génère la sortie suivante :

```
arr(0) = 0
```

```
arr(1) = 1
```

```
arr(2) = 4
```

```
arr(3) = 9
```

```
arr(4) = 16
```

```
arr(5) = 25
```

Exemple de recherche dans un tableau:

Dans un tableau de String rechercher dans quel élément et à quelle position se trouve la string "MN".

```
Dim Tableau() As String = {"ABCDEFGH",  
    "HIJKLMNPO"}  
Dim AChercher As String = "MN"  
Dim i As Integer  
Dim position As Integer  
For i = 0 To Tableau.Length - 1  
    position = Tableau(i).IndexOf(AChercher)  
    If position >= 0 Then Exit For  
Next i
```

Exemple de tri de 2 tableaux:

On crée un tableau de clé et un tableau des valeurs, à chaque clé est lié une valeur.

On trie à partir du tableau des clé myKeys, le tableau myValues est modifié pour 'suivre' le tri des clé. La Sub PrintKeysAndValues affiche les résultats.

```
Public Shared Sub Main()  
    ' *****Création des tableaux.  
    Dim myKeys() As String = {"red", "GREEN", "YELLOW",  
        "BLUE", "purple", "black", "orange"} 'Tableau des clé  
    Dim myValues() As String = {"strawberries", "PEARS",  
        "LIMES", "BERRIES", "grapes", "olives", "cantaloupe"}
```



```
'tableau des éléments
```

```
'Affichage du tableau non trié
```

```
Console.WriteLine("Tableau non trié:")
```

```
PrintKeysAndValues(myKeys, myValues)
```

```
' Tri les éléments 1 à 3 puis affichage.
```

```
Array.Sort(myKeys, myValues, 1, 3)
```

```
Console.WriteLine("Après tri d'une partie du tableau:")
```

```
PrintKeysAndValues(myKeys, myValues)
```

```
    ' Tri la totalité du tableau.
```

```
Array.Sort(myKeys, myValues)
```

```
Console.WriteLine("Après tri de la totalité du tableau:")
```

```
PrintKeysAndValues(myKeys, myValues)
```

```
End Sub 'Fin de Main
```

```
    ' Routine affichant dans la console les clés et  
valeurs
```

```
Public Shared Sub PrintKeysAndValues(ByVal myKeys() As  
[String], ByVal myValues() As [String])
```

```
Dim i As Integer
```

```
For i = 0 To myKeys.Length - 1
```

```
    Console.WriteLine(" {0,-10}: {1}", myKeys(i),  
myValues(i))
```

```
Next i
```

```
Console.WriteLine()
```

```
End Sub 'PrintKeysAndValues
```

Création de tableau avec CreateInstance

```
' Créons un tableau d'entier (Int32) comprenant 5  
éléments.
```

```
Dim myArray As Array =
```

```
Array.CreateInstance(GetType(Int32), 5)
```

```
Dim i As Integer
```

```
For i = myArray.GetLowerBound(0) To  
myArray.GetUpperBound(0)
```

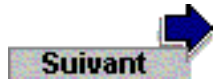
```
    myArray.SetValue(i + 1, i)
```

```
Next i
```

```
□
```

Merci Microsoft pour les exemples.

**On insiste donc sur le fait d'un tableau est de type 'par
référence' (et pas 'par Valeur'), on y reviendra.**



Site **LDF** Cours  : **VB.net**

1.6.5	Les Collections	 Suivant	 Sommaire
--------------	------------------------	--	---

Une alternative aux tableaux est l'usage de **Collection**.

Fait partie de [System.Collections](#)

Une collection fonctionne plutôt comme un **groupe d'éléments** dans laquelle il est possible d'ajouter ou d'enlever un élément à n'importe quel endroit sans avoir à se préoccuper de sa taille ni où se trouve l'élément.

Le nombre d'élément n'est pas défini au départ comme dans un tableau. Dans une collection il n'y a que les éléments que l'on a ajouté.

Les éléments sont repérés grâce à un **index** ou avec une **Clé unique**

Les items affichées dans une ListBox donne une idée concrète de ce qu'est une collection.

Exemple simpliste permettant de comprendre la notion de collection:

Soit la collection `Col`, au départ elle est vide.

J'ajoute des élément (ou items) à cette collection.

```
Col.Add ("Toto")
```

Voici la collection:

Toto

La collection a maintenant 1 élément.

```
Col.Add("Lulu")
```

```
Col.Add("Titi")
```

Toto
Lulu
Titi

La collection a 3 éléments maintenant.

```
Col.Remove(2) enlève l'élément numéro 2
```

Toto
Titi

La collection n'a plus que 2 éléments maintenant.

On voit que le nombre d'élément n'est pas connu à l'avance, il varie en fonction des éléments ajoutés (ou retirés)

Un élément est repéré par son indice.

```
Col.Item(2) contient "Titi" (le second Item de la collection)
```

L'objet Collection:

'Collection' existait déjà en VB6!!

L'objet collection utilise un couple Clé-Valeur, pour chaque élément.

Clé	Valeur
69	Rhone
75	Paris
83	Var
1	Ain

Ici le premier élément à pour clé: 69, pour valeur:'Rhône'

C'est pratique car cela permet de retrouver une valeur à partir de la clé.

Pour utiliser une collection d'objets, vous devez premièrement créer l'objet **maCollection**.

```
Dim maCollection As New Collection
```

Dès que cet objet est créé, vous pouvez ajouter (avec Add), enlever ou manipuler des éléments.

On utilise la syntaxe: `NomCollection.Add(élément, Clé)`

```
maCollection.Add("Bonjour", "30")
```

```
maCollection.Add("Monsieur", "31")
```

```
maCollection.Add("Et", "32")
```

```
maCollection.Add("Madame", "33")
```

Il peut y avoir 2 autres paramètres:

```
maCollection.Add(Element, Clé, Before, After)
```

Before **ou** After peuvent être utilisés pour placer l'élément à insérer avant ou après un élément de la collection. Si Before ou After est un nombre c'est l'index des éléments qui est utilisé, si c'est une string c'est la clé.

Pour récupérer un objet de la collection, on peut utiliser

▫ **l'index:**

```
Label1.Text = maCollection.Item(2) 'Affiche le second élément: Monsieur
```



Attention le premier élément est ici l'élément 1 (l'index va de 1 à Count); c'est hérité du VB6!!

▫ **La clé**

```
Label1.Text = maCollection.Item("33") 'Affiche Madame
```

Liste d'objets: ArrayList .

La ArrayList est une collection particulière: On peut y mettre des **objets** : chaînes, nombre... rien n'empêche que le premier élément soit un entier, le second une chaîne...**Il n'y a pas de clé.**



Attention le premier élément est ici l'élément 0 (l'index va de 0 à count-1) ; c'est du .NET!!

```
Dim L As New ArrayList() 'On crée une collection ArrayList
```

```
Dim L As ArrayList = ArrayList.Repeat("A", 5)
```

'On crée une ArrayList de 5 éléments contenant chacun "A" (on répète "A")

```
L.Add("Bonjour") 'On ajoute un élément à la collections
```

```
MsgBox(L(0)) 'On affiche le premier élément
```

On affiche le premier élément L(0)

On pourra aussi écrire `L.Item(0)` pour pointer le premier élément.

```
MsgBox(L.Count.ToString) 'On affiche le nombre d'élément.
```

Attention c'est le nombre d'éléments. S'il y a 3 éléments dans la ArrayList ce sont les éléments d'index 0,1,2.

```
L.Remove("Bonjour") 'On enlève un élément de la liste
```

```
L.RemoveAt(0) 'On enlève l'élément 0 de la liste
```

```
L.Sort() 'Trie la collection
```

```
L.Clear() 'Efface tous les éléments
```

`L.Contains (élément)` ' Retourne True
si la liste contient élément.

Insert permet d'insérer à un index spécifié.

```
L.Insert( position, Ainserrer)
```

`InsertRange` insère une ArrayList dans une Autre ArrayList.

Pour **parcourir une collection**, 2 méthodes :

'Avec l'index de l'item

```
For i=0 to L.Count-1
```

```
    A=L.Item(i)
```

```
Next i
```

NB: Comme vu plus haut, on utilise Count pour trouver le nombre d'élément, aussi la boucle doit balayer de 0 à count-1.

'Avec For Each

```
Dim o As Objet
```

```
For Each o in L
```

```
    A=o
```

```
Next
```

Autres collections :

StringCollection

Ne peut contenir que des chaînes (cela devrait aller plus vite)

HashTable

comporte des couples **clé-élément**, des paires **clé-valeur**

.

Clé	Valeur
69	Rhone
75	Paris
83	Var
1	Ain

La clé toujours **unique** permet de retrouver la valeur

H.Add(Clé,Valeur) Ajoute un élément

H.Item(Clé) Retourne l'élément correspondant à une clé.

H.ContainsKey(Clé) Retourne True si la Clé est dans la table.



Attention le premier élément est ici l'élément 1 (index allant de 1 à count)

SortedList

C'est une HashTable mais avec ordonnancement des paires par tri à partir de la clé.

Queue

Collection de type FIFO (First In, First Out)

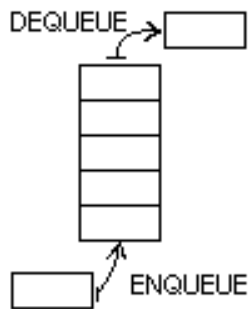
Premier arrivé premier servi.

C'est la queue devant un cinéma, le premier arrivé, prend son billet le premier.

DeQueue supprime et retourne l'objet de début de liste

EnQueue ajoute un objet en fin de liste

Peek retourne l'objet de début sans le supprimer



Stack

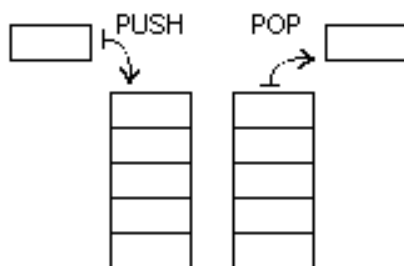
Collection de type **pile** (ou **stack**) LIFO (Last In, First Out)

Dernier entré, premier sortie.

Ce type de stack (pile) est très utilisé en interne par les programmes informatiques: on stocke dans une stack les adresses de retour des procédures appelées, au retour on récupère l'adresse du dessus.

Push insère un objet en haut de la pile

Pop enlève et retourne un objet en haut de la pile



On peut utiliser une pile dans un programme pour gérer le déplacement de l'utilisateur dans un arbre, les éléments en cours sont stockés par Push, pour remonter en chemin inverse, on Pop.



Attention le premier élément est ici l'élément 1 (élément d'index 1 à count)

Exemple:

```
Dim MaPile As New Stack()
```

```
Dim Str As String
```

'Ajouter des éléments à la pile

```
MaPile.Push ("A")
```

```
MaPile.Push ("B")
```

```
MaPile.Push ("C")
```

'Récupérer un objet de la pile:

```
Srt =MaPile.Pop()
```

Str sera égal à "C"

Si la pile est vide et que l'on 'Pop', une exception non gérée du type 'System.InvalidOperationException' se produit.(une erreur se produit et cela plante!!)

BitArray

Crée une collections de booléens. La valeur de chaque élément est True ou False.

Creation de BitArray.

```
Dim myBA As New BitArray(5)
```

```
Dim myBA As New BitArray(5, False)
```

```
Dim myBA() As Byte = {1, 2, 3, 4, 5}
```

```
Dim myBA As New BitArray(myBytes)
```

□ Généralisation de la notion de collection.

Certains objets ont une liste de donnée, Vb les organise en **Collections**.

Une collection peut donc faire partie des propriétés d'un objet.

Exemple:

On verra plus loin d'un contrôle nommé `TextBox` peut contenir du texte, et bien, ce contrôle à une collection nommé `.lines` qui contient chaque ligne du texte (s'il y en a plusieurs)

Si le texte contient 3 lignes, elles seront dans la collection `.lines`

```
Textbox1.lines(0) 'remarquer, ici le premier élément est 0!!
```

```
Textbox1.lines(1)
```

```
Textbox1.lines(2)
```

L'indice des éléments va de 0 à count-1

Autres exemple:

Les contrôles `ListBox` possède une collection `Items` dans laquelle est placé tous les éléments contenus dans la liste. Pour ajouter un élément on utilisera la méthode `Add` de la collection `Items`: `ListBox.Items.Add()`

Un tas d'objets possède des collections.

Encore plus: chaque formulaire possède une **Collections Controls**. Il s'agit d'une collection qui contient tous les contrôles de ce formulaire.

Pourquoi le premier élément est 0 ou 1?



Le `.NET Framework` normalise les collections comme étant des collections de base zéro (`ArrayList` par exemple). La classe `Collections` de `Visual Basic` sert principalement à assurer une compatibilité avec des versions précédentes et fourni des collections de base 1.

Exemples sur les collections:

Créer une `ArrayList`, une queue, ajouter la queue à la `ArrayList`, chercher un élément, insérer un élément.

Les collections font partie de l'espace de nom **`System.Collections`**

```
Imports System.Collections
```

```
' Créer une ArrayList.  
Dim myAL As New ArrayList()
```

```
myAL.Insert(0, "un")
myAL.Insert(1, "deux")
```

' Créer une Queue.

```
Dim myQueue As New Queue()
myQueue.Enqueue("trois")
myQueue.Enqueue("quatre")
```

' Copies la Queue dans ArrayList à l'index 1.

```
myAL.InsertRange(1, myQueue)
```

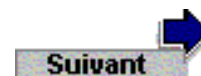
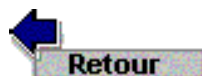
' Chercher "deux" et ajouter "moins de deux" avant

```
myAL.Insert(myAL.IndexOf("deux"), "moins de deux")
```

' Ajouter "!!!" à la fin.



```
myAL.Insert(myAL.Count, "!!!")
```

```
□
```



Site **LDF** Cours **VB.net**



<h2>1.6.6</h2>	<h2>Les Structures</h2>		
		Suivant	Sommaire

Permet de **regrouper des données de type différent**:

(En Vb6 il y avait les types définis par l'utilisateur, ils sont remplacés par les structures.)



Les structures sont intéressantes quand vous voulez utiliser des variables contenant plusieurs **informations de différent type**.

Exemple :

Vous voulez définir une variable contenant une adresse composée d'un numéro, de la rue, de la ville.

Il faut d'abord **définir la structure** (au niveau Module ou Class, **pas** dans une procédure)

```
Public Structure Adresse
    Dim Numero      As Integer
    Dim Rue         As String
    Dim Ville       As String
End Structure
```

Puis dans une procédure il faut **déclarer** la variable :

```
Dim MonAdresse As Adresse
```

La variable **MonAdresse** est déclaré comme une adresse, elle contient donc:

- un numéro 'stocké' dans **MonAdresse.Numero**

une nom de rue 'stocké' dans `MonAdresse.Rue`

un nom de ville 'stocké' dans `MonAdresse.Ville`

On pourra enfin l'utiliser :

```
MonAdresse.Numero=2
```

```
MonAdresse.Rue= "Grande rue"
```

```
MonAdresse.Ville= "Lyon"
```

On peut aussi utiliser le mot clé `With` pour ne pas avoir à répéter le nom de la variable (et cela va plus vite).

`With MonAdresse`

```
    .Rue= "Grande rue"
```

```
    .Ville= "Lyon"
```

```
End With
```

`With` est utilisable sur tous les objets.

Il est possible de travailler sur un tableau de structures:

```
Dim Adresses(99) as Adresse
```

 'Permet de travailler sur un tableau de 100 adresses

```
Adresses(33).Rue="Place de la mairie"
```

On peut utiliser une variable déclarée par une structure comme paramètre d'une fonction:

```
Sub AfficheAdresse( ByVal Une Adresse As Adresse)
```

```
    ...Imprimer l'adresse
```

```
End sub
```

Pour imprimer l'adresse 33 on écrira `AfficheAdresse (Adresse(33))`

Attention quand dans une structure il y a un tableau, il faut l'initialiser:

on veut définir une structure dans laquelle il y a 25 données DriveNumber.

On aurait tendance à écrire:

```
Public Type DriveInfo  
  
    DriveNumber(25) As Integer FAUX  
  
    DriveType As String  
  
End Type
```

' En Visual Basic .NET il faut faire:

1-Méthode par initialize

```
Public Structure  
DriveInfo DriveNumber() As  
Short  
    Dim DriveType As  
String  
    Public Sub Initialize()  
        ReDim  
DriveNumber(25)  
    End Sub  
End Structure
```

On voit qu'une structure peut comporter une méthode.

Ici, on a créé une méthode `Initialize`.

'Exemple de routine utilisant la structure.

```
Function AddDrive(ByRef Number As Short, ByRef  
DriveLabel As String) As Object  
    Dim Drives As  
DriveInfo  
    Drives.Initialize()  
    Drives.DriveNumber(0) =  
123  
    Drives.DriveType =  
"Fixed"  
End Function
```

2-Autre manière de faire:

```
Public Structure  
DriveInfo DriveNumber() As  
Short  
    Dim DriveType As  
String  
End Structure
```

```

Function AddDrive(ByRef Number As Short, ByRef
DriveLabel As String) As Object
    Dim Drives As
DriveInfo
    Drives = DriveInfo.FromName(25)(3)=12
    Drives.DriveType =
    "Fixed"
End Function

```

Si on utilisait 100 variables Drives, il faudrait 'Redim' le tableau pour chaque variable!!

Allons plus loin:

Une structure hérite de **System.ValueType**

Les structures sont des types 'valeur'.

Une variable d'un type structure contient directement les données de la structure, alors qu'une variable d'un type classe contient une référence aux données, ces dernières étant connues sous le nom d'objet.

Cela a de l'importance: si je crée une variable avec une structure, que je copie cette variable dans une seconde, le fait de modifier la première variable ne modifie pas la seconde.

Prenons l'exemple donné par Microsoft:

```

Structure Point
    Public x, y As Integer
    Public Sub New(x As Integer, y As Integer)
        Me.x = x
        Me.y = y
    End Sub
End Structure

```

On définit une structure Point et on définit une méthode **New** permettant de saisir les valeurs:

Public Sub New est un **constructeur**.

Pour saisir les valeurs de x et y on peut utiliser:

```
Dim a As Point
```


a.x=10

a.y=10

ou utiliser le constructeur:

```
Dim a As New Point(10,10)
```

En partant de la déclaration ci-dessus, le fragment de code suivant affiche la valeur 10 :

```
Dim a = new Point(10, 10)
Dim b = a
a.x = 100
Console.WriteLine(b.x) 'b est donc bien différent de a
```

L'assignation de a à b crée une copie de la valeur, et b n'est donc pas affecté par l'assignation à a.x. Si, en revanche, Point avait été déclaré comme une classe, la sortie aurait été 100 puisque a et b auraient référencé le même objet.

Enfin, les structures n'étant pas des types 'référence', il est impossible que les valeurs d'un type structure soient null (elles sont égales à 0 après la création).

Les structures peuvent contenir plein de choses:

On a vu qu'elle peuvent contenir:

Des **variables** de différent type.

Des **tableaux**.

Des **méthodes** : on a vu l'exemple de Initialize et de New.

Mais aussi

Des **objets**.

D'**autres Structures**.

Des **procédures**.

Des **propriétés**.

Exemple donné dans l'aide (et modifié par moi) :

Débutant: A relire peut-être ultérieurement quand vous saurez utiliser les Classes.

Cet exemple définit une structure **Employee** contenant une procédure **CalculBonus** et une propriété **Eligible**.

```

Public Structure Employee
Public FirstName As String
Public LastName As String
' Friend members, accessible partout dans le programme.
Friend EmployeeNumber As Integer
Friend WorkPhone As Long
' Private members, accessible seulement dans la structure.
Private HomePhone As Long
Private Level As Integer
Public Salary As Double
Public Bonus As Double
' Procedure .
Friend Sub CalculateBonus(ByVal Rate As Single)
    Bonus = Salary * CDBl(Rate)
End Sub
' Property pour retourner l'éligibilité d'un employé.
Friend ReadOnly Property Eligible() As Boolean
    Get
        Return Level >= 25
    End Get
End Property
End Structure

```

Utilisons cette structure:

```

Dim ep As Employee      'Déclaration d'une variable Employee

ep.Salary = 100         'On saisit le salaire

ep.CalculateBonus(20)  'On calcul le bonus

TextBox1.Text = ep.Bonus.ToString      'On affiche le bonus

```

Cela ressemble aux Classes !! Non?

Portée:

Vous pouvez spécifier l'accessibilité de la structure à l'aide des mots clé: **Public**, **Protected**, **Friend** ou **Private** ou garder la valeur par défaut, **Public**. Vous pouvez déclarer chaque membre en spécifiant une accessibilité. Si vous utilisez l'instruction **Dim** sans mot clé, l'accessibilité prend la valeur par défaut, **Public**.

```

Private Mastructure

Public i As Integer

...

End Structure

```

En conclusion les structures sont maintenant très puissantes et peuvent contenir autant de chose que les modules de Classes , on verra cela plus loin. Mais les structures sont référencées par valeur alors que les Classes le sont par référence



Retour



Index





Sommaire



Suivant



	<p>Les variables 1.6.7 par valeur ou par référence.</p>	<p>Suivant </p>	<p> Sommaire</p>
--	--	--	---

Résumons la notion très importante de variable par valeur ou par référence.

La variable 'par Valeur':

Contient réellement une valeur.

Prenons pour exemple une variable de type 'Long'

```
Dim L As Long
```

```
L= 1456
```

L occupe 8 octets nécessaire pour coder un long, ici L a une valeur de 1456, donc dans ces 8 octets est codé 1456.

Sont des variables par 'Valeur':

Les Integer, les Long les Short

Les Single, Double, Decimal

Les Booleans, Char, Date

Les Structures

Les énumérations

La variable 'par Référence':

Elle ne contient pas la valeur de l'objet mais son adresse en mémoire, sa référence.

`Dim O As Object`

O contient l'adresse de l'objet codée sur 4 octets.

Sont des variables par référence:

Les Objets

Les Strings

Les tableaux

Les Classes

Influence sur l'Affectation':

Posons le problème:

Travaillons sur A et B, 2 variables ayant la même 'nature'.

A existant déjà, faisons:

`Dim B=A`

Puis modifions la valeur de A, **cela modifie t-il B?**

Les variables par Valeur ou par Référence ne réagissent pas de la même manière:

Si le type de variable est **par valeur** (valable pour les entiers, les Long.. les structures..), chaque variable ayant sa valeur, B n'est pas modifié.

Si le type de variable est **par référence** (valable pour les

tableaux, les objets, les string..), chaque variable est définie par sa référence (son lieu physique); faire A=B entraîne que A et B ont même référence. Si on modifie A, B est modifié car il pointe au même endroit.

Voyons des exemples:

Même si on affecte une variable **par valeur** à une autre, les deux variables restent différentes: elles conservent leur propre espace de stockage:

```
Dim L As Long
```

```
Dim P As Long
```

```
L=0
```

```
L=P      'on affecte P à L
```

```
P=4      'on modifie P
```

```
=> L=0 P=4      Modifier P n'a pas modifié L
```

Par contre si on affecte **une variable par référence** à une autre, elle pointe toutes les 2 sur le même endroit mémoire: si j'en modifie une, cela modifie l'autre.

'Créons une Classe contenant un entier (Exemple à relire quand vous aurez étudié les Classes)

```
Class Class1
```

```
    Public Value As Integer = 0
```

```
End Class
```

```
Dim C1 As New Class1()
```

```
Dim C2 As Class1 =C1      'on crée C2, on affecte C1 à C2
```

```
C2.Value = 123           'on modifie C2
```

```
=> C1.Value=123  C2.Value=123      Modifier C2 a modifié C1 car elles pointent sur le même endroit mémoire.
```

On se méfiera donc du type 'référence ' ou 'Valeur' des

données que l'on utilise: exemple des Tableaux

Exemple sur les tableaux qui sont 'Par référence':

```
Dim A(3) As String
A(1) = "a"
Dim B(3) As String
B(1) = "b"
B = A
A(1) = "c"
Labell.Text() = B(1)      'Affiche 'c'
```

En effet un tableau est 'par référence' et le fait de faire `A=B` donne la même adresse mémoire aux 2 tableaux, aussi , modifier l'un modifie l'autre.

`B= A.Clone` aurait copié le tableau A dans B en conservant 2 tableaux distinct et la dernière instruction aurait affiché 'a'.

Remarque: `A(2)=B(2)` affecte un élément d'un tableau à un élément d'un autre tableau, cela ne modifie que **la valeur** d'un élément et n'affecte pas le tableau.

Le cas particulier des 'String' qui sont 'Par référence':

Attention: par contre :

```
Dim A As String
A = "a"
Dim B As String
B = "b"
B = A
A = "c"
Labell.Text() = B      'Affiche 'a'
```

Bien que cela soit par référence, `B=A` affecte simplement la valeur de A à B, si on modifie ultérieurement A, B n'est pas modifié. (idem pour clone et copy!!) Pour une string il parait donc impossible de la dupliquer, elle se comporte comme une variable par valeur!! **Avez -vous des idées pour expliquer cela?**

Valeur après déclaration:

Après création (avant initialisation) une variable par Valeur contient 0,

```
Dim L As Long 'L contient 0
```

Par contre une String (par référence) qui a été créée par Dim et non initialisée contient Nothing.

```
Dim O As Object 'O contient Nothing: il ne pointe sur aucun objet.
```

On peut le tester par `If IsNothing(O) then..` ou `If O Is Nothing..`

Pour les tableaux, bien que le tableau soit 'par Référence', c'est le type de variable utilisé dans le tableau qui décide de la valeur des éléments après déclaration.

```
Dim T(3) As Long '=>T(0)=0
```

Comparaison:

Une variable par Valeur peut être comparée à une autre par "=",

```
Dim L As Long=12
```

```
Dim P As Long=24
```

```
If L=P Then..
```

Par contre une variable par référence peut être comparée à une autre par "Is".

```
Dim O As Object
```

```
Dim Q As Object
```

```
If O Is Q then..
```

NB: pour les String '=' et 'Is' peuvent être utilisés.

Il existe une instruction permettant de voir si une variable est de type 'Par référence'

Cet exemple utilise la fonction **IsReference** pour vérifier si plusieurs variables font référence à des types référence.

```
Dim R as Boolean
```

```
Dim MyArray(3) As Boolean
```

```
Dim MyString As String
```

```
Dim MyObject As Object
```

```
Dim MyNumber As Integer
```

```
R = IsReference(MyArray) ' R= True. Tableau
```

```
R = IsReference(MyString) ' R= True. String
```

```
R = IsReference(MyObject) ' R= True. Objet
```

```
R = IsReference(MyNumber) ' R= False. Entier
```



Retour





Index



Sommaire



Suivant

1.6.8	Les variables 'Objet' et les autres.	Suivant 	 Sommaire
-------	--------------------------------------	---	--

Il existe un autre type de variable: le type 'Objet'.

La variable 'Objet':

Parfois on ne sait pas ce que va contenir une variable: un Integer? une String? un Single?
Pour résoudre ce problème on utilise une variable de type 'Objet'

`Dim V As Object`

Ensuite:

`V=12`

est accepté, et V sera considéré comme un type Integer

`V=12.6`

est accepté, et V sera considéré comme un type Single

`V="Visual Basic"`

est accepté aussi, et V sera considéré comme un type String

Les 3 affectations `V=` peuvent se suivre sans planter, l'objet contenant successivement un type Integer, Single et String.

On rappelle qu'une variable objet est une variable 'Par référence'.

Comment savoir quel type de variable contient la variable 'Objet'?

Si on fait `V.GetType.ToString` cela retourne 'System.string' indiquant que V contient bien une String.

`V.GetType.Name` retourne 'String'

Pour tester si V est une String, il y a une autre manière:

```
If TypeOf V Is String Then
```

```
End if
```

Comment utiliser les membres du type d'une variable objet?

Exemple: mettre une string dans une variable Objet, connaître la longueur de la String.

- « *Si Option strict=Off* (On force VB à ne pas être trop Strict!!! On verra cela plus loin)

```
Dim V As Object
```

```
V="VB"
```

```
MessageBox.Show(V.length) affiche 2
```

- « *Si Option strict=On* (On force VB à ne rien tolérer)

MessageBox.Show(V.length) déclenche une **erreur**:
les liaisons tardives ne sont pas acceptées.

MessageBox.Show(V.GetType().GetField(n).GetValue(V).ToString)
déclenche une **exception**.

Il faut écrire simplement:(Merci le forum de
developpez.com)

```
Dim V As Object
```

```
V="VB"
```

```
MessageBox.Show(DirectCast(V,  
String).Length.ToString)
```

ou

```
MessageBox.Show(CType(V, String).Length.ToString)
```

DirectCase et CType transforme un type de
variable en un autre, DirectCase est moins
'tolérant' car la variable qui reçoit doit être
du bon type.

Une autre méthode consiste à transformer par
Ctype le contenu de l'objet vers une variable
String, puis à afficher la longueur de cette
variable String.

```
Dim V As Object
```

```
V="VB"
```

```
Dim S As String

S = CType(V, String)

MessageBox.Show(S.Length.ToString)
```



Du fait que les membres utilisés avec une variable Objet ne sont pas définis à l'écriture du programme (on ne sait même pas quel type de variable sera dans l'objet, on n'en connaît donc pas les membres), la recherche du membre se fait à l'exécution, c'est plus long, de plus les contrôles et vérifications se font à l'exécution.

Cela se nomme **une liaison tardive**, à éviter donc. On évitera donc d'utiliser si possible des variables 'Objet'.

Utilisez plutôt des variables typées (des variables String , Integer..) au départ, quand on les utilise, les contrôles et appels sont vérifiés dès le départ, on appelle cela une **liaison anticipée ou précoce**.

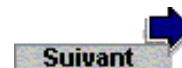
Les variables d'autres types:

On verra que l'interface utilisateur est composée de contrôles, ces contrôles étant des objets, et bien, on peut déclarer une variable de type 'contrôles': bouton, ou textbox ou formulaire.

```
Dim B As Button      'crée une variable B de type Button
```

Ensuite on peut utiliser les membres de la classe Button
`B.BackColor`

```
Dim T As TextBox     'crée une variable T de type TextBox
```



Site **LDF** Cours **VB.net**



	Soyons 1.7 Strict et Explicite.	Suivant →	← Sommaire
--	--	------------------	-------------------

Option Strict

VB est naturellement très arrangeant (trop sympa !!) car il est configuré avec Option Strict On :

Par défaut il transforme, quand c'est possible, et si nécessaire un type de variable en un autre type.

Si je passe un nombre qui est en double précision (Double) dans une variable en simple précision (Single), VB accepte, au risque de perdre de la précision (s'il y a un très grand nombre de chiffre significatif).

Ainsi :

`Dim D As Double`

`Dim S As Single`

`D=0.123456789`

S=D

`MessageBox.Show(s)` ' affiche 0,1234568 le 9
est perdu car un single à 7 chiffres
significatifs.

Cela peut être ennuyeux si c'est des calculs
d'astronomie !! et le programmeur ne s'en rend
pas forcément compte !!

Pour éviter cela il faut activer l'**OPTION
STRICT** à **ON**

Menu Projet > Propriétés de Nom de projet.

Page de propriétés de Langage VB.

Propriétés communes, génération.

En face de **Option Strict**, mettre **On**

Maintenant **seules les conversions effectuées
explicitement seront autorisées.**

`S=D` est souligné dans le code pour signaler une conversion
interdite.

(Par contre `D=S` est accepté car on passe d'une variable à
une variable plus précise)

Il faudra maintenant, pour notre exemple,
écrire :

`S= CType(D,Single)`

Cela entraîne une conversion de la valeur
Double en Single ; **s'il y a perte de**

précision, elle se produit quand même, **MAIS** le programmeur **SAIT** qu'il y a conversion, il prendra ou pas **EN CONNAISSANCE DE CAUSE** le risque.

Avec Option Strict le langage VB.Net devient bien moins tolérant:

Ecrire un programme avec Option Strict à Off, ça passe; mettre Option Strict à On un tas d'instruction coince!! même certains exemples Microsoft!! Car sans s'en rendre compte on passe d'un type de variable à l'autre sans arrêt!!

Autre exemple sur les conversions implicites:

Avec Option Strict=Off

```
Dim n As Integer=12
```

```
MessageBox(n)
```

Affiche 12 : le contenu de l'entier n a été transformé automatiquement en String pour être affiché.

Avec Option Strict=On

```
Dim n As Integer=12
```

```
MessageBox(n)
```

```
Plante
```

Il faut transformer explicitement n en String et écrire:

```
MessageBox(n.ToString)
```

Liaisons tardives:

Avec Option Strict=On VB **refuse les liaisons tardives:**

```
Dim V As Object
```

```
V="VB"
```

```
MessageBox.Show(V.Length) est refusé
```

Il faut écrire

```
MessageBox.Show(CType(V, String).Length.ToString)
```

'Option Strict Off' permet n'importe quoi. C'est du Basic au mauvais sens du terme.

'Option Strict On' oblige à une grande rigueur. C'est du VB.Net

Option explicit

Pour la déclaration des variables nous avons dit que toute variable utilisée devait être déclarée.

Par défaut c'est vrai.

Ouvrir Menu Projet > Propriétés de Nom de projet.

Page de propriétés de Langage VB.

Propriétés communes, génération.

En face de **Option Explicit**, il y a **On**

On pourrait (c'est fortement déconseillé) mettre cette option à Off.

Cela ne rend plus obligatoire la déclaration des variables.

`MaVariable=10` sans déclaration préalable est acceptée.

Cela présente certains inconvénients : Si on fait une faute de frappe en tapant le nom d'une variable, VB accepte le nouveau nom et crée une nouvelle variable objet distinct.

`Dim MaVariable` MaVariable avec un b

`MaVariabble=10` Faute de frappe(bb) Je crois avoir mis 10 dans Mavariabale

En fait j'ai mis 10 dans une nouvelle variable nommée MaVariabble

Mavariabale à toujours une valeur=0

Donc, c'est clair et sans appel : **Laisser Option Explicit à On, ce qui oblige à déclarer toutes les variables avant de les utiliser** Dans ce cas si vous tapez le nom d'une variable non déclarée, elle est soulignée en bleue.

Option strict et Explicit dans un module:

On peut aussi indiquer **dans un module** les options; ces instructions doivent être tapées avant toutes les autres.

```
(Général) (D)
Option Strict On
Option Explicit On

Imports System.ComponentModel
Imports System.Drawing
Imports System.Windows.Forms
Imports Microsoft.VisualBasic
Imports System.Globalization

Public Class Calculator
    Inherits System.Windows.Forms.Form

    'Variables de données.
    Private mOp1, mOp2 As Double
    Private mNumOps As Integer
```



Site LDF Cours VB.net



1.8	<h2 style="color: purple;">Les constantes</h2>		
-----	--	--	--

Comme les variables, elles ont un nom et un type, mais leurs valeurs sont 'constantes'.

On les déclare par le mot **Const**, on peut les initialiser en même temps avec =

Exemple :

```
Const NomFichier= "medical.dic" 'constante
chaîne de caractères.
```

```
Const i As Integer =1 'constante
Integer
```

Intérêt des constantes ?

Améliore la **lisibilité**.

Si une constante doit être modifiée ultérieurement, il suffit en mode conception, de modifier sa valeur ce qui modifie sa valeur dans l'ensemble du code de l'application.

Améliore la vitesse.

On rappelle que **seuls les types primitifs peuvent avoir des constantes** (Byte, Boolean, Short, Integer, Long, Single, Double, Decimal, Date, Char, String)

Constantes prédéfinies:

Il existe une liste de **constantes prédéfinies** dans différentes Classes :

ControlChars.CrLf Chr\$(13)+Chr\$(10) qui sert à sauter à la ligne dans une chaîne de caractères :

Si on affiche **"VISUAL" & ControlChars.CrLf & "BASIC"**

On obtient à l'écran :

VISUAL

BASIC

ControlChars.Tab Chr\$(9) = caractère de tabulation

ControlChars.NullChar Aucun caractère

ControlChars.Nothing Chaîne vide

ControlChars.Back

Taper **ControlChars.** Et comme d'habitude vous obtiendrez la liste des constantes.

Couleurs:

On peut aussi utiliser **les couleurs définies par VB**

`System.Drawing.Color.Blue` 'Pour le bleu

Math:

Si `Import System.Math` a été tapé,

`PI` contient 3,14...

`E` contient la base log naturel

Touche du clavier:

Il est parfois nécessaire de savoir si une touche précise à été tapée par l'utilisateur au clavier, pour cela il faut connaître les touches, mais pas besoin de se souvenir du codes des touches, il suffit de taper **Keys**. et la liste des touches s'affiche. Cliquer sur le nom de la touche recherchée et vous obtenez la constante correspondant à la touche:

`Keys.Right` 'Désigne le code de la touche '->'

True False:

On rappelle que `True` et `False` sont des valeurs Booléens faisant partie intégrante de VB.

Pour les anciens de VB6 ne plus utiliser -1 et 0 (D'ailleurs c'est maintenant 1 et 0).



Utiliser largement ces constantes fournies par VB, cela améliore la lisibilité et la maintenance.

Enum

Le bloc **Enum** permet de créer une liste (une énumération) de constantes créées par le programmeur :

```
Enum TypeFichier
```

```
    DOC
```

```
    RTF
```

```
    TEXTE
```

```
End Enum
```

Les constantes ainsi créées sont `TypeFichier.DOC` ,
`TypeFichier.RTF`, `TypeFichier.TEXTE`

Le bloc Enum doit être dans l'en-tête du module.

C'est bien pratique car en écrivant le code, dès que je tape **TypeFichier.** la liste (DOC RTF TEXTE) apparaît.

Ensuite, on peut utiliser dans le programme les constantes créées.

`TypeFichier.DOC` par exemple.

Chaque constante de l'énumération a une valeur par défaut.

Par défaut

```
TypeFichier.Doc =0
```

```
TypeFichier.RTF =1
```

```
TypeFichier.TEXTE=2
```

..

Parfois le nom utilisé dans l'énumération est suffisant en soi et on n'utilise pas de valeur : Dans un programme gérant des fichiers, un flag prendra la valeur `TypeFichier.DOC` pour indiquer qu'on travaille sur les fichiers .DOC. Peu importe la valeur de la constante, mais d'autres fois il faut que chaque constante de l'énumération possède une valeur particulière.

Je peux imposer une valeur à chaque constante de l'énumération :

```
Enum TypeFichier
```

```
    DOC=15
```

RTF=30

TEXTE=40

End Enum

Je peux même donner plusieurs valeurs avec And et Or à condition d'utiliser l'attribut Flags.

```
<Flags(> Enum TypeFichier
```

```
    DOC=15
```

```
    RTF=30
```

```
    TEXTE=40
```

```
    TOUS= DOC AND RTF AND TEXTE
```

End Enum

Les énumérations sont des types qui héritent de **System.Enum** et qui représentent symboliquement un ensemble de valeurs. Par défaut ses valeurs sont des 'Integer' mais on peut spécifier d'autres types: Byte, Short, Integer ou Long.

L'exemple suivant déclare une énumération dont le type sous-jacent est Long :

```
Enum Color As Long
```

```
    Red
```

```
    Green
```

```
    Blue
```

```
End Enum
```

Noter que Vb contient un tas de constantes classées avec la manière Enum.

Exemple:

Quand on ferme une MessageBox. (une fenêtre qui affiche un message), cela retourne une valeur qui contient:

```
'MsgBoxResult.Yes',
```

```
'MsgBoxResult.No'
```

'MsgBoxResult.Cancel'

en fonction du bouton qu'a utilisé l'utilisateur pour sortir de la fenêtre MessageBox (appuié sur les boutons Oui, Non Cancel) .



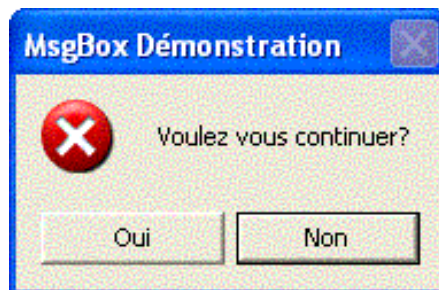
1.9	La surcharge.	Suivant 	 Sommaire
-----	---------------	---	--

Surcharge

Quand on utilise une méthode avec des paramètres, il y a parfois possibilité d'utiliser, avec la même méthode, un nombre différent de paramètres ou des paramètres de nature différentes: on appelle cela **surcharger la méthode**.

Chaque manière d'écrire les paramètres s'appelle une **signature**.

Exemple:



Voici une fenêtre MessageBox:

pour ouvrir une fenêtre MessageBox il y a 12 signatures, en voici 2:

```
Reponse= MessageBox.show(TexteAAfficher, Titre, TypeBouton et Icone, BoutonParDéfaut)
```

Ici on donne 4 paramètres.

```
Reponse= MessageBox.show(TexteAAfficher)
```

Ici 1 seul paramètre.

On voit qu'on peut appeler la méthode `MessageBox.Show` avec un nombre différent de paramètres.

Comme on ne peut pas connaître toutes les signatures, VB nous aide:

Si on tape `R= MessageBox.show` VB affiche dans un cadre une signature, de petites flèches permettent de faire défiler tous les autres signatures:

R=[MessageBox.Show](#) |

D1 ▲ 6 sur 12 ▼ Show (**text As String**) As System.Windows.Forms.DialogResult

D1 **text:** Texte à afficher dans le message.





1.1 0	Les Opérateur	Suivant	Sommaire
----------	------------------	---------	----------

S

Pour travailler sur les variables on utilise des opérateurs (addition, soustraction...).

Addition : +

Dans le cas de variables numériques.

$A=B+C$

si $B=2$ et $C=3 \Rightarrow A=5$

On peut écrire:

$A=A+1$

Dans ce cas, on affecte à la variable A son ancienne valeur +1, si $A=2$ au départ il aura ensuite pour valeur 3.

$A+=1$ est équivalent à $A=A+1$

Cela incrémente la variable A.

Soustraction : -

$$B=C-D$$

$A-=1$ est équivalent à $A=A-1$

Multiplication :*

$$B= C*D$$

Division : /

On remarque que ce n'est pas « : » qui est l'opérateur de division. (Ce signe sert de séparateur quand plusieurs instruction sont sur la même ligne.)

$$B=C/D$$

Division entière:\

$$\text{Si } A=10\backslash 3 \Rightarrow A=3$$

Puissance : ^

`A=B^3` `'A=B*B*B`

Modulo :

C'est le reste de la division par un nombre :

`10 Mod 3` donne 1

Exemple A est-il multiple de 3 ?

Si `A Mod 3 = 0` , A est un multiple de 3

`If A Mod 3 = 0 then`

Concaténation : &

C'est une mise bout à bout des chaînes de caractères.

Si

`A= "VISUAL"`

`B= " "`

C= "BASIC"

D=A & B & C donne D="VISUAL BASIC"

Le signe + peut aussi être utilisé mais il est plutôt réservé aux additions de variables numériques.

&= permet aussi la concaténation A&B est équivalent à A= A&B

Priorités :



L'ordre des calculs se fait en fonction de la priorité des opérateurs.

S'il y a plusieurs opérateurs ^ a la priorité la plus forte puis * et / puis + et -

Pour être complet, voyons les priorités par ordre décroissant:

^ élévation à la puissance

- négation unaire

/ et * multiplier et diviser

\ division entière

mod modulo

+ et- addition et soustraction.

Exemple $2+3^3$ donne 29 car VB effectue

(3^3)+2 et non pas 125 (2+3)^3

S'il y a plusieurs opérateurs de même priorité, l'ordre des calculs se fait de gauche à droite.



Pour éviter toute faute d'interprétation utiliser des parenthèses :

2+(3^3) lève toute ambiguïté.

Comparaison :

- = égal
- > supérieur à
- < inférieur à
- >= supérieur ou égal
- <= inférieur ou égal
- <> Différent de

Le résultat d'une comparaison est **True** (Vrai) ou **False** (Faux)

Exemple :

```
Dim A As Integer=2
```

```
Dim B As Integer=3
```

`If A=B then..`

A étant différent de B, A=B prend la valeur False et le programme passe à la ligne en dessous (pas après then).

Ici le signe = n'indique pas une affectation mais une expression à évaluer.

Ici aussi on peut combiner les opérateurs et mettre des parenthèses:

```
R= (C<>D)AND (D=2)
```

Comparaison de chaîne de caractères:

Les chaînes de caractères sont comparées en fonction du tri alphabétique.

A<B<C.....<Y<Z<a<b<c.....y<z<à<é..

```
Dim A As String="A"
```

```
Dim B As String="Z"
```

`If A<B then..`

A est bien inférieur à B, donc A<B prend la valeur True et le programme saute après Then.

La casse (majuscules ou minuscule) est différenciée.

Si on veut comparer sans tenir compte du fait que c'est en majuscule ou minuscule, il faut d'abord transformer les 2 chaînes en minuscule par exemple.

On veut comparer A= "aaa" et B= "AAA"

Normalement A est différent de B :

```
A=B retourne False
```


Par contre `A.ToLower=B.ToLower` retourne True (Vraie)

En utilisant `Option Compare Text` en début de module, on ne différencie plus la casse: "A" devient égal à "a".

Logique : Not And Or ElseOr Xor

SI A et B sont des expressions Booléens:

`A And B` retourne True si A et B sont vrais

`A Or B` retourne True si une des 2 est vrai

`A Xor B` retourne True si une et une seule est vrai

`Not A` retourne True si A était faux et vice versa

On entend par expression Booléen le résultat de l'évaluation d'une condition:

`A=B` retourne True si A=B et False si A différent de B.

Exemple

Si A différent de B.. peut s'écrire `IF Not(A=B)..`

Si A compris entre 2 et 5 peut s'écrire `IF A>=2 And A<=5..`

Rentrons dans le détails:

Les opérateurs **And**, **Or** et **Xor** sont évalués comme suit en fonction du type d'opérandes :

- q Pour le type **Boolean** :
 - q Une opération **And** logique est effectuée sur les deux opérandes.
 - q Une opération **Or** logique est effectuée sur les deux opérandes.
 - q Une opération **Or** exclusif logique est effectuée sur les deux opérandes.
- q Pour les types **Byte, Short, Integer, Long** et tous les types énumérés, l'opération spécifiée est réalisée sur chaque bit de la représentation binaire des deux opérandes :
 - q **And** : Le bit de résultat est 1 si les deux bits sont 1. Sinon, le résultat est 0.
 - q **Or** : Le bit de résultat est 1 si l'un des deux bits est 1. Sinon, le résultat est 0.
 - q **Xor** : Le bit de résultat est 1 si l'un des deux bits est 1 mais pas les deux. Sinon, le bit de résultat est 0 (c'est-à-dire $1 \text{ Xor } 0 = 1$, $1 \text{ Xor } 1 = 0$).


Les opérateurs **AndAlso** et **OrElse** sont uniquement définis sur le type **Booléen**, ils sont **plus rapide** car ils n'évaluent pas la seconde expression si ce n'est pas nécessaire.

Déplacement de bits: << et >>

Les opérateurs binaires << et >> effectuent des opérations de déplacement de bits. Ces opérateurs sont définis pour les types **Byte, Short, Integer** et **Long**.

L'opérateur << **décale à gauche les bits** du premier opérande du nombre de positions spécifié. Les bits de poids fort situés en dehors de la plage du type de résultat sont éliminés, et les positions libérées par les bits de poids faible sont remplies par des zéros.

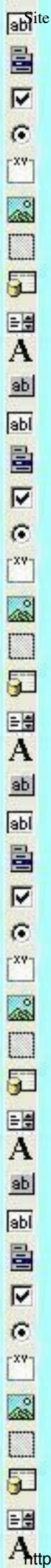
L'opérateur >> **décale à droite les bits** du premier opérande du nombre de positions spécifié. Les bits de poids faible sont éliminés et, si l'opérande de gauche est positif, les positions libérées par les bits de poids fort sont mises à zéro ; s'il est négatif, elles sont mises à un. Si l'opérande de gauche est de type **Byte**, les bits de poids fort disponibles sont remplis par des zéros.

 Retour

 Index



 Sommaire

Suivant 



Site **LDF** Cours **VB.net**



1.11	Les Structures de contrôle	Suivant 	 Sommaire
-------------	-----------------------------------	--	---

Elles permettent de gérer le déroulement du code.

On étudiera :

If Then

Select Case

For Next

Do Loop

While End While

For Each

Switch

IIF

If then

Permet de créer une **structure décisionnelle** :

```
If Condition Then
```

```
End if
```

Si Condition vraie alors.....

Une instruction ou un bloc d'instructions peut être exécuté si une condition est vraie.

Exemple:

```
If A=B Then MsgBox("A=B")
```

Si A = B alors on affiche dans une fenêtre MessageBox « A=B »

'Noter que tout est sur une seule ligne (Pas besoin de End If), on aurait pu écrire:

```
If A=B then
```

```
    MsgBox("A=B")
```

```
End If
```

On peut **tester une condition fausse** et dans ce cas utiliser **Not**.

```
If Not A=B Then MsgBox("A différent de B")
```

Si A et B sont différent (Not A=B signifie NON égaux) afficher "A différent de B".

Il peut y avoir **des opérateurs logiques** dans la condition:

```
If A=B And C=D then. □
```

Autre exemple :

```
If Not IsNumeric(N) then
    MsgBox ("R n'est pas un nombre")
    Exit Sub
End if
```

Si N n'est pas numérique alors afficher dans une boîte de dialogue: « R n'est pas un nombre » puis quitter la procédure (Exit Sub)

Noter bien que comme il y a plusieurs instructions après Then on crée un bloc d'instruction de plusieurs lignes entre Then et End If.

Au lieu de

```
If Condition=True Then
End if
```

On peut écrire:

```
If Condition Then
End if
```

Condition étant de toute manière évaluée pour voir si elle est vraie.

On peut aussi utiliser la structure :

```
Si..Alors..Sinon
```

```
If condition then
.. 'effectué si condition vraie
..
Else
    ..'effectué si condition fausse
    ..
End if
```

Des structures If Then peuvent être **imbriquées** :

```
If..
    If.
        ..
    Else
        If.
            ..
        End if
    End if
End If
```

Pour bien repérer les différents niveaux, utiliser les tabulations et décaler le 'If then' et son code au même niveau.

Pour vérifier s'il n'y a pas d'erreur, compter les 'if', il doit y en avoir autant que des 'end If'. VB souligne le 'if' si il n'y a pas de 'end if'.

Dernière syntaxe:

vbintro

```
If Condition1 Then
    ..
ElseIf condition2 Then
    ..
ElseIf condition3 Then
    ..
end if
```

```
Si condition1...
```

```
Sinon si condition2
```

```
Sinon si condition3
```

```
Fin Si
```

Select Case

créer une structure décisionnelle permettant d'exécuter **un grand nombre de blocs de code différents en fonction de la valeur d'une expression :**

```
Select Case expression
```

```
Case valeur1
```

```
    'code effectué si expression=valeur1
```

```
    .....
```

```
Case valeur2
```

```
    'code effectué si expression=valeur2
```

```
    .....
```

```
Case valeur3
```


'code effectué si expression=valeur3

.....

..

Case Else

'code effectué dans tous les autres cas

End Select

Attention si expression=valeur1 le code entre Case Valeur1 et Case valeur2 (et uniquement celui là) est effectué, puis l'exécution saute après End Select.

Exemple d'un code affichant le jour de la semaine :

N est un entier contenant le numéro d'ordre du jour (entre 1 et 7)

Select Case N

Case 1

MsgBox "Lundi"

□

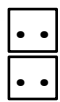
Case 2

MsgBox "Mardi"

□

Case 3

MsgBox "Mercredi"



Case Else

```
MsgBox "Nombre pas entre 1 et 7"
```

End select

Nous venons d'utiliser une expression simple après chaque Case mais on peut utiliser des expressions plus complexes :

Plusieurs clauses d'expression peuvent être séparées par des virgules.

Select Case N

Case 8,9,10

```
'Effectuer le code si N=8 ou N=9 ou  
N=10
```

Le mot clé **To** permet de définir les limites d'une plage de valeurs correspondantes pour N.

Select Case N

Case 8 To 20

```
'Effectuer le code si N est dans la  
plage 8 à 20
```

Le mot clé **Is** associé à un opérateur de comparaison (=, <>, <, <=, > ou >=) permet de spécifier une restriction sur les valeurs

correspondantes de l'expression. Si le mot clé Is n'est pas indiqué, il est automatiquement inséré.

Select Case N

Case Is >= 5

Effectuer le code si N supérieur ou égal à 5.

Vous pouvez utiliser plusieurs expressions ou plages dans chaque clause Case (séparées par des virgules). Par exemple, la ligne suivante est valide :

Case 1 To 4, 7 To 9, 11, 13, Is > MaxNumber

Vous pouvez aussi indiquer des plages et des expressions multiples pour des chaînes de caractères. Dans l'exemple suivant, Case correspond aux chaînes qui sont absolument identiques à « aaa », aux chaînes comprises entre «ccc» et «ddd» dans l'ordre alphabétique, ainsi qu'à la valeur de Var :

Case "aaa", "ccc" To "ddd", Var

For Next

permet de faire des **boucles**.

Les boucles sont très utilisées pour parcourir une plage de valeur qui permet par exemple de

parcourir tous les éléments d'un tableau ou pour effectuer de manière itérative un calcul.

Le nombre de boucle va être déterminé par une variable qui sert de **compteur**.

Le nombre d'exécution est déterminé au départ de la boucle car le compteur a une valeur de départ, une valeur d'arrêt.

Pour variable allant de 'début' à 'fin'

Boucler

donne en VB

```
For variable=début To fin
```

```
..
```

```
Next variable
```

```
Dim i as integer
```

```
For i=1 to 10
```

```
MsgBox i.toString
```

```
Next i
```

En langage courant : Pour i allant de 1 à 10, afficher la valeur de i dans une MessageBox.

La **variable compteur** va prendre successivement les valeurs 1 puis 2 puis 3..... jusqu'à 10 et effectuer à chaque fois le code qui est entre For et Next.

Si on décompose :

i=1 Affiche « 1 », arrivé à Next, remonte à For, i =2 , affiche « 2 ».....

..... i=10 , affiche « 10 » poursuit après Next.

Il peut y avoir un pas (**Step**), le compteur s'incrémente de la valeur du pas à chaque boucle.

```
Dim i as integer
```

```
For i=1 to 10 Step 2
```

```
MsgBox i.toString
```

```
Next i
```

Affiche 1 puis 3 puis 5 puis 7 puis 9

Attention si la valeur de sortie de boucle est inférieure à celle d'entrée il faut indiquer un **pas négatif**

```
Dim i as integer
```

```
For i=10 to 1 Step -2
```

```
MsgBox i.toString
```

```
Next i
```

Affiche 10 puis 8 puis 6 puis 4 puis 2

Bien sur on peut utiliser des expressions calculées :

For i=A to B*10 Step X-2

MsgBox i.toString

Next i

La variable boucle peut être déclarée après For, dans ce cas cette variable n'existe que dans la boucle:

```
For K As Integer = 1 To 10
    ...
Next K
```

Do Loop

Permet aussi de faire des **boucles** mais sans que le nombre de boucle (d'itération) soit déterminé au départ.

C'est la condition d'arrêt qui détermine la sortie de la boucle :

Après **Do** on doit mettre **Until** (Jusqu'à ce que) ou **While** (Tant que)

Do Until condition

Code

Loop

'Boucler jusqu'à ce que : condition est une condition de sortie.

Si condition est fausse, effectuer le code, boucler et recommencer le code jusqu'à ce que

condition soit vraie.

A chaque boucle la condition est évaluée.

Exemple pour chercher un mot dans une liste :

Lire Premier Mot

```
Do Until MotCherché=MotPointé
    Pointer Mot suivant
Loop
```

On peut aussi utiliser **While** (Tant que)

Lire Premier mot

```
Do While MotCherché<>MotPointé
    Pointer Mot suivant
Loop
```

Tant que le mot cherché est différent du mot pointé continuez à boucler.

La condition peut être mise **en fin de boucle**, cela permet d'effectuer au moins une fois le code. Cela évite aussi d'avoir à démarrer le processus avant la boucle : dans notre exemple cela permet d'éviter de lire le premier mot :

Les mots sont dans un tableau Mot(); premier élément Mot(0)

```
IndexMot=-1
```

```
Do
```

```
    IndexMot=IndexMot+1
```

```
Loop While MotCherché<>Mot(IndexMot)
```

Il faudrait en plus boucler jusqu'à la fin du tableau et pas plus.

Exemple complet : Imposer la saisie d'un nombre négatif à l'utilisateur :

On utilise **InPutBox** qui ouvre une fenêtre et attend une réponse.

Dim Reponse as Single

Do

```
    Reponse=InPutBox(« Entrer un nombre négatif. »)
```

```
Loop While Reponse>=0
```

Si le nombre n'est pas négatif, la boucle fonctionne et la boîte InPutBox s'ouvre de nouveau.

While, End While

Permet une boucle qui tourne tant qu'une condition est vraie.

While Condition

...

End While

Exemple: on incrémente un compteur , on sort quand il est égal à 20.

```
Dim Counter As Integer = 0
```

```
While Counter < 20 ' Test la valeur du compteur.
```

```
Counter += 1 ' Incrémente le compteur.
```

```
End While
```


For Each

C'est une variante de la boucle For mais elle permet de **parcourir les objets d'une collection**.

Prenons un contrôle **ListBox** il a une collection **Items** qui contient tous les éléments de la ListBox

ListBox.item(0) contient la première ligne

ListBox.item(1) contient la seconde ligne

ListBox.item(2)...contient la troisième.

Parcourir tous les éléments de la ListBox et les mettre dans une variable V s'écrirait :

```
Dim V as string
Dim item as objet
For Each item in ListBox.items
    V=V+item
Next
```

La variable de boucle peut être déclarée après For.

Switch.

Switch est utilisé avec des **couples d'arguments**, si le premier est vrai, le second est retourné.

```
Réponse=Switch( Expression1, Reponse1, Expression2,
Reponse2)
```

Si Expression2 est vrai Reponse2 est retourné.

```
Monnaie= Microsoft.VisualBasic.Switch(Pays = "USA",
"Dollar", _
Pays = "FRANCE", "Euro", Pays = "Angleterre", "Live")
```

Si `Pays="FRANCE"`, cette expression est vrai, le second objet du couple est retourné.

Retourne `Euro`

IIf

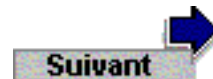
IIf est utilisé avec 3 arguments.

Si le premier argument est vrai , le second est retourné.

Si le premier argument est faux c'est le troisième qui est retourné.

```
Reponse = IIf( Nombre > 0, "Positif", "Négatif ou 0")
```

Comme dans Switch on peut utiliser des procédures comme argument.



Site **LDF** Cours **VB.net**



1.12	<h2>Procédures et paramètres</h2>		
------	---	--	--

Revenons sur les procédures et leurs paramètres.

Quand on appelle une **procédure** (un sous-programme, une routine), le logiciel 'saute' au sous-programme, il effectue celui-ci puis revient effectuer ce qui est sous l'appel.

En VB les procédures sont des **Sub** ou des **Function**.

On peut fournir aux procédures des **paramètres** qui sont envoyés à la fonction.

Exemple:

```
Function Carré ( V as Single) as Single
    Return V*V
End Function
```

Cela crée une fonction qui se nomme 'Carré' , on peut lui envoyer un paramètre (elle accepte un Single)

Cette fonction retourne le carré du paramètre fourni.

Pour l'utiliser :

```
Dim resultat as Single
```

```
resultat= carré(2)          'resultat est alors égal à 4
```

On appelle la fonction carré avec le paramètre 2, elle retourne 4.

Les paramètres peuvent être des variables:

```
Dim resultat as Single
```

```
Dim valeur as Single=3
```

```
resultat= carré(valeur)
```

Les parenthèses.

Rappel, même s'il n'y a pas de paramètre, mettre des () lors de l'appel de procédure.

```
MaRoutine()
```

Par Valeur, Par Référence.

Il y a 2 manières d'envoyer des paramètres :

Par valeur : (By Val) c'est la valeur (le contenu de la variable) qui est envoyée.

Par référence : (By Ref) c'est l'adresse (le lieu physique où se trouve la variable) qui est envoyée. Si la Sub modifie la variable, cette modification sera visible dans la procédure appelante après le retour.

Exemple de procédures:

```
Sub MaProcedure (ByRef x as Long, ByVal y As Long)
```

```
End Sub
```

Si j'appelle cette procédure à partir d'une procédure nommée Main() :

```
Sub Main()
```

```
    MaProcédure (A, B)
```

```
End sub
```

C'est l'adresse de **A** qui est envoyée et la valeur contenue dans la variable **B**. Elles se retrouvent dans les variables **x** et **y** de la procédure **MaProcédure**.

.Si dans cette dernière je modifie **x** , **A** est modifié dans la Sub Main (puisque **x** et **A** pointe sur le même endroit). Si dans **Maprocédure** je modifie **y** , **B** n'est pas modifié.



ATTENTION: Pas défaut les paramètres sont transmis PAR VALEUR

Pour la clarté du code et pour éviter toute ambiguïté, spécifier **ByRef ou **ByVal**, c'est plus lisible , plus clair.**

```
Taper Sub MaProcédure (ByRef x as Long, ByVal x As Long)
```

```
Plutôt que Sub MaProcédure ( x as Long, x As Long)
```

Optional

Un paramètre ou argument peut être **Optional**, c'est à dire facultatif.

Indique que cet argument n'est pas requis lorsque la

procédure est appelée. Si ce mot clé est utilisé, tous les arguments suivants doivent aussi être facultatifs et déclarés à l'aide du mot clé **Optional**. Chaque déclaration d'argument facultative doit indiquer une valeur par défaut.

```
Sub MaRoutine (Optional X As Integer=0)
```

Tableau de paramètres.

Il est possible d'envoyer un tableau comme paramètre.

Exemple:

```
Dim Reponses(10) As Integer
```

```
'Appel de la Sub
```

```
Affiche( Reponses())
```

La Sub Affiche débute par:

```
Sub Affiche ( R() As Integer )
```

```
End Sub
```

ParamArray

Parfois il faut envoyer des paramètres **de même type** mais dont on ne connaît pas le nombre, dans ce cas on utilise **ParamArray** (Liste de paramètres):

```
Function Somme ( ByVal ParamArray Valeurs() as Integer) As Integer
```

```
    Dim i as Integer
```

```
    Dim Total as Integer
```

```
    For i=0 to Valeurs.Length-1
```

```
        Total += Valeurs(i)
```

```
    Next i
```

```
    Return Total
```

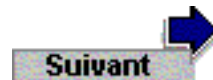
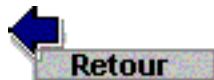
```
End Sub
```

Pour appeler cette fonction:

Dim LeTotal As Integer

LeTotal= Somme (2, 5, 6, 8 ,5)

A noter que le paramètre **ParamArray** doit être le dernier des paramètres, c'est obligatoirement un paramètres ByVal et comme on l'a dit, tous les paramètres sont de même type.



Site LDF Cours VB.net



1.13	Portée des variables	Suivant	Sommaire
------	----------------------------	---------	----------

Quand on déclare une variable, jusqu'ou est-elle visible?

Dans les procédures.

Si on déclare une variable dans une procédure, elle est visible uniquement dans cette procédure, c'est une variable **locale**:

```
Sub MaProcedure (ByRef X As Integer)
```

```
    Dim Y As Integer
```

```
    ...
```

```
End sub
```

`Y` est déclaré en début de procédure, on pourra travailler avec `Y` dans la procédure jusqu'à `End Sub`

Dans une autre procédure `Y` ne sera pas visible (l'utilisation de `Y` déclencherait une erreur.)

Après `End Sub` `Y` n'existe plus, son contenu est perdu.

Il en est de même pour `X` qui est déclaré sur la ligne `Sub`.

Une autre procédure pourra déclarer et utiliser une variable `Y`, mais, même si elle a le même nom cela ne sera

pas la même: chaque variable Y est uniquement visible dans sa procédure.

Variable statique:

Si à la place de Dim on utilise **Static**, la variable est dite 'Statique': A la sortie de la procédure, la valeur continue d'exister et garde sa valeur, lors des appels suivants de la procédure, on retrouve la valeur de la variable.

Exemple

```
Sub compteur
  Dim A as integer
  Static B as integer
  A +=1
  B +=1
End sub
```

A chaque appel de cette procédure A prend la valeur, 0 puis 1 puis disparaît.

B prend les valeurs 0, puis 1, puis 2... (incrémentation à chaque appel)

Dans un bloc d'instruction.

Si vous déclaré une variable dans un bloc, elle ne sera visible que dans ce bloc:

```
Do
  Dim Compteur A integer
  Compteur +=1
...
Loop
```

La variable **Compteur** existe uniquement entre **Do** et **Loop**

Dans la section déclaration d'un Module.

Dans la **section déclaration d'un module**, on utilise à la place de Dim:

Private; dans ce cas la variable est propre au module, elle

est visible dans toutes les procédures du module, pas dans les autres modules.

Public; dans ce cas la variable est accessible dans la totalité du programme.

```
Public A as String
```

A est accessible partout.

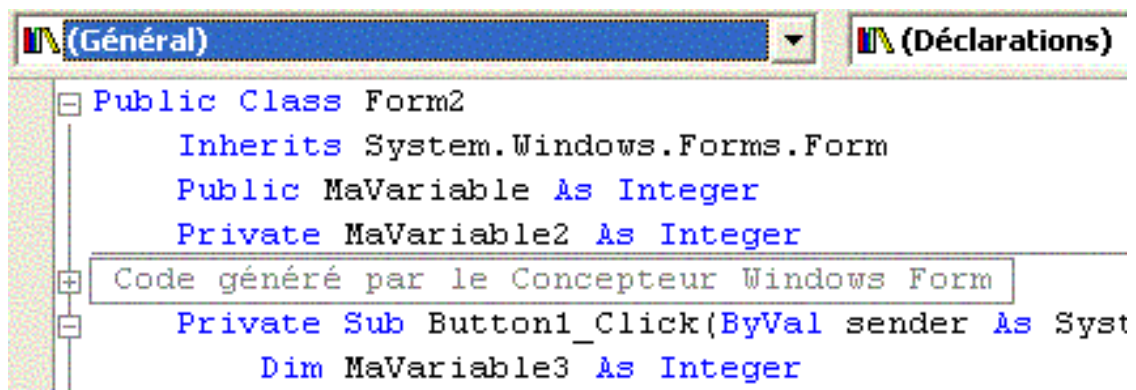
Dans la section déclaration d'une fenêtre, d'un formulaire.

Private; indique dans ce cas que la variable est propre au formulaire, elle est visible dans toutes les procédures du formulaire, pas dans les autres modules ou formulaires.

Public; indique de même une variable UNIQUEMENT visible dans le formulaire.

Elle est visible hors du formulaire à condition de la préfixer avec le nom du formulaire.

Exemple:



```
(Général) (Déclarations)
Public Class Form2
    Inherits System.Windows.Forms.Form
    Public MaVariable As Integer
    Private MaVariable2 As Integer
    Code généré par le Concepteur Windows Form
    Private Sub Button1_Click(ByVal sender As Syst
        Dim MaVariable3 As Integer
```

Dans l'exemple ci dessus:

`MaVariable` est visible dans le formulaire. et hors du formulaire à condition d'utiliser `NomFormulaire.MaVariable`.

`MaVariable2` est visible dans le formulaire .

`MaVariable3` n'est visible que dans la procédure `Button1_Click`.

En pratique:



Pour se repérer et se souvenir **quelle est la portée d'une variable**, on utilise une lettre en début du nom de la variable:

`g_MaVariable` sera public (g comme global).

`m_Variable2` sera accessible au niveau du module.

Dans un **module standard**, on met toutes les **variables Public** accessibles par tous. Leurs noms débute par **g**. Ce sont les variables (et constantes) générales utilisées dans la totalité de l'application: état du programme, utilisateur en cours...

Dans **chaque formulaire** on met dans la section déclarations, les **variables du module**: état du formulaire. Variable permettant l'affichage...

Dans **chaque procédure** les **variables locales**, compteur de boucle...

Pour les variables locales on peut donc utiliser un **même nom** dans différentes procédures, par exemple, on nomme souvent I les variables de boucle dans toutes les procédures, par contre



il faut éviter de donner un même nom à des variables dont la portée se recoupe.

VB l'accepte et utilise la variable la plus proche, celle du bloc ou du module...mais c'est dangereux et générateur de bugs.



Retour



Index



Sommaire



Suivant



1.14	Le Hasard, l'aléatoire	Suivant	Sommaire
------	---------------------------	---------	----------

Comment obtenir un nombre aléatoire?

`Rnd()` fournit un nombre aléatoire entre 0 et 1 (sans jamais attendre 1): valeur inférieure à 1 mais supérieure ou égale à zéro; ce nombre aléatoire est un **Single**.

En fait ,si on fait des `Rnd()` successifs, le nombre aléatoire précédemment généré est utilisé pour le calcul du nombre aléatoire suivant, ce qui fait que la suite de nombre aléatoire est toujours la même.

`Randomize()` initialise le générateur de nombres aléatoires. Si on ne donne pas d'argument, `Randomize` utilise la valeur de l'horloge interne pour initialiser; cette valeur est dû au hasard, aussi le `Rnd` qui suit va être dû au hasard.

Si on n'utilisait pas `Randomize()` avant `Rnd()`, la fonction `Rnd()` fournirait toujours les mêmes nombres aléatoire dans le même ordre.

En résumé:

En fait `Rnd` fournit une suite de nombre **pseudo aléatoire** (le suivant étant calculé à partir du précédent), la suite est toujours la même, seule le premier change et est initialisé par `Randomize` qui est basé soit sur l'horloge système (et qui à priori initialise à une valeur toujours différente) s'il n'y a pas d'argument soit sur un argument.

Pour obtenir plusieurs fois **les mêmes séries de nombres** , utiliser **Randomize** avec un argument numérique puis appelez `Rnd()` avec un argument négatif.

Simuler un jeu de lancé de dé.

Comment obtenir un nombre entier entre un et six au hasard?

```
Dim MyValue As Integer
```

```
Randomize ' Initialise le générateur de nombre aléatoire.
```

```
MyValue = CInt(Int((6 * Rnd()) + 1)) ' Génère un nombre  
aléatoire entre 1 et 6.
```

`Rnd()` fournissant un nombre aléatoire entre 0 et 1, je le multiplie par 6 et j'ajoute 1 pour qu'il soit entre 1 et 7 sans atteindre 7 (il peut être entre 1 et 6,999), je prend sa valeur entière: il est maintenant entre 1 et 6, enfin je le transforme en Integer.



Site **LDF** Cours **VB.net**



1.15	Récursivité.		
		Suivant	Sommaire

Une procédure est récursive si elle peut s'appeler elle même.

Exemple: calcul de 'Factorielle'.

On rappelle que $N!$ (factorielle N) = $1*2*3*...*(N-2)*(N-1)*N$

Exemple Factorielle 3 = $1*2*3$

```
Dim R As Long
```

```
R=Factorielle(3) 'retournera 6
```

Cette fonction n'est pas fournie par VB, créons **une fonction Factorielle SANS récursivité**:

```
Function Factorielle (ByVal N as Long) As Long
    Dim i As Long
    Resultat=1
    For i= 1 to N
        Resultat=i* Resultat
    Next i
    Return Resultat
end Function
```

Cela crée une fonction recevant le paramètre N et retournant un long.

La boucle effectue bien $1*2*3*...*N-1*N$.

Factorielle avec 'Récursivité':

Une autre manière de calculer une factorielle est d'utiliser la **récursivité**:

VB gère la récursivité.

Comment faire?

On sait que $N! = N * (N-1) * (N-2) \dots 3 * 2 * 1$

on remarque donc que `Factorielle N! = N * Factorielle(N-1)`

`N! = N*(N-1)!` : en sachant que `1!=1`

Créons la fonction:

Si N=1 la fonction retourne 1 sinon elle retourne N*factorielle(N-1)

```
Function Factorielle (ByVal N as Long) As Long
    If N=1 then
        Return 1
    Else
        Return N* Factorielle(N-1)
    End If
end Function
```

Dans la fonction Factorielle on appelle la fonction Factorielle, c'est bien récursif.

Pour N=4:

La fonction 'descend' et appelle chaque fois la factorielle du nombre inférieur.

La fonction Factionnelle est appelée 4 fois :

Factorielle (4) appelle Factorielle(3) qui appelle Factorielle(2) qui appelle Factorielle (1)

Puis la fonction remonte en retournant le résultat de chaque factorielle.

Factorielle (1) retourne 1

Factorielle (2)retourne 2 '2*factorielle(1)

Factorielle (3) retourne 6 '3*factorielle(2)

Factorielle (4) retourne 24 '4*factorielle(3)

Vb gère cela avec **une pile des appels**. il met dans une pile les uns au dessus des autres les appels, quand il remonte, il dépile de haut en bas (Dernier rentré, premier sortie)

Attention: La pile a une taille maximum, si N est trop grand, on déclenche une erreur de type StackOverflow.



Site **LDF** Cours **VB.net**



1.19	Le GOTO		
		Suivant	Sommaire

Faut-il utiliser le GOTO ?

Goto.

Permet un saut non conditionnel : **aller à** ; saute vers une étiquette:

...

Goto FIN

...

FIN:

FIN: est une étiquette, un mot en début de ligne qui désigne un endroit du code; pour créer une étiquette, taper en début de ligne un mot suggestif de l'endroit, puis ajouter ":".

Le programme saute de la ligne **GoTo FIN** à l'étiquette **FIN:** puis se poursuit après **FIN:**

Le GoTo est souvent utilisé avec une instruction If:

If A=0 Then Goto FIN

..

FIN:

L'utilisation du Goto est peu élégante et à éviter; s'il y a plusieurs GoTo le programme devient vite illisible. On peut remplacer avantageusement la ligne précédente par:

```
If A<>0 Then
```

```
  ..
```

```
End if
```



Site LDF Cours VB.net



1.20	<h2 style="text-align: center;">Les Classes, les objets.</h2>	<p style="text-align: center;">Suisvant </p>	<p style="text-align: center;"> Sommaire</p>
------	---	--	--

Résumons un peu la notion de Classe, d'objet, de surcharge, de classe statique.

Classes.

Nous avons vu qu'on utilise des **objets**.

Il existe **des 'types d'objet'** qui définissent les caractéristiques communes des objets. Ces types se nomment les **Classes**.

Exemple:

La Classe System.Windows.Forms contient les 'Forms' et les 'Control'

On rappelle que c'est ces classes que l'on utilise comme 'moule' pour **instancier** (créer) un objet.

`Dim B As New Form` ' crée un formulaire à partir de la Classe Form (Fenêtre).

B hérite de toutes les caractéristiques de la Classe Form.

Essayons de comprendre:

Pour utiliser un objet en VB, il faut.

Que la DLL correspondante soit chargée dans le projet. (La DLL c'est un fichier exécutable '.dll' qui contient le code nécessaire). En VB.NET on appelle cela la '**Référence**'. Exemple de DLL:

```
System.dll
```

Que l'espace de nom soit importé: une DLL contient des Classes d'objet. Pour utiliser une Classe il faut l'inclure dans le programme donc il faut l'importer à partir de la DLL . On va par exemple importer l'espace de nom 'System.Windows.Forms' (contenue dans System.dll et qui contient les Classes 'Form' et 'control'):

```
Import System.Windows.Forms
```

On peut maintenant utiliser les Classes contenues dans cet espace de nom et créer un objet. par exemple on va créer une fenêtre avec la Classe Form contenue dans System.Windows.Forms.

```
Dim Form1 As Form
```

Form1 est donc un objet formulaire qui hérite de tous les membres (Propriétés, méthodes) de la Classe Form, on peut donc utiliser une méthode de cet objet:

```
Form1.Show() 'pour faire apparaître la  
fenêtre
```

ou une propriété:

```
Form1.BackColor=RED 'pour modifier la couleur  
de fond
```

Les Classes les plus courantes sont déjà chargées et disponibles, ce qui simplifie un peu les choses.

Voyons le détail des choses:

Les différentes 'Classes'.

Il existe 3 types de Classes

-Les Classes spécifiques que l'on crée de toute pièce dans les modules de Classe. (On verra cela plus loin)

En VB, on peut créer une classe, ses propriétés, méthodes..

-Les classes prédéfinies du Framework. Ces classes de bases sont regroupées en bibliothèques sous la dénomination 'Espace de noms' et font partie:

- **Du Framework**
- **Du Common Language Runtime.**

Il existe ainsi de manière générale des classes

pour les formulaires Windows
(WindowsForms),

pour le Web (WebForms),

pour l'accès aux données,

les réseaux,

la sécurité....

Quand on crée un nouveau projet, les Classes le plus souvent utilisées sont automatiquement chargées dans le projet:

Voir l'onglet "Explorateur d'objet"

Sont à disposition lors de la création d'un nouveau projet:

▫ **Quelques classes du Framework:**

`System,`

`System.data,`

`System.drawing,`

`System.Windows.forms`

Ce dernier contient les `Controls`

▫ **Le Common Language Runtime (CLR).**

Comme ces Classes sont chargées au départ cela permet d'emblée de créer des feuilles, des contrôles..(qui sont dans les WindowsForms et les Controls).

-Les classes fournies par des tiers.

On peut ajouter des références (DLL) permettant d'ajouter des classes nouvelles, cela permet d'ajouter de nouvelles fonctionnalités à VB: Exemple: les pilotes de base de données..

Imports:

Certains espaces de noms ne sont pas chargés, l'espace de noms `Math` n'est pas chargé par

exemple. (Bien que la référence, la dll qui se nomme System soit présente dans le projet.)

Si je veux utiliser `Round` pour arrondir un nombre il faut d'abord importer l'espace de nom `Math`:

Pour cela il faut taper en haut de la fenêtre (au dessus de `public Class`)

```
Imports System.Math
```

Ensuite, est accepté:

```
Label1.Text = (Round(1.2)).ToString 'qui affiche 1.
```

Si l'Import n'a pas été fait, `System.Math.Round(1.2)` est accepté aussi.

Noter bien que comme `Math` fait partie de `System`, la référence (la DLL correspondante) est déjà chargée.

Autre exemple: si on veut utiliser les fichiers, il faut importer `System.IO`.

Portée de l'espace de noms:

Si un seul espace de noms est spécifié (`Import System`), tous les membres à nom unique de cet espace de noms sont présents. Si un espace de noms et le nom d'un élément de l'espace de noms sont spécifiés (`Import System.Math`), seuls les membres de cet élément sont disponibles sans qualification.

Exemple:

```
Import System
```

Permet d'utiliser `System.ArgumentException` mais pas `Systeme.Math.round`

Pour utiliser Round il faut Importer `System.Math`

Propriété ambiguë:

Certaines propriétés sont communes à plusieurs classes, il peut y a avoir ambiguïté et il faut utiliser dans ce cas la syntaxe complète.

Cela semble le cas pour `left` qui est une propriété de `Microsoft.VisualBasic.Strings` mais aussi une propriété des contrôles.

`MonControle.left=250` est accepté

`Chaine= left(C,2)` pose des problèmes.

Pour lever l'ambiguïté il faut écrire `Microsoft.VisualBasic.left(C,i)` par exemple quand on utilise `left` pour manipuler des chaînes. (C'est ce que j'ai compris!!)

`Chaine= Microsoft.VisualBasic.left(C,2)` est accepté.

Alias:

Parfois pour simplifier l'écriture ou pour éviter des ambiguïtés on peut utiliser des **Alias**:

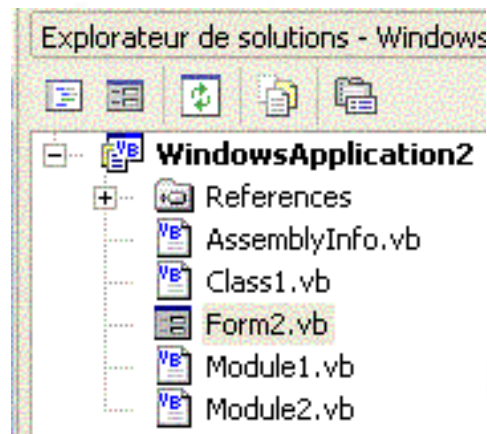
`Imports STR= Microsoft.VisualBasic.Strings` importe l'espace de nom `String` mais le désigne sous le nom de `STR` (`STR` est un Alias); `STR` est utilisé ensuite:

`Chaine=STR.left(C,i)`

Références:

Pour qu'une classe soit utilisée, il faut que le composant correspondant (la DLL) soit chargé; on a vu que par défaut quelques composants du Framework (`System.dll..`) et le CLR (`microsoft.dll`) était chargés.

Dans '**Explorateur de solutions**' double-cliquer la rubrique **références** pour voir les **DLL (références déjà chargées)**.



Si vous souhaitez utiliser un autre composant dans votre application et qu'il n'est pas chargé, il faut ajouter la référence de ce composant. Dans la fenêtre de l'explorateur de solutions, cliquez le bouton droit de la souris puis cliquez sur 'Ajouter une référence'. La boîte de dialogue 'Ajouter une référence de Visual Studio .NET' propose trois options :

- .NET - Répertorie tous les composants .NET Framework pouvant être référencés. (Certains sont déjà chargé comme System..) et les composants extérieurs.
- COM - Répertorie tous les composants COM pouvant être référencés.
- Projets - Répertorie tous les composants réutilisables créés dans des projets locaux.

Exemple:

Pour avoir la compatibilité VB6 par exemple charger [Microsoft.VisualBasic.Compatibility](#) et [Microsoft.VisualBasic.Compatibility.Data](#), vous aurez ensuite accès à cet espace de nom.



En résumé:

Les références (correspondent aux DLL) permettent de charger des composants, des Classes du Framework ou d'autres classes.

L'instruction 'Imports' permet d'importer des espaces de nom venant de ses références.

Cela donne accès dans le programme à des classes appartenant à ces espaces de noms. on pourra donc instancer des objets grâce à ces Classes puis utiliser des méthodes.

Noter que dans les Classes, il existe une structure arborescente.

La premier Classe (en haut) est `System`.

Dessous il y a entre autres `System.WindowsForms`

Dessous `System.WindowsForms.Controls`

Enfin `System.WindowsForms.BackColor` par exemple.

Remarque sur les objets et leurs méthodes :

Héritage:

Les classes héritent des membres (propriétés, méthodes) dont elles sont issues:

Exemple :

Un contrôle `Button` hérite des membres de `System.Windows.Forms`. (les propriétés `Name`, `Left`, `Right`, `Visible`, `Enabled`.. par exemple.)

L'objet lui-même a **des membres** (propriétés et méthodes) :

▫ **Elles sont accessibles directement :**

Exemple pour **.Trim** de la chaîne A

```
A.Trim(" ")
```

A étant une String, la méthode **.trim** relative au String est utilisable.

on appelle cela une méthode d'instance car elle travaille sur une instance d'un objet.

- D'autres sont accessibles uniquement par **les méthodes de la classe :**

'La Classe de leur nature, de leur type.

Dans la classe 'String' j'utilise la méthode **Compare** pour comparer 2 chaînes.

```
c=String.Compare(a,b)
```

'La Classe de l'opération à effectuer

Dans la Classe Math j'utilise la méthode **Abs** (Valeur absolue)

```
c=Math.Abs(-12)
```

On appelle cela une méthode partagée car on l'utilise directement à partir d'une classe.

Langage Visual Basic.

Dans VB.Net il y a donc possibilité de travailler avec:

- **Les Classes**, leurs propriétés, leurs méthodes. Les Forms, Controls, les classes des variables (String, Int32..) sont disponibles par défaut.
- **Les instructions VB** du Common Langage Runtime disponible par défaut.

- Les instructions de la **bibliothèque de compatibilité VB6**. il faut dans ce cas importer [Microsoft.VisualBasic.Compatibility](#) et [Microsoft.VisualBasic.Compatibility.Data](#)

En effet, par défaut, par soucis de faciliter les migrations de VB6 vers VB.NET, VB.NET contient toutes les fonctions venant de VBA.

Exemple:

Pour la manipulation des nombres

'Int' fait partie du CLR,

'Round' fait partie de la classe Math,

'Randomize' et 'Rnd' font partie de la bibliothèque de compatibilité.

Parfois certaines fonctions font double emploi et ont des équivalents dans les 2 ou 3 catégories.

Les Classes sont souvent plus riches avec multiples surcharges et, si j'ai bien compris, comme le CLR, sont communes à tous les langages utilisant le Framework .Net. Si vous voulez passer au C#, les classes sont les mêmes.

Par contre, les instructions devant de la compatibilité VB6 sont propre à VB.NET. Seront-elles conservées dans les futures versions de VB.NET?

Classe statique

Certaines Classe sont dites **Statiques** car elles existent d'emblé et on peut travailler dessus sans que l'on aie besoin de les instancer:

Exemple:

La Classe Directory (répertoire):

[Directory.GetCurrentDirectory](#) 'est utilisable directement pour obtenir le répertoire courant.

Par contre avec **une Classe non statique** il faut instancer

l'objet que l'on va utiliser:

Pour la classe DirectoryInfo (information sur le répertoire), on doit instancer avant usage un DirectoryInfo particulier:

```
Dim D As DirectoryInfo
```



```
D= New DirectoryInfo( MonDossier)
```

C'est un peu théorique, mais on verra au fur et à mesure des exemples pratiques de cela.



Site LDF Cours VB.net



E 1.1	Exemples : Petites routines.	Suisvant 	 Sommaire
-------	-------------------------------------	--	--

On prendra des exemples de routines très simples ne contenant que du code:

- q Avec les strings
- q Avec les nombres

Avec les 'String'.

Vous avez une chaîne de caractères, comment afficher, le premier caractère puis les 2 premiers, puis 3..?

Dans un formulaire (une fenêtre), il y a un TextBox1(zone de texte) (avec sa propriété Multiline=True)

```
Dim C As String = "DUBONET"
Dim Tx As String
Dim i As Integer
For i = 1 To Len(C)
    Tx += Microsoft.VisualBasic.Left(C, i) +
    ControlChars.CrLf
Next i
TextBox1.Text = Tx
```

Mettre ce code dans Form_Load puis lancer le programme.

Affiche:

D
DU
DUB
DUBO
DUBON
DUBONE
DUBONET

On remarque: Tx est une string permettant de stocker temporairement la string à afficher; a chaque boucle on ajoute la nouvelle string (Tx += est équivalent à Tx=Tx+..) et un caractère de retour à la ligne.

Left fait partie de l'espace de nom Microsoft.VisualBasic.

Avec les 'nombres'.

Somme de N entiers.

Calculer par exemple pour Nombre=20 la
Somme=1+2+3+4...+18+19+20

```
Dim Somme As Integer      'Variable somme
Dim Nombre As Integer=20
Dim i As Integer          'Variable de boucle
```

```
For i=0 To Nombre
    Somme += Nombre
Next i
```

On rappelle que `Somme += Nombre` est équivalent à `Somme =Somme+ Nombre`

Afficher les tables de multiplication.

On fait 2 boucles:

celle avec i (qui décide de la table: table des 1, des 2..)

On affiche 'table des' puis valeur de i

celle avec j (allant de 1 à 10 pour chaque table)

Pour chaque ligne, on affiche la valeur de i puis ' X '
puis la valeur de j puis ' = ' puis la valeur de i fois j

ControlChars.CrLf permet un saut à la ligne

A chaque fois que l'on a quelque chose à afficher, on l'ajoute à la variable String T

A la fin on affecte T à la propriété text d'un TextBox pour rendre visible les tables.

```
Dim i As Integer
Dim j As Integer
Dim T As String
```

```
For i = 1 To 10
    T += ControlChars.CrLf
    T += "Table des " & i & ControlChars.CrLf
    For j = 1 To 10
        T += i.ToString & " X " & j.ToString & "=" & i * j
    & ControlChars.CrLf
    Next j
Next i
TextBox1.Text = T
```

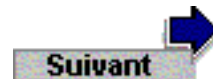
Affiche:

Table des 1

1 X 1 =1



1 X 2 =2

...



Site **LDF** Cours **VB.net**



<p>E 1.2</p>	<p>Exemples :</p> <p>Petits programmes de maths.</p>	<p>Suivant </p>	<p> Sommaire</p>
---------------------	--	--	---

On prendra des exemples de routines mathématiques simples:

- q Calcul de l'hypoténuse d'un triangle rectangle
- q Calcul de factorielle (avec ou sans récursivité)
- q Un nombre est-il premier?

Calcul de l'hypoténuse d'un triangle rectangle.

On crée pour cela une fonction, on envoie 2 paramètres de type Single: les 2 cotés du triangle; la fonction retourne l'hypoténuse.

```
Function Hypotenuse (ByVal Side1 As Single, ByVal Side2 As Single) As Single
    Return Sqrt((Side1 ^ 2) + (Side2 ^ 2))
End Function
```

Pour les nuls, on rappelle que le carré de l'Hypoténuse est égal à la somme des carrés des 2 autres cotés.

Factorielle.

On rappelle que $N!$ (factorielle N) = $1*2*3*...*(N-2)*(N-1)*N$

Exemple Factorielle 3 =1*2*3

```
Dim R As Long
```

```
R=Factorielle(3) 'retournera 6
```

Cette fonction n'est pas fournie par VB, créons **une fonction Factorielle**:

```
Function Factorielle (ByVal N as Long) As Long
    Dim i As Long
    Resultat=1
    For i= 1 to N
        Resultat=i* Resultat
    Next i
    Return Resultat
end Function
```

Cela crée une fonction recevant le paramètre N et retournant un long.

Une boucle effectue bien 1*2*3...*N-1*N.

Factorielle avec 'Récursivité':

Une autre manière de calculer une factorielle est d'utiliser la **récursivité**:

Une procédure est récursive si elle peut s'appeler elle même.

VB gère la récursivité.

Comment faire pour les factorielles?

On sait que Factorielle N= N * Factorielle(N-1)

N!= N*(N-1)! : en sachant que 1!=1

Créons la fonction:

```
Function Factorielle (ByVal N as Long) As Long
    If N=1 then
        Return 1
    Else
        Return N* Factorielle(N-1)
    End If
end Function
```

Dans la fonction Factorielle on appelle la fonction Factorielle, c'est bien récursif.

Pour N=4, la fonction Factorielle est appelée 4 fois :Factorielle (4) puis Factorielle(3) puis Factorielle(2) puis Factorielle (1)

Factorielle (1) retourne 1

Factorielle (2)retourne 2 '2*factorielle(1)

Factorielle (3)retourne 6 '3*factorielle(2)

Factorielle (4) retourne 24 '4*factorielle(3)

Vb gère cela avec **une pile des appels**. il met dans une pile les uns aux dessus des autres les appels, quand il remonte, il dépile de haut en bas (Dernier rentré, premier sortie)

Attention: La pile a une taille maximum, si N est trop grand, on déclenche une erreur de type StackOverflow.

Un nombre est-il premier?

Un nombre premier est seulement divisible par 1 et lui-même.

Pour voir si N est entier on regardera successivement si ce nombre est divisible par 2 puis 3 puis 4... jusqu'a N-1

Un nombre est divisible par un autre si la division donne un entier. Comment voir si un nombre est entier? Pour ma part, j'utilise la méthode suivante: A est entier si $A = \text{Int}(A)$.

```
Dim IsPremier As Boolean
Dim N As Double=59
Dim I As Double
I=2: IsPremier=True
Do
    If N/I= Int(N/I) then
        IsPremier=False
    Else
        i += 1
    End if
```

Loop While IsPremier=True And I<N

Pour 59 IsPremier sera égal à True.

On peut améliorer la routine en remarquant:

Si un nombre n'est pas premier il admet 2 diviseurs dont un est inférieur à racine N.

on peut donc :

-Vérifier que le nombre n'est pas pair puis

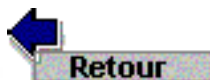
-Vérifier s'il est divisible par les nombres allant de 3...jusqu'à racine de N en ne tenant compte que des nombres impaires.

Remarque pour ceux qui veulent tester le code :

Pour utiliser la routine sur les nombres premiers, il faut créer une petite interface: dans un formulaire créer un bouton nommé 'Button1' et une TextBox nommée 'TextBox1', enfin mettre dans la routine Button1_Click le code ci-dessous.



Quand on lance le programme, on saisie un nombre dans le textbox , puis on clique sur le bouton , cela affiche True ou False dans une MessageBox, is le nombre est premier ou non:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim IsPremier As Boolean
    Dim N As Double
    Dim I As Double
    N = Val(TextBox1.Text)
    I = 2 : IsPremier = True
    Do
    If N / I = Int(N / I) Then
    IsPremier = False
    Else
    I += 1
    End If
    Loop While IsPremier = True And I < N
    MsgBox(IsPremier.ToString)
End Sub
```



Site **LDF** Cours **VB.net**



E 1.3	Exemples : Programmes de Tri et de recherche.		
--------------	--	--	---

On a parfois besoin de trier par ordre alphabétique un tableau de string.

Il existe maintenant des méthodes de tri 'automatique' entièrement gérées par VB grâce à la méthode 'sort'.

Il existe aussi des routines de tri entièrement écrites en VB, elles deviennent inutiles mais c'est didactique de voir comment elles fonctionnent.

Parfois il faut chercher dans un tableau un élément; la aussi on peut écrire les routines ou utiliser les méthodes VB

Tri avec la méthode SORT.

Pour un tableau unidimensionnel.

```
Dim Animals(2) As String
Animals(0) = "lion"
Animals(1) = "girafe"
Animals(2) = "loup"
Array.Sort(Animals)
```

Et le tableau est trié!!

On rappelle que l'on ne peut pas trier un tableau multidimensionnelle, mais il y a des ruses. (voir rubrique:tableau)

Les Collections peuvent être triées automatiquement aussi.

Enfin si la propriété **Sorted** d'une ListBox est à **True**, la liste est triée automatiquement quand on la charge.

...

Routine de Tri.

Pour trier un tableau de chaînes de caractères, il faut comparer 2 chaînes contiguës, si la première est supérieure (c'est à dire après l'autre sur le plan alphabétique) on inverse les 2 chaînes, sinon on n'inverse pas. Puis on recommence sur 2 autres chaînes en balayant le tableau jusqu'à ce qu'il soit trié.

Tout l'art des routines de tri est de faire le moins de comparaisons possible pour trier le plus vite possible.

Voyons une des routines les plus rapides, le **Bubble Sort** (ou tri à bulle); on le nomme ainsi car l'élément plus grand remonte progressivement au fur et à mesure jusqu'à la fin du tableau comme une bulle.

Une boucle externe allant de 1 à la fin du tableau balaye le tableau N fois, une seconde boucle interne balaye aussi le tableau et compare 2 éléments contigus et les inverse si nécessaire. La boucle interne fait remonter 1 élément vers la fin du tableau, la boucle externe le fait N fois pour remonter tous les éléments.

```
Dim i, j , N As Integer      'Variable de boucle i, j ; N=
                             nombre d'éléments-1
Dim Temp As String
N=4 'tableau de 5 éléments.
Dim T(N) As String 'élément de 0 à 4
For i=0 To N-1
    For j=0 To N-1
        If T(j)>T(j+1) then
            Temp=T(j): T(j)=T(j+1):T(j+1)=Temp
        End if
    Next j
Next i
```

Next i

Remarque: pour inverser le contenu de 2 variables, on doit écrire

`Temp=T(j): T(j)=T(j+1):T(j+1)=Temp` (L'instruction qui faisait cela en VB6 et qui se nommait Swap n'existe plus)

Cette routine tri bien le tableau mais n'est **pas optimisée**: il n'est pas nécessaire que la boucle interne tourne de 0 à N-1 à chaque fois car après une boucle, le dernier élément est à sa place. Pour i=0 la boucle interne tourne jusqu'à N-1, pour i=1 jusqu'à N-2...

Cela donne:

```
Dim i, j, N As Integer 'Variable de boucle i, j ; N=
nombre d'éléments-1
Dim Temp As String
N=4 'tableau de 5 éléments.
Dim T(N) As String 'élément de 0 à 4
For i=0 To N-1
    For j=0 To N-i-1
        If T(j)>T(j+1) then
            Temp=T(j): T(j)=T(j+1):T(j+1)=Temp
        End if
    Next j
Next i
```

Il existe d'autres méthodes encore plus rapides (Méthode de Shell et Shell-Metzner).

Recherche dans une liste.

On a une liste de string, on veut chercher où (en quelle position) se trouve une string.

Pour une liste non triée, on n'a pas d'autres choix que de comparer la string cherchée à chaque élément du tableau, on utilisera donc une boucle:

```
N=4 'tableau de 5 éléments.
Dim T(N) As String 'élément de 0 à 4
T(0)="vert"
T(1)="bleu"
T(2)="rouge"
T(3)="jaune"
```

```

T(4)="blanc"
Dim i As Integer 'Variable de boucle
Dim AChercher As String= "rouge" 'String à chercher
For i=0 To N
    If T(i)=AChercher then
        Exit For
    End if
Next i

```

'i contient 2

Pour une liste triée (suite ordonnée), on peut utiliser la méthode de recherche dichotomique: On compare l'élément recherché à l'élément du milieu du tableau, cela permet de savoir dans quelle moitié se situe l'élément recherché. De nouveau on compare à l'élément recherché à l'élément du milieu de la bonne moitié...jusqu'à trouver. Pour cela on utilise les variables Inf et Sup qui sont les bonnes inférieure et supérieure de la zone de recherche et la variable Milieu. On compare l'élément recherché à l'élément du tableau d'indice milieu, si il sont égaux on a trouvé , on sort; si ils sont différent on modifie Inf et Sup pour pointer la bonne plage puis on donne à Milieu la valeur du milieu de la nouvelle plage et on recommence.

```

Dim N As Integer
Dim T(N) As String 'élément de 0 à 4
Dim Inf, Sup, Milieu As Integer '
Dim Reponse As Integer 'contient le numero de l'élément
                        'ou -1 si élément non trouvé
Dim i As Integer 'Variable de boucle
Dim AChercher As String= "c" 'String à chercher

```

N=4 'tableau de 5 éléments.

```

T(0)="a"
T(1)="b"
T(2)="c"
T(3)="d"
T(4)="e"

```

Inf=0: Sup=N

Do

```

    if inf>Sup then Reponse=-1: Exit Do
    Milieu= INT((Inf+Sup)/2)
    If Achercher=T(Milieu) then Reponse=Milieu:Exit Do
    If Achercher<T(Milieu) then Sup=Milieu-1
    If Achercher>T(Milieu) then Inf=Milieu+1

```

Loop

'Reponse =2

La recherche dichotomique est rapide car il y a moins de comparaisons.

Mais comme d'habitude VB.Net possède des propriétés permettant de rechercher dans un tableau trié ou non et cela sans avoir à écrire de routine.

Binarysearch recherche un élément dans un tableau **trié** unidimensionnel.(algorithme de comparaison binaire performant sur tableau trié: probablement une recherche dichotomique)

Exemple :

```
I=Array.BinarySearch(Mois, "Février")
```

IndexOf

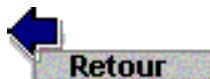
Recherche un objet spécifié dans un tableau unidimensionnel (trié ou non), retourne l'index de la première occurrence.

```
Dim myIndex As Integer = Array.IndexOf(myArray, myString)
```

```
□
```

Retourne -1 si l'élément n'est pas trouvé.

LastIndexOf fait une recherche à partir de la fin.



Site **LDF** Cours **VB.net**



E 1.4	Exemples : Petits calculs financiers.	Suivant →	← Sommaire
--------------	--	------------------	-------------------

Conversion Francs=>Euros.

Si un objet coute 100F, cela fait combien d'euros?

```
Dim Valeur As Double=100
Dim Resultat As Double
```

```
Resultat =Math.Round((Valeur / 6.55957), 2)
```

On divise par 6.55957 puis on arrondi à 2 chiffres après la virgule.

Coût d'augmentation de la vie.

Si un objet de 100 augmente de 3% par an, combien coûtera -t-il dans 10 ans.

```
Dim Prix As Decimal=100
Dim Taux As Decimal=3
Dim Periode As Integer=10
Dim i As Integer
For i= 1 to Periode
    Prix=Prix+(Prix*Taux/100)
```

Next i

On peut remplacer les 3 dernières lignes par:

```
Prix=Prix*(1+Taux/100)^Periode
```

Noter que l'on utilise des variables de type décimales, c'est une bonne habitude pour faire des calculs financiers (pas d'erreurs d'arrondis).

Remboursement d'un prêt.

Quel est le remboursement mensuel d'un prêt d'une somme S durant une durée D (en année) à un taux annuel T ?

$R = S \times T / 1 - (1+T)^{-D}$ (ici avec T en % mensuel et D en mois)

```
Dim R, S , D, T As Decimal
```

```
S=5000 '5000
```

```
D=15 'Sur 15 ans
```

```
T=4 '4% par an
```

```
T=T/12/100 'Taux au mois
```

```
D=D*12 'Durée en mois
```

```
R=S*T/(1-(T+1)^(-D))'Formule connue par tous bon comptable!!
```

Si on voulait afficher le résultat dans un label (on verra cela plus loin)

```
Label1.text= R.ToString("C")
```

Ici le résultat est transformé en chaîne de caractères (grâce à ToString) au format monétaire ("C"), on obtient '36,98 ' que l'on met dans le label pour l'afficher.

Ultérieurement on verra un exemple plus complet utilisant les fonctions financières de VB.



3.1	L'interface Utilisateur.	Suivant 	 Sommaire
-----	--------------------------	---	--

Elle correspond aux fenêtres et contrôles que voit l'utilisateur.

On a vu que le développeur **dessine** cette interface en mode conception (Design) dans l'IDE.

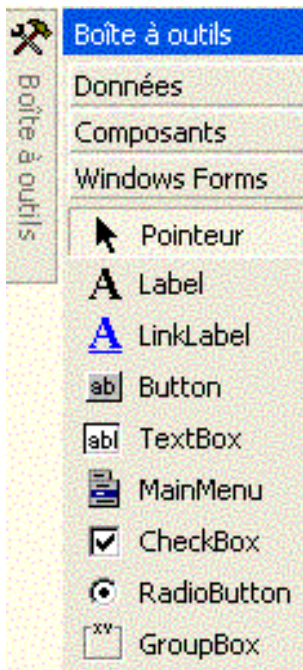
Rappel:

Comment créer une fenêtre?

Menu **Projet, Ajouter un formulaire Windows**, cliquer sur **WindowsForm**, une fenêtre 'Form1' apparaît. On a bien créé une fenêtre avec la classe `WindowsForms.Form` (En fait on a créé une Classe 'Form1')

Comment ajouter un bouton?

Cliquer sur '**Boite à Outils**' à gauche, bouton `WindowsForms`, puis bouton '**Button**', cliquer dans `Form1`, déplacer le curseur sans lâcher le bouton, puis lâcher le bouton : un bouton apparaît.

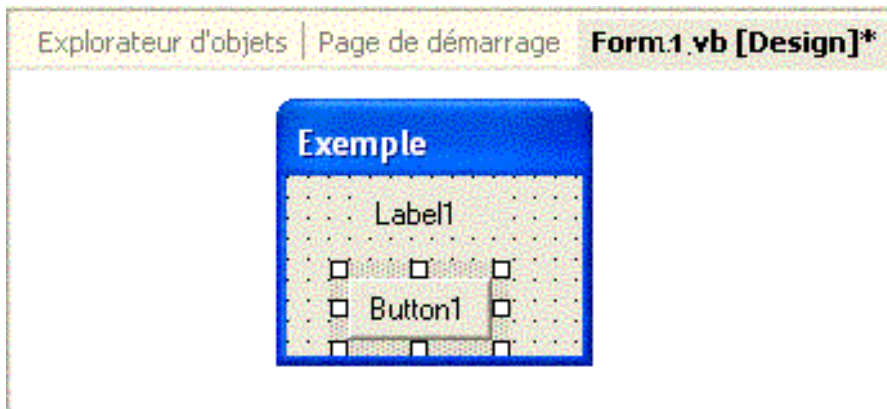


Comment ajouter un label?

Un label est un contrôle qui permet d'afficher un texte.

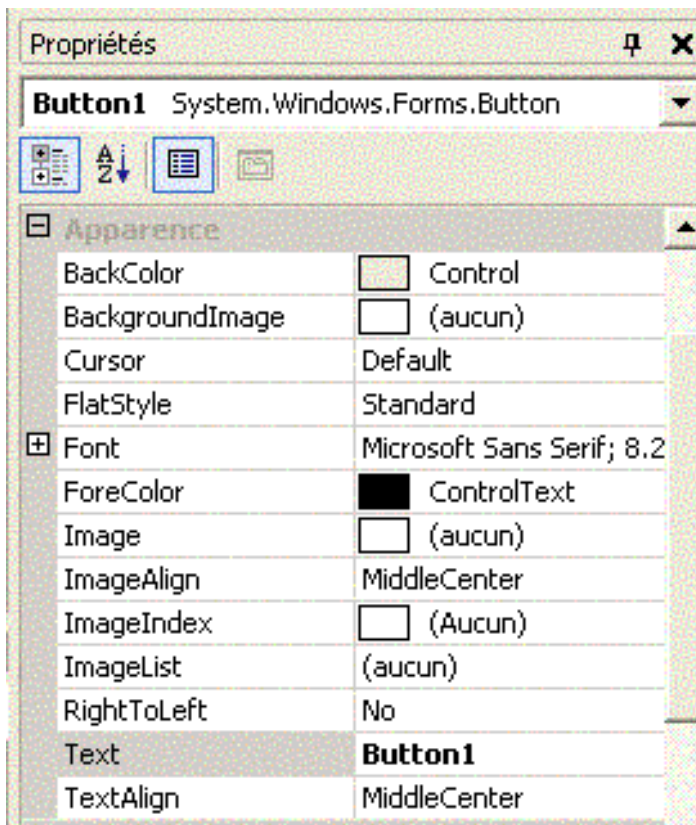
Comme pour le bouton cliquer sur 'Boîte à Outils' à gauche, bouton WindowsForms, bouton 'Label' et mettre un contrôle label sur la fenêtre.

On obtient dans la fenêtre principale:



Modifier les propriétés de l'objet.

Il suffit ensuite de modifier les propriétés de l'objet pointé (celui qui est entouré de petit carrés) pour lui donner l'aspect désiré. Les propriétés sont accessibles dans la **fenêtre de propriétés** de droite.



Dans le code des procédures les propriétés des objets sont aussi accessibles.

`Button1.Text="OK"` permet par exemple de modifier la propriété **Text** d'un bouton.

Noter que **pour modifier la taille des objets**, on peut le faire très facilement à la souris en cliquant sur les petits carré entourant l'objet et en tirant les bords. (on peut interdire les modifications de taille et de position des contrôles par le menu *Format* puis *verrouiller les contrôles* une fois que leurs tailles et positions est bien définies.)

Tous les objets ont des propriétés communes.

Celles héritées de la Classe '**Control**' qu'il faut connaître:

Name : il s'agit du nom de l'objet tel qu'il est géré par l'application.

Par défaut, VB baptise tous les objets que vous créez de noms génériques, comme Form1, Form2, Form3 pour les fenêtres, List1, List2 pour les listes...

Accessible en mode conception uniquement.



Il est vivement conseillé, avant toute autre chose, de rebaptiser les objets que vous venez de créer afin de donner des noms plus évocateurs.

Le bouton sur lequel est écrit « OK » sera nommé `BoutonOK`.

La liste qui affiche les utilisateurs sera nommée `ListUtilisateurs`.

Il est conseillé de débiter le nom de l'objet par un mot évoquant sa nature:

`BoutonOk` ou `BtOk` ou `ButtonOk`, `btnOk` c'est comme vous voulez.

Microsoft conseille:

`btn` pour les Boutons

`lst` pour les ListBox

`chk` pour les CheckBox

`cbo` pour les combos

`dlg` pour les DialogBox

`frm` pour les Form

`lbl` pour les labels

`txt` pour les Textbox

`tb` pour les Toolbar

`rb` pour les radiobutton

`mm` pour les menus

`tmr` pour les timers

Text : il s'agit du texte qui est associé à l'objet.

Dans le cas d'une fenêtre c'est le texte qui apparaît dans la barre de titre en haut.

Pour un TextBox ou un Label c'est évidemment le texte qui est affiché.

On peut modifier cette propriété en mode conception ou dans le code :

Exemple : Avec du code comment faire pour que le bouton `ButtonOk` porte l'inscription 'Ok'

```
ButtonOk.Text= "Ok"
```

Enabled : accessible

Indique si un contrôle peut répondre à une interaction utilisateur.

La propriété Enabled permet l'activation ou la désactivation des contrôles au moment de l'exécution. Vous pouvez désactiver les contrôles ne s'appliquant pas à l'état actuel de l'application. Vous pouvez également désactiver un contrôle pour interdire son utilisation. Par exemple, un bouton peut être désactivé pour empêcher l'utilisateur de cliquer dessus. Si un contrôle est désactivé, il ne peut pas être sélectionné. Un contrôle désactivé est généralement gris.

Exemple : désactiver le ButtonOk

```
ButtonOk.Enabled=False
```

Visible :

Indique si un contrôle est visible ou non.

```
ButtonOk.Visible=False
```

 fait disparaître le bouton.

Attention pour rendre visible une fenêtre on utilise la méthode .Show.

Font :

Permet le choix de la police de caractères affichée dans l'objet.

BackColor ForeColor:

Couleur du fond, Couleur de l'avant plan

Pour un bouton Forecolor correspond au cadre et aux caractères.

```
ButtonOk.ForeColor=  System.Drawing.Color.Blue
```

Tag:

Permet de stocker une valeur ou un texte lié à l'objet. Chaque objet a un Tag qui peut contenir du texte.

On l'utilise souvent comme un Flag lié à l'objet.

Par exemple: une liste peut contenir la liste des CD ou des DVD ou des K7, quand je charge la liste des CD, je rajoute

List1.Tag="CD" cela permet ultérieurement de voir ce qu'il y a dans la liste.

Il y a bien d'autres propriétés.

Evènements liés aux objets

On a vu que les objets de l'interface utilisateur ont des **procédures déclenchées par les évènements de cet objet.**

2 exemples:

1- Quand l'utilisateur clique sur un bouton Ok , la procédure `ButtonOk_Click` s'effectue.

2-Quand l'état (coché ou non coché) d'une case à cocher nommée CouleurRouge change, la procédure `CouleurRouge.CheckedChanged` est activée.

La syntaxe **complète** de la procédure est:

```
Private Sub CouleurRougeCheckedChanges (ByVal sender As System.Objet, ByVal e As System.EventArgs) Handles CouleurRouge.CheckedChanged
```

```
End Sub
```

Détaillons:

La procédure évènement est **privée** (Private).

Après le nom Sub il y a **un nom de procédure** (CouleurRougeCheckedChanges)

Handles indique quel objet et évènement à déclenché la procédure. (On verra qu'il peut y en avoir plusieurs.)

A noter que **Sender** contient le contrôle ayant déclenché l'évènement et **e** l'évènement correspondant.

`sender.Name` contient par exemple le nom du contrôle ayant déclenché l'évènement.

On voit que quand on crée un objet, ses procédures évènements sont automatiquement créés.

On se rend compte que dans une procédure évènement on peut modifier (en mode conception) ou lire (en mode Run) quel objet et quel évènement à déclenché la procédure. On peut même indiquer plusieurs objets liés à cette procédure.

Certains évènements sont communs à tous les contrôles:

Click

DoubleClick

GotFocus

LostFocus

KeyUp

KeyPress

KeyDown

Il y a toujours des méthodes **Changed** déclenchées par un changement d'état: **CheckedChanged** pour une case à cocher, **TextChanged** pour un contrôle texte.

Pour ne pas alourdir les exemples, nous écrivons souvent une version simplifiée de l'en-tête de la procédure.

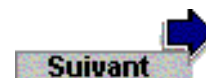
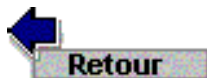
En résumé:

Le programmeur dessine les fenêtres et contrôles.

Il peut modifier les propriétés des objets dessinés:

Par la **fenêtre de propriétés (en mode conception)**.

Par **du code (des instructions) dans les procédures**.



Site **LDF** Cours **VB.net**



3.2	Les Forms.		
		Suivant	Sommaire

Elle correspondent aux fenêtres ou 'formulaires'.

Créer une fenêtre en mode conception:

Menu **Projet**, **Ajouter un formulaire Windows**, cliquer sur **WindowsForm**, une fenêtre 'Form1' apparaît. On a bien créé une fenêtre avec la classe `WindowsForms`.

Toute l'interface se trouve sur des fenêtres.

En VB.net on parle de **formulaire**.



Propriétés:

Bien sur, la fenêtre possède des propriétés qui peuvent être modifiées en mode design dans la fenêtre 'Propriétés' à droite ou par du code:

Name :Nom du formulaire.

Donner un nom explicite. `FrmDemarrage`

Dès qu'une fenêtre est créée on modifie immédiatement ses propriétés en mode conception pour lui donner l'aspect que l'on désire.

Text : C'est le texte qui apparaîtra dans la barre de titre en haut.

Text peut être modifié par le code : `Form1.text= "Fenêtre"`

Icon : propriété qui permet d'associer à la Form un fichier icône. Cette icône s'affiche dans la barre de titre, tout en haut à gauche. Si la Form est la Form par défaut du projet, c'est également cette icône qui symbolisera votre application dans Windows.

Comment créer une icône?

Dans l'IDE de VB.

Menu Fichier>Nouveau>Fichier
cliquez sur Icon , Vb ouvre une fenêtre Icon1 (dans l'éditeur d'images de Visual Studio.Net) Cela permet de créer ou modifier une icône (Fichier>Ouvrir>Fichier pour modifier).

Comment enregistrer? Click droit dans l'onglet 'Icon1' ouvre un menu contextuel permettant d'enregistrer votre Icône.

WindowState :

Donne l'état de la fenêtre : Plein écran
(`FormWindowState.Maximized`), normale
(`FormWindowState.Normal`), dans la barre de tache
(`FormWindowState.Minimized`).

Exemple: mettre une fenêtre en plein écran avec du code.

`me.WindowState =FormWindowState.Maximized`

(Quand on tape `Me.WindowsState= Vb` donne la liste, l'énumération)

ControlBox

Si cette propriété à comme valeur `False`, les boutons de contrôle situés à droite de la barre de la fenêtre n'apparaissent pas.

MaximizeBox

Si cette propriété à comme valeur `False`, le boutons de contrôle 'Plein écran' situés à droite de la barre de la fenêtre n'apparaît pas.

MinimizeBox

Si cette propriété à comme valeur `False`, le boutons de contrôle 'Minimize' situés à droite de la barre de la fenêtre n'apparaît pas.

FormBorderStyle

Permet de choisir **le type des bords de la fenêtre** : sans bord (`None`), bord simple (`FixedSingle`)ne permettant pas à l'utilisateur de modifier la taille de la fenêtre, bord permettant la modification de la taille de la fenêtre (`Sizable`)..

Exemple:

`Me.FormBorderStyle =FormBorderStyle.Sizable`

StartPosition :

Permet de choisir la position de la fenêtre lors de son ouverture.

Fenêtre au centre de l'écran ? à la position qui existait lors de la conception ...?

`Me.StartPosition =FormStartPosition.CenterScreen`

MinSize et MaxSize

Donne les dimensions minimums et maximums que l'on peut utiliser pour redimensionner une fenêtre.

Opacity

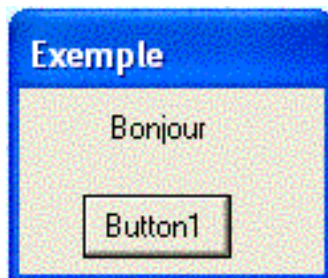
Allant de 0% (0) à 100% (1), permet de créer un formulaire plus ou moins transparent.

Pour 0 il est transparent, pour 1 il est totalement opaque (normal)

Exemple:

```
Me.FormBorderStyle= Sizable
Me.ControlBox=False
Me.Size = New Size(100, 100)
Me.StartPosition = FormStartPosition.CenterScreen
Me.Opacity= 0.75
```

Donne au milieu de l'écran, la fenêtre:



Ouvrir une fenêtre

On vient de dessiner une Form1 et une Form2 c'est donc les Class 'Form1 et ''Form2' (les moules) que l'on a dessiné.

Si dans une routine de la Form1 on veut **ouvrir une seconde fenêtre de type Form2**, il faut:

Créer un Objet fenêtre (formulaire) avec le moule Form2:

```
Dim f As New Form2()
```

La nouvelle instance f de la Class 'form2' est un objet fenêtre.

Pour la faire apparaître j'utilise la méthode:
`.ShowDialog`.

`f.ShowDialog()`

La fenêtre `f` est **modale car on a utilisé `ShowDialog`** : quand elle est ouverte, on ne peut pas aller dans une autre fenêtre de l'application avant de sortir de celle là. (A titre d'exemple les fenêtres `MessageBox` sont toujours Modales).

Utiliser `.show` pour ouvrir une feuille non modale.

Attention: une fenêtre est un objet et est 'visible' suivant les règles habituelles des objets.

Si on instance une fenêtre à partir d'une procédure, elle sera visible dans cette procédure. Si elle est 'Public' et instancée dans un module standard, elle sera visible partout.

Evènements:

Au cours de l'exécution:

Quand la feuille est **chargée** la procédure `Form1_Load()` est activée.

On pourra donc y mettre le code initialisant la feuille.

`Form1_Activated()` est exécuté ensuite car la feuille deviendra active.

`Form1.GotFocus()` est enfin exécuté puisque la fenêtre prend le focus.

`Form1.Enter ()` est exécuté lorsque l'utilisateur entre dans la fenêtre.

Dès qu'une propriété change de valeur un évènement 'PropriétéChanged' se déclenche:

`Form1.BackColorChanged` se déclenche par exemple quand la couleur du fond change.

Form1.Resized se déclenche quand on modifie la taille de la fenêtre. (c'est intéressant pour interdire certaines dimensions)

Form1.Leave survient dans il y a perte du focus.

Bien sur il existe aussi **Form1.Desactivate** quand la fenêtre perd le focus et n'est plus active.

Form1.Closing se produit pendant la fermeture de la fenêtre (on peut annuler cette fermeture en donnant à la variable Cancel la valeur True)

Form1.Closed se produit lorsque la fenêtre est fermée.

Il y en a beaucoup d'autres comme par exemple les évènements qui surviennent quand on utilise la souris (MouveUp, MouseDown, MouseMove) ou le clavier (KeyUp, KeyDown, KeyPress) sur la fenêtre ; Left Right, Size, Position pour positionner la fenêtre ou définir sa taille..

Méthodes :

On a déjà vu que pour faire apparaître une fenêtre il faut utiliser **.ShowDialog** (pour qu'elle soit modale) ou **.Show** (pour non modale).

Me.Close ferme le formulaire.

Me.Activate l'active s'il est visible

Me.Hide rend la fenêtre invisible.

Pour mettre la fenêtre au premier plan.

TopMost laisse toujours la fenêtre au premier plan alors que **BringToFront** ramène la fenêtre au premier plan temporairement.

System.Windows.Forms.Form

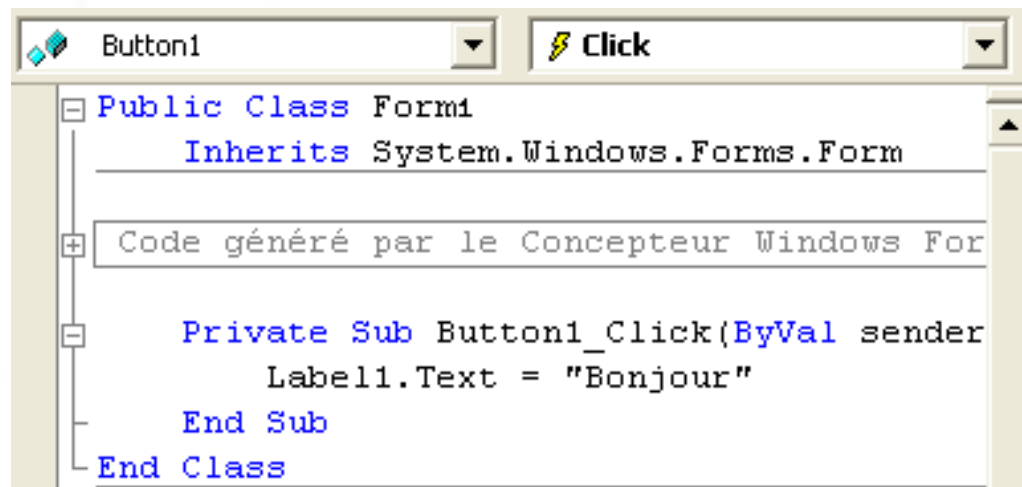
On se rend compte que quand on dessine une fenêtre Form2 par exemple, VB crée une nouvelle classe '**Class Form2**'

```
Public Class Form2
```

```
End Class
```


Elle hérite de **System.Windows.Forms.Form**: on voit bien dans le code:

Inherits System.Windows.Forms.Form



```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Code généré par le Concepteur Windows For

    Private Sub Button1_Click(ByVal sender
        Label1.Text = "Bonjour"
    End Sub
End Class
```

Elle contient

-du code généré automatiquement par le concepteur Windows Forms (on peut le voir en cliquant sur le petit '+') et qui crée la fenêtre et ses contrôles.

-les procédures liées aux évènements.

Quand on tape `Dim f As New Form2()`, on crée une instance de la Class Form2.

Formulaire d'avant plan:

Pour définir au moment de la conception un formulaire en tant que formulaire d'avant-plan d'une application.

- Dans la fenêtre **Propriétés**, attribuez à la propriété **TopMost** la valeur **true**.

Pour définir par code un formulaire en tant que formulaire d'avant-plan d'une application.

- Dans une procédure, attribuez à la propriété **TopMost** la valeur **true**.

```
Me.TopMost = True
```

 **Retour**

 **Index**

 **Sommaire**

Suivant 

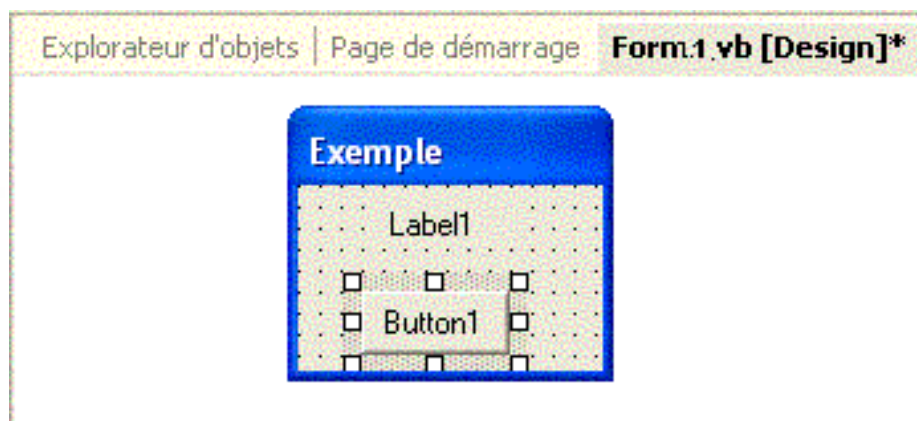
Site LDF Cours VB.net



Ils sont omniprésents dans les 'formulaires'.

Créer un bouton :

Cliquer sur 'Boite à Outils' à gauche , bouton Windows Forms, puis bouton 'Button', cliquer dans Form1, déplacer le curseur sans lâcher le bouton, puis lâcher le bouton : un bouton apparaît.



Modifier ses propriétés:

Name est utilisé pour lui donner un nom explicite (BoutonOk BoutonCancel)

FlatStyle donne un aspect au bouton (Flat, standard, System)



Text contient le texte à afficher sur le bouton. **ForeColor** correspond à la couleur de ce texte (**BackColor** étant la couleur du fond)

Si on y inclut un « & » la lettre qui suit sera souligné et sert de raccourci clavier. &Ok donne sur le bouton Ok

TextAlign permet de positionner le texte dans le bouton.

Image contient le nom de l'image à afficher sur le bouton (si on veut afficher une image, on le fait en mode Design; noter que quand on distribue l'application, il n'y a pas besoin de fournir le fichier contenant l'image avec l'application). (**AlignImage** permet de positionner l'image sur le bouton.)



On peut aussi puiser une image dans une **ImageList** grâce à la propriété **ImageList** et **ImageIndex**, on peut ainsi changer d'image.

La propriété **BackgroundImage** permet de mettre une image de fond.

Exemple:

```
button1.Text="Ok" affiche 'Ok' dans le bouton.
```

Utiliser les événements:

L'évènement principalement utilisé est **Click()** : quand l'utilisateur clique sur le bouton la procédure

```
Private Sub Button_Click(..)
```

```
End Sub
```

est traitée.

Cette procédure contient le code qui doit être exécuté lorsque l'utilisateur clique sur le bouton.

Le bouton peut être sélectionné grâce à un clic de souris, à la touche ENTRÉE ou à la BARRE D'espacement si le bouton a le focus.

Créer un bouton Ok ou Cancel:

Parfois il faut permettre aux utilisateurs de sélectionner un bouton en appuyant sur la touche ENTRÉE même si le bouton n'a pas le focus.

Exemple: Il y a sur la fenêtre un bouton "Ok" qui doit être enfoncé quand l'utilisateur tape 'Enter' au clavier, c'est le bouton qui 'valide' le questionnaire (et qui le ferme souvent).

Comment faire?

Définissez la propriété `AcceptButton` de la Form en lui donnant le nom du bouton.

Cela permet au formulaire d'avoir le comportement d'une boîte de dialogue.

Création d'un bouton par code:

L'exemple suivant crée un Button nommé `Button1` sur lequel on voit "Ok", on modifie certaines de ses propriétés et l'ajoute à Form.

```
Private Sub InitializeMonButton()  
Dim bouton1 As New Button1()  
bouton1.Text="Ok"  
' Ajouter le bouton à la Form  
Controls.Add(bouton1)  
End Sub
```

Il faut par code créer aussi les évènements liés à ce bouton: dans ce cas il faut déclarer le bouton plutôt avec la syntaxe contenant `WithEvents` et en haut du module.

```
Private WithEvents Button1 As Button
```

Puis écrire la sub évènement.

```
Sub OnClique ( sender As Objet, EvArg As EventArgs)  
Handles Button1  
  
End Sub
```

Ainsi VB sait que pour un évènement sur le `Button1` , il


faut déclencher la Sub OnClique.

(On reviendra sur cela)

Couleur transparent dans les images Des boutons:

On a vu qu'on pouvait mettre une image dans un bouton , il faut pour cela donner à la propriété **Image** le nom du fichier contenant l'image, ceci en mode Design.

Mais l'image est souvent dans un carré et on voudrait ne pas voir le fond (rendre la couleur du fond **transparente**)

Voici l'image  , je voudrais ne pas afficher le 'jaune' afin de voir ce qu'il y a derrière et donner

l'aspect suivant



Dans Visual Basic 6.0, la propriété **MaskColor** était utilisée pour définir une couleur qui devait devenir transparente, permettant ainsi l'affichage d'une image d'arrière plan.

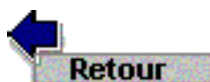
Dans Visual Basic .NET, il n'existe pas d'équivalent direct de la propriété **MaskColor**!! Cependant, on peut ruser et définir la transparence:

Dans le " Code généré par le Concepteur Windows Form " après la définition du bouton ou dans **Form_Load** ajouter:

```
Dim g As New System.Drawing.Bitmap(Button1.Image)
g.MakeTransparent(System.Drawing.Color.Yellow)
Button1.Image = g
```

On récupère le **Bitmap** de l'image du bouton , on indique que le jaune doit être transparent, on remet le **BitMap**.

Bien sur il y a intérêt à choisir une couleur (toujours la même) qui tranche pour les fonds de dessin et ne pas l'utiliser dans le dessin lui même.



Site **LDF** Cours : **VB.net**



3.4	Les TextBox		
	.	Suivant	Sommaire

Les contrôles permettant de saisir du texte sont :

Les TextBox

Les RichTextBox

Les contrôles TextBox.

Contrôle qui contient du texte qui peut être modifié par l'utilisateur du programme.

C'est la propriété `Text` qui contient le texte qui a été tapé par l'utilisateur.

Exemple hyper simple: Comment demander son nom à l'utilisateur ?

Il faut créer un label dont la propriété `Text` contient "Tapez votre nom:", suivi d'un TextBox nommé `txtNom` avec une propriété `Text=""` (Ce qui fait que la TextBox est vide), enfin un bouton nommé `btOk` dont la propriété `Text="Ok"`. Cela donne :

Tapez
votre
nom:

`txtNom.Select()` dans `Form_Load` donne le focus à la TextBox

Une fois que l'utilisateur a tapé son nom , il clique sur le bouton 'Ok' :

Dans `btOk_Click` il y a :

```
Dim Nom As String
```

```
Nom= txtNom.Text
```

La variable Nom contient bien maintenant le nom de l'utilisateur.

Un TextBox correspond à un **mini éditeur de texte**. (Mais sans enrichissement: sans gras, ni italique... du moins pour être exact, l'enrichissement affecte la totalité du texte et pas seulement une partie) La police de caractères affectant la totalité du texte peut simplement être modifiée par la propriété **Font**. La couleur du texte peut être modifiée par **ForeColor**, mais la totalité du texte aura la même couleur.



La propriété **.text** permet aussi de modifier le texte visible dans le contrôle.

```
TextBox1.text= "Bonjour" 'Affiche 'Bonjour' dans le contrôle.
```

Parmi les multiples propriétés de ce contrôle, signalons :

Multiline : autorise ou non l'écriture sur plusieurs lignes.

Scrollbars : fait figurer une barre de défilement horizontale ou verticale (ou les deux)

PasswordChar : crypte le texte entré sous forme d'étoiles.

MaxLength : limite le nombre de caractères qu'il est possible de saisir.

```
TextBox1.MaxLength= 3 'limite la saisie à 3 caractères.
```

```
TextBox1.MaxLength= 0 'ne limite pas la saisie.
```

TextLength donne la longueur du texte

En mode **MultiLine** la collection **Lines** contient dans chacun de ses éléments une des lignes affichées dans le contrôle:

```
TextBox1.Lines(0) contient la première ligne
```

```
TextBox1.Lines(1) la seconde...
```

Les TextBox contiennent une méthode Undo:annulation de la dernière modification.

La propriété **CanUndo** du TextBox doit être à True.

Ensuite pour modifier:

```
If textBox1.CanUndo = True Then
  textBox1.Undo()
  ' Vider le buffer Undo.
  textBox1.ClearUndo()
End If
```

Ajouter au texte:

On peut ajouter du texte au texte déjà présent dans le TextBox

`textBox2.AppendText(MonText)`

C'est équivalent à `textBox2.Text=textBox2.Text+MonText`

Evènements liés aux TextBox:

KeyDown survient quand on appuie sur le touche.

KeyPress quand la touche est enfoncée.

KeyUp quand on relâche la touche.

Ils surviennent dans cet ordre.

KeyPress permet de récupérer la touche tapée dans `e.KeyChar` (mais pas F1, F2..)

KeyDown et KeyUp permettent aussi de voir si MAJ ALT CTRL ont été pressés.

On peut récupérer la touche pressée (dans `e.KeyChar`), mais impossible d'en modifier la valeur (`e.KeyChar` est en lecture seule par exemple)

Comment récupérer la totalité du texte qui est dans le TextBox?

```
T= textBox1.Text
```

Comment mettre les lignes saisies par l'utilisateur dans un tableau ?

```
Dim tempArray() as String
```

```
tempArray = textBox1.Lines
```

 'On utilise la collection **Lines**

Comment récupérer la première ligne ?

```
T= textBox1.Lines(0)
```

Si une partie du texte est **sélectionnée** par l'utilisateur, on peut la récupérer par:

```
T= TextBox1.SelectedText
```

Pour **sélectionner** une portion de texte on utilise:

```
TextBox1.SelectionStart=3
```

 'position de départ

```
TextBox1.SelectionLength=4
```

 'nombre de caractère sélectionné

On peut aussi écrire:

```
TextBox1.Select(3,4)
```

puis

```
TextBox1.SelectedText="toto"
```

 'remplace la sélection par 'toto'

Comment positionner le curseur après le troisième caractère?

En donnant à la propriété SelectionStart la valeur 3

```
TextBox1.SelectionStart=3
```

SelectionLength doit avoir la valeur 0

Comment interdire la frappe de certains caractères dans une TextBox?

Exemple:

Ne permettre de saisir que des chiffres.

Pour cela il faut utiliser l'évènement `KeyPress` du `textBox` qui retourne un objet `e` de type `KeyPressEventArgs`. `e.KeyChar` contient le caractère pressé, mais il est en lecture seule!! on ne peut le modifier. Pour annuler la frappe (dans notre exemple si le caractère n'est pas un chiffre) il faut faire `e.Handled=True`

`IsNumeric` permet de tester si le caractère est numérique.

```
Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
If IsNumeric(e.KeyChar) Then
    e.Handled = False
Else
    e.Handled = True
End If
End Sub
```

Compter combien de fois on a tapé certains caractères?

```
Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
Select Case e.KeyChar
' Compte les backspaces.
Case ControlChars.Back
    Nombrebackspace = Nombrebackspace + 1
' Compte les 'ENTER' .
Case ControlChars.Lf
    Nombrereturn = Nombrereturn + 1
' Compte les ESC .
Case Convert.ToChar(27)
    NombreEsc = NombreEsc + 1
' Compte les autres.
Case Else
    keyPressCount = keyPressCount + 1
End Select
End Sub
```

Petite parenthèse:

Pour comparer les caractères il y a 2 méthodes:

```
if e.KeyChar=Convert.ToChar(27) then
```

ou

```
if AscW(e.Keychar)=27 then
```

Différentes manières de récupérer ce qui a été tapé:

On a vu que `TextBox.text` contient la totalité du texte; si on l'utilise dans l'évènement `TextBox1_TextChanged`, on récupère le nouveau texte dès que l'utilisateur a tapé quelque chose.

TextBox1_KeyPress() TextBox1_KeyUp() permettent de récupérer le caractère qui à été tapé.

Y a t-il un moyen de modifier le caractère tapé? les propriétés de e comme e.KeyChar (dans KeyPress) ou e.KeyCode, e.KeyData e.KeyValue dans les évènements KeyPress et KeyDown sont en lecture seule!!! Une solution est de modifier directement le texte:

Exemple: Si l'utilisateur tape ',' afficher '.' à la place.

```
Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
    Dim pos As Integer
    pos = TextBox1.SelectionStart 'on mémorise la position du curseur
    If e.KeyChar = "," Then
        e.Handled = True 'on ne valide pas le caractère ',' qu n'apparaîtra pas.
        TextBox1.Text = TextBox1.Text.Insert(pos, ".") 'on insère un '.'
        TextBox1.SelectionStart = pos + 1 'on avance le curseur d'un caractère
    End If
End Sub
```



Autre solution?

Le contrôle RichTextBox :

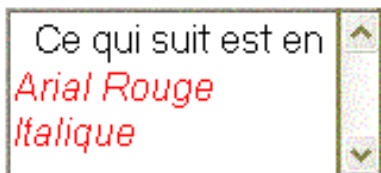
Si vous êtes débutant passer à un rubrique suivante, vous reviendrez plus tard à la gestion du code RTF.

Rich Text veut dire 'Texte enrichi'

Le contrôle `RichTextBox` permet d'afficher, d'entrer et de manipuler du texte mis en forme. Il effectue les mêmes tâches que le contrôle `TextBox`, mais il peut également afficher des polices, des couleurs et des liens, charger du texte et des images incorporées à partir d'un fichier, ainsi que rechercher des caractères spécifiques.



Le contrôle `RichTextBox` a les possibilités d'un traitement de texte comme Word.



Qu'est ce que RTF?

Le format du texte que l'on peut mettre dans une RichTextBox est le format **RTF** (Rich Text Format = Format de Texte Enrichi)

Explication : un texte peu être enregistré en brut (sans enrichissements) en RTF (Utilisable dans la plupart des traitements de texte), au format Word (.doc)...

Pour utiliser les fonctionnalités du RichTextBox il faut utiliser la propriété .Rtf

Quand j'affecte un texte à la propriété **.Text** il est affiché tel quel, sans tenir compte de l'enrichissement.

Quand j'affecte un texte à la propriété **.Rtf** du contrôle pour l'afficher, s'il contient des enrichissement au format RTF , l'enrichissement est affiché :

Les bases du codage RTF

Le texte doit débiter par '{' et se terminer par '}', il peut aussi débiter par "{\rtf1\ansi "

Ensuite les enrichissements s'effectuent par des **balises** qui indiquent le début et la fin de l'attribut, une balise commence par le caractère '\' :

Entre **\b** et **\b0** le texte sera en gras (Bold)

Exemple :

Ajoute le texte "Ce texte est en **gras.**" à un contrôle RichTextBox existant.

```
RichTextBox1.Rtf = "{\rtf1\ansi Ce texte est en \b gras\b0.}"
```

Voici les principaux attributs :

- \b** **\b0** ce qui est entre les 2 balises est en gras
- \i** **\i0** ce qui est entre les 2 balises est en italique
- \par** fin paragraphe (passe à la ligne)
- \f** font **\f1 .. \f0** font numéro 1 entre les 2 balises
- \plain** ramène les caractères par défaut
- \tab** caractère de tabulation
- \fs** taille de caractère **\fs28** = taille 28

Mettre un espace après la balise.

Comment afficher un texte enrichi?

`RichTextBox1.RTF= T` 'T étant le texte enrichi

Mettre un texte en couleurs, utiliser plusieurs polices :

Mettre la **table des couleurs** en début de texte :

```
{ \colortbl \red0\green0\blue0;\red255\green0\blue0;\red0\green255\blue0;}
```

Après `Colortbl` (Color Table) chaque couleur est codée avec les quantités de rouge vert et bleue.

Les couleurs sont repérées par leur ordre: couleur 0 puis 1 puis 2.. et séparées par un ';'.

Dans notre exemple couleur 0=noir; couleur 1=rouge; couleur 2=vert

Pour changer la couleur dans le texte on utilise `\cf` puis le numéro de la couleur :

```
« \cf1 toto \cf0 } » 'toto est affiché en rouge.
```

Pour **modifier les polices de caractère**, le procédé est similaire avec une **Font Table** :

```
{\fonttbl  
{\fo\froman Symbol;}  
{\f1\fswiss Arial;}  
}
```

Pour passer en Arial `\f1 ..\f0`

Exemple complet :

```
"{\rtf1\ansi  
{ \colortbl  
\red0\green0\blue0;  
\red255\green0\blue0;  
\red0\green255\blue0;}  
{\fonttbl  
{\fo\froman Symbol;}
```

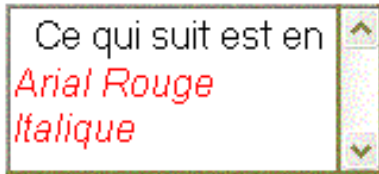
```
{\f1\fswiss Arial;}
```

```
}
```

```
Ce qui suit est en \f1 \cf1 \i Arial Rouge Italique \f0 \cf0 \i0
```

```
} »
```

Cela donne :



Nb : Si vous copier coller l'exemple pour l'essayer, enlever les sauts à la ligne.

Comment modifier l'aspect du texte qui a été sélectionné ?

On n'est plus dans le cas où on affiche d'emblée la totalité du texte, mais dans le cas où l'utilisateur veut modifier son texte qui est déjà dans le contrôle.

Exemple: L'utilisateur sélectionne une portion du texte dans le contrôle puis clique sur un bouton nommé 'Rouge' pour mettre la sélection en rouge.

Dans `BoutonRouge_Click()` écrire :

```
RichTextBox1.SelectionColor = System.Drawing.Color.Red
```

De même pour modifier la police, la hauteur de la police, l'aspect gras ou non :

```
RichTextBox1.SelectionFont = New Font("Tahoma", 12, FontStyle.Bold)
```

Enfin le texte peut être enregistré dans un fichier :

```
richTextBox1.SaveFile(fileName, RichTextBoxStreamType.RichText)
```

Si on remplace `.RichText` par `.PlainText` c'est le texte brut et non le texte enrichi qui est enregistré

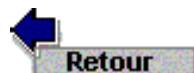
Pour lire un fichier il faut employer `.LoadFile` avec la même syntaxe.

Comment faire une recherche dans le texte?

La fonction `Find` permet de rechercher une chaîne de caractères dans le texte :

```
richTextBox1.Find(searchText, searchStart, searchEnd,  
RichTextBoxFinds.MatchCase)
```

La méthode retourne l'emplacement d'index du premier caractère du texte recherché et met en surbrillance ce dernier ; sinon, elle retourne la valeur -1.



Site **LDF** Cours **VB.net**



3.5	Les Labels	Suivant 	 Sommaire
------------	-------------------	---	--

Il y a 2 sortes de Label :

Les Label

Les LinkLabel

Les labels.

On en a déjà utilisé pour **afficher du texte non modifiable par l'utilisateur.**

Les contrôles Label sont généralement utilisés pour **fournir un texte descriptif à un contrôle.** Vous pouvez par exemple utiliser un contrôle Label pour ajouter un texte descriptif à un contrôle TextBox. Ceci a pour but d'informer l'utilisateur du type de donnée attendu dans le contrôle.

Exemple hyper simple: Comment indiquer à l'utilisateur qu'il faut saisir son nom?

Donner votre nom:

La légende qui s'affiche dans l'étiquette est contenue dans la propriété **Text** du label.

Pour modifier le texte du label1 par du code:

```
Label1.Text="Donner votre Prénom"
```

La propriété **Alignement** vous permet de définir l'alignement du texte dans l'étiquette (centré, à droite, à gauche), **BorderStyle** permet de mettre une bordure (un tour) ou non..

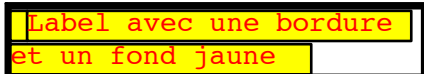
Label1.Text="Label avec une bordure et un fond jaune"

Label1.BorderStyle=BorderStyle.FixedSingle

Label1.ForeColor=Color.Red

Label2.BackColor=Color.Yellow

donne:



Remarque: la plupart du temps les labels sont modifiés en mode design, directement dans la fenêtre des propriétés.

Il est également possible d'y afficher une image avec la propriété **.Image**

Remarque:

La mise à jour de l'affichage du Label (comme les autres contrôles d'ailleurs) est effectuée en fin de Sub:

Si on écrit:

```
Dim i As Integer
For i = 0 To 100
    Label1.Text = i.ToString
Next i
```

La variable i prend les valeurs 1 à 100, mais à l'affichage rien ne se passe pendant la boucle, VB affiche uniquement 100 à la fin; si on désire voir les chiffres défiler avec affichage de 0 puis 1 puis 2... il faut rafraîchir l'affichage à chaque boucle avec la méthode **Refresh()**:

```
Dim i As Integer
For i = 0 To 100
    Label1.Text = i.ToString: Label1.Refresh()
Next i
```

Les LinkLabel

Permettent de créer un **lien** sur un label

Text Indique le texte qui apparaît.

LinkArea définit la zone de texte qui agira comme un lien; dans la fenêtre de propriété taper 11 ;4 (on verra que c'est plus simple que de le faire par code)

Les 4 caractères à partir du 11ème seront le lien, ils seront soulignés. Ne pas oublier comme toujours que le premier caractère est le caractère 0.

L'événement **LinkClicked** est déclenché quand l'utilisateur clique sur le lien. Dans cette procédure on peut permettre le saut vers un site Internet ou toute autre action.

Exemple :

```
LinkLabel1.text= "Visitez le site LDF"
```

```
LinkLabel1.LinkArea = New System.Windows.Forms.LinkArea(11, 4)
```

'Pourquoi faire simple !!

Cela affiche :

Visitez le [site](#) LDF

Si l'utilisateur clique sur le mot site, la procédure suivante est déclenchée :

```
Private Sub LinkLabel1.LinkClicked...
```

```
End Sub
```

Il est possible de modifier la couleur du lien pour indiquer qu'il a été utilisé:

Si **VisitedLinkColor** contient une couleur `e.visited=True` modifie la couleur.

(e est l'élément qui a envoyé l'évènement, j'en modifie la propriété Visited.)

On peut y inclure une action quelconque, en particulier un saut vers un site Web:

```
System.Diagnostics.Process.Start("http://plasserre.developpez.com/")
```

'correspond au code qui ouvre un browser Internet (Internet Explorer ou Netscape) et qui charge la page dont l'adresse est indiquée.

La collection Links permet d'afficher plusieurs liens dans un même texte, mais cela devient vite très compliqué.



Site **LDF** Cours **VB.net**



3.6	<h2 style="color: purple;">Les Cases à cocher</h2>		
-----	--	--	--

Il y a 2 sortes de case à cocher :

Les **CheckBox**.

Les **RadioButton**.

- Les " cases à cocher " (**CheckBox**) : Elles sont carrées, et indépendantes les unes des autres, si l'utilisateur coche une case , cela n'a pas d'influence sur les autres cases du formulaire, qu'elles soient regroupées dans un cadre pour faire plus joli ou non.
- Les " boutons radio " (**RadioButton**) : Ils sont ronds et font toujours partie d'un groupe (Ils sont dans une fenêtre ou dessinés dans un objet **GroupBox**). Ce groupe est indispensable, car au sein d'un groupe de **RadioButton**, un seul bouton peut être coché à la fois : si l'utilisateur en coche un, les autres se décochent.

CheckBox

RadioButton

Il faut **regrouper les radios boutons** dans des 'GroupBox' par exemple pour rendre les groupes indépendants:

<p>Format</p> <p><input type="radio"/> RTF</p> <p><input type="radio"/> TEXTE BRUT</p>	<p>Sauvegarde auto</p> <p>OUI <input type="radio"/></p> <p>NON <input type="radio"/></p>
--	--

Ici si je clique sur le bouton 'OUI' à droite, cela décoche 'NON' mais n'a pas d'influence sur le cadre Format

La propriété **Text**, bien sur, permet d'afficher le libellé à coté du bouton, on peut aussi mettre une image avec la propriété **Image**. **CheckAlign** permet de mettre la case à cocher à droite ou à gauche du texte, **TextAlign** permet d'aligner le texte.

Exemple pour le bouton en haut à droite.

```
RadioButton3.Text="OUI"
```

```
RadioButton3.TextAlign= MiddleCenter 'Middle=hauteur,  
center = horizontale
```

```
RadioButton3.CheckAlign=MiddleRight
```

La propriété la plus intéressante de ces cases est celle qui nous permet de savoir si elle est cochée ou non. Cette propriété s'appelle **Checked**. Sa valeur change de **False** à **True** si la case est cochée.

```
RadioButton.Checked=True 'Coche le bouton
```

`If RadioButton.Checked=True Then` ' Teste si le bouton est coché.

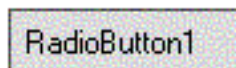
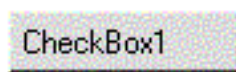
`End If`

La procédure **RadioButton.CheckedChange()** permet d'intercepter le **changement d'état** d'un bouton.

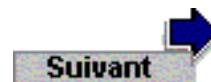
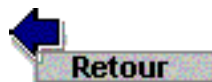
Pour le CheckButton **TreeState** permet de définir 3 états au lieu de 2 (coché, indéterminé=grisé, non coché)

CheckedState indique un des 3 états (alors que **Checked** n'en indique que deux.)

Appearance peut aussi donner une **apparence de bouton** à la case à cocher. Il est enfoncé ou pas en fonction de la valeur de **Checked**.



Ici les 2 boutons ont une `Appearance=Button` , celui du haut n'est pas coché, l'autre est coché (enfoncé).





3.7	Les Contrôles 'liste'	Suivant →	← Sommaire
------------	------------------------------	------------------	-------------------

Il y a 4 sortes de listes:

Les **ListBox**.

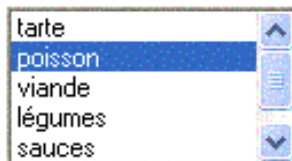
Les **CheckedListBox**

Les **Combos**

Les **ListView**.

Les 'ListBox'

Le contrôle **ListBox** affiche une liste d'éléments (d'objets) dans laquelle l'utilisateur peut faire un ou plusieurs choix.



La liste contient "tarte", "poisson", "viande", "légumes", "sauces".

Ici l'élément "poisson" est sélectionné, il est en bleu.

La listBox contient une collection d'"Item":

Elle n'a pas de nombre initialement défini d'élément.

Si j'ajoute un élément à la ListBox, cela ajoute un élément à la collection Items

Items est une *collection* contenant tous les éléments (les objets) chargés dans la liste.

`ListBox1.Items` est la collection du contrôle `ListBox1`

La propriété **Items.Count** indique le nombre d'éléments contenus dans la liste . Attention **le premier élément est toujours l'élément 0** , aussi le nombre d'éléments est égal au numéro de l'élément le plus haut plus un.

Pour ajouter ou supprimer des éléments dans un contrôle ListBox:

Utilisez la méthode **Items.Add**, **Items.Insert**, **Items.Clear** ou **Items.Remove**. En mode conception, vous pouvez également utiliser la propriété **Items**.

Exemples :

Vider la ListBox:

```
ListBox1.Items.Clear()
```

Ajouter l'élément "poisson"

```
ListBox1.Items.Add("poisson")
```

Ajouter '4'

```
ListBox1.Items.Add(4.ToString)
```

ou

```
ListBox1.Items.Add(4) 'accepté car les items sont des objets.'
```

Insérer 'lulu' en 4^{ème} position

```
ListBox1.Items.Insert(4, "lulu")
```

Les `listBox` acceptent des objets, elles affichent généralement ce qu'il y a dans la propriété 'Text' de l'objet.

Charger dans une `ListBox1` les nombres de 1 à 100 :

```
For i = 1 To 100
```

```
    ListBox1.Items.Add(i.ToString)
```

[Next](#) [i](#)

Comment enlever des éléments?

' Enlever l'élément d'index 5:

```
ListBox1.Items.RemoveAt(5)
```

' Enlever l'élément sélectionné:

```
ListBox1.Items.Remove(ListBox1.SelectedItem)
```

' Enlever l'élément "Tokyo":

```
ListBox1.Items.Remove("Tokyo")
```

Comment lire l'élément 3?

```
T=ListBox1.Items(3).ToString
```

Comment rechercher l'élément qui contient une chaîne de caractères?

`List1.FindString("pa")` retourne le numéro du premier élément commençant par 'pa'.

`x=List1.FindString("pa",12)` retourne le numéro de l'élément commençant par 'pa' en cherchant à partir du 12 ème élément.

`x=List1.FindStringExact("papier")` 'permet de rechercher l'élément correspondant exactement à la chaîne

Comment sélectionner un élément par code?

```
ListBox1.SetSelected(x, True)
```

L'utilisateur double-clique sur un des éléments, comment récupérer son numéro ?

Grâce à `SelectedIndex`.

```
Private Sub ListBox_DoubleClick.
```

```
    N=ListBox1.SelectedIndex
```

```
End If
```

'N contient le numéro de l'élément sélectionné. Attention comme d'habitude, si je sélectionne « 3 » c'est en faite l'élément numéro 2.

`SelectedIndex` retourne donc un entier correspondant à l'élément sélectionné dans la zone de liste. Si aucun élément n'est sélectionné, la valeur de la propriété `SelectedIndex` est égale à -1.

La propriété `SelectedItem` retourne l'élément sélectionné ("poisson" dans l'exemple si dessus).

Et la multi sélection, quelles éléments ont été sélectionnés?

La propriété **SelectionMode** indique le nombre d'éléments pouvant être sélectionnés en même temps.

Lorsque plusieurs éléments sont sélectionnés, la valeur de la propriété **SelectedIndex** correspond au rang du premier élément sélectionné dans la liste. Les collections **SelectedItems** et **SelectedIndices** contiennent les éléments et les numéros d'index sélectionnés.



Si la propriété Sorted est à True, la liste est triée automatiquement.

On peut 'charger' une **ListBox** automatiquement avec un tableau en utilisant **Datasource**:

```
Dim LaList() As String = {"one", "two", "three"}  
ListBox1.DataSource = LaList
```

On peut aussi utiliser **AddRange**:

```
Dim Ite(9) As System.Object  
Dim i As Integer  
For i = 0 To 9  
Ite(i) = "Item" & i  
Next i  
  
ListBox1.Items.AddRange(Ite)
```

Comment connaître l'**index de l'élément que l'on vient d'ajouter** ? (et le sélectionner)

```
Dim x As Integer  
x = List1.Items.Add("Hello")  
List1.SelectedIndex = x
```

On utilise la valeur retournée (x dans notre exemple) par la méthode **Add**.

(NewIndex n'existe plus en VB.NET)

Comment affecter à chaque élément de la liste un numéro, une clé?

Exemple : je charge dans une **ListBox** la liste des utilisateurs mais quand on clique sur la liste, je veux récupérer le numéro de l'utilisateur.

Comment donc, à chaque élément de la listbox, donner un numéro (différent de l'index).

En VB6 on utilisait une propriété (**ListBox.ItemData()**) pour lier à

chaque élément de la listBox un nombre (une clé); cela n'existe plus en VB.Net!!

Il faut utiliser les fonctions de compatibilité:

```
VB6.SetItemData(ListBox1, 0, 123)
```

 'pour lier à l'élément 0 la valeur 123.

C'est pas simple!!

Une alternative, pas très élégante:

Ajouter l'élément "toto"+chr\$(9)+chr\$(9)+ clé (clé n'est pas visible car les caractères tabulation l'ont affichée hors de la listBox)

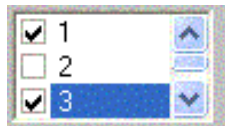
Quand l'utilisateur clique sur la ligne, on récupère la partie droite donc la clé.

Quand on charge une ListBox directement avec une base de données, il y a une solution pour gérer une clé.

Lorsque la propriété **MultiColumn** a la valeur **true**, la liste s'affiche avec une barre de défilement horizontale. Lorsque la propriété **ScrollAlwaysVisible** a la valeur **true**, la barre de défilement s'affiche, quel que soit le nombre d'éléments.

CheckedListBox

C'est une Listbox mais avec une **case à cocher** sur chaque ligne.



Attention : **SelectedItems** et **SelectedIndices** ne déterminent pas les éléments qui sont cochés, mais ceux qui sont en surbrillance.

La collection **CheckedItems** vous donne par contre les éléments cochés. La méthode **GetItemChecked** (avec comme argument le numéro d'index) détermine si l'élément est coché.

Exemple :

Pour déterminer les éléments cochés dans un contrôle CheckedListBox :

Tester chaque élément de la collection **CheckedItems**, en commençant par 0. Notez que cette méthode fournit le numéro que porte l'élément dans la liste des éléments cochés, et non dans la liste globale. Par conséquent, si le premier élément de la liste n'est pas coché alors que le deuxième l'est, le code ci-dessous affiche une chaîne du type « Item coché 1 = Dans la liste : 2 ».

```
If CheckedListBox1.CheckedItems.Count <> 0 Then
```

```
    'S'il y a des éléments cochés une boucle balaye les
```

éléments cochés

' (collection CheckedItems) et affiche le numéro de l'élément DANS LA LISTE toutes lignes.

```
Dim x As Integer
Dim s As String = ""
For x = 0 To CheckedListBox1.CheckedItems.Count - 1
    s = s & "Item coché " & (x+1).ToString & " = " &
« Dans la liste : » &
CheckedListBox1.CheckedItems(x).ToString &
ControlChars.CrLf
Next x
MessageBox.Show(s)
End If
```

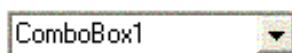
On rappelle comme toujours que quand on parle du 3eme élément cela correspond à l'index 2.

Les ComboBox

Les listes Combo (Liste combiné) possèdent deux caractéristiques essentielles par rapport aux ListBox.

Elles sont **modifiables** : c'est-à-dire que l'utilisateur a la possibilité d'entrer un élément qui ne figure pas au départ dans la liste. Cette caractéristique concerne donc les données proprement dites ; cela se traduit par la présence d'une zone de texte en haut de la liste.

Elles peuvent être **déroulantes** ou déjà déroulée: c'est-à-dire qu'on ne voit qu'un seul élément de la liste à la fois, et qu'il faut cliquer sur la flèche du côté pour " déplier " la liste, ou bien que la liste est déjà visible. C'est la propriété **DropDownList** qui gère cela.



La combobox du bas a sa `DropDownList=Simple`

L'utilisateur peut donc cliquer dans la liste (ce qui met le texte cliqué dans la zone texte), ou taper un nouveau texte.

Items.Add (méthode) ajoute un élément à une liste.

Items.Clear (méthode) efface tous les éléments d'une

liste

Items.Count (propriété) renvoie le nombre d'éléments d'une liste

Multiselect (propriété) permet la sélection multiple

Item.Remove (méthode) supprime un élément de la liste

Sorted (propriété) trie les éléments d'une liste

Comment récupérer la zone texte quand elle change ?

Elle est dans la propriété **Text**.

On utilise l'évènement **TextChanged** qui se déclenche quand le texte est modifié.

```
Private Sub ComboBox1_TextChanged(ByVal sender As Object,  
ByVal e As System.EventArgs) Handles  
ComboBox1.TextChanged
```

```
Label1.Text = ComboBox1.Text
```

```
End Sub
```

Le Contrôle ListView :

De plus en plus puissant, le contrôle **ListView** permet d'afficher des listes multi colonnes, ou des listes avec icône ou case à cocher.

En mode conception :

La propriété **View** permet de déterminer l'aspect général du contrôle, elle peut prendre les valeurs :

Details permet une liste avec sous éléments et titre de colonnes.

Liste utilise un ascenseur horizontal.

LargeIcon

SmallIcon

Par programmation cela donne :

```
ListView1.View= View.Details
```

Utilisons le mode détails (Appelé mode Rapport)

Nombre	Carré	Cube
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216

Comment remplir les en-têtes de colonnes?

En mode conception il y a une ligne **Columns**, le fait de cliquer sur le bouton d'expansion (...) ouvre une fenêtre, cliquer sur 'Ajouter' permet d'ajouter une colonne ; la propriété **Text** permet de donner un libellé qui apparaîtra en haut de la colonne. On peut ainsi nommer les 3 colonnes (« Nombre », « Carré », « Cube » dans notre exemple)

Par programmation cela donne :

```
ListView1.Columns.Add (« Nombre », 60,
HorizontalAlignment.Left)
```

..

Pour remplir le tableau, on pourrait, sur la ligne Items de la fenêtre des propriétés, cliquer sur ... et rentrer les valeurs 'à la main'.



En pratique on crée les colonnes, le nom des colonnes en mode conception, on remplit le tableau par programmation :

Exemple : Faire un tableau de 3 colonnes, mettre les nombres de 1 à 100 dans la première, leur carré dans la seconde, leur cube dans la troisième.

Pour chaque ligne je crée un objet **ListViewItem**, sa propriété **Text** contient le texte de la première colonne ; j'ajoute à cet objet des **SubItems** qui correspondent aux colonnes suivantes. Enfin j'ajoute le **ListViewItem** au contrôle **ListView**.

```
Dim i As Integer
For i = 1 To 100
    Dim LVI As New ListViewItem
    LVI.Text = i.ToString
    LVI.SubItems.Add((i * i).ToString)
    LVI.SubItems.Add((i * i * i).ToString)
```

```
ListBox1.Items.Add(LVI)
Next i
```

Comment intercepter le numéro de la ligne qui a été cliquée par l'utilisateur (et l'afficher)?

```
Private Sub ListBox1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles ListBox1.Click
```

```
    Label1.Text =
    ListBox1.SelectedIndices(0).ToString
```

```
End Sub
```

Si la propriété **MultiSelect** est à **False** il y a, bien sur, une seule ligne sélectionnée, sinon les lignes sélectionnées sont dans la collection **SelectedIndices()**.

Si on voulait récupérer le texte de la ligne sélectionnée, il aurait fallu utiliser :

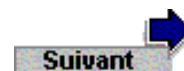
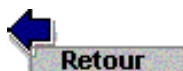
```
ListBox1.SelectedItems(0)
```

Si la propriété **GridLine** est à **True**, des lignes matérialisant les cases apparaissent.

Si la propriété **CheckBox** est à **True** , des cases à cocher apparaissent.

Attention : si la somme des colonnes est plus large que le contrôle, un ascenseur horizontal apparaît !!

Pour ne pas voir cet ascenseur, ruser sur la largeur des colonnes (c'est le 2eme paramètre de la méthode **.Columns.Add**)



3.8	Les fenêtres toutes faites.	Suivant 	 Sommaire
-----	--------------------------------------	---	--

Il existe :

Les MessageBox.

Les InputBox

Ces fenêtres toutes faites facilitent le travail :

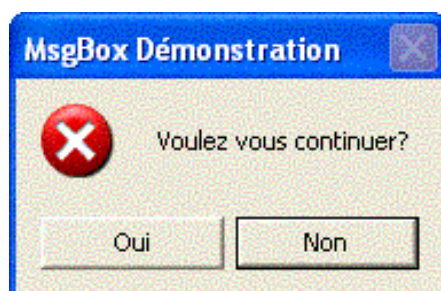
MessageBox :

Ouvre une fenêtre qui présente un message.

C'est une fonction qui affiche un message dans une boîte de dialogue, attend que l'utilisateur clique sur un bouton (Ok ou Oui-Non..), puis retourne une information qui indique le bouton cliqué par l'utilisateur.

Reponse= **MessageBox.show**(TexteAAfficher, Titre, TypeBouton et Icône, BoutonParDéfaut)

Exemple:



Paramètres

TexteAAfficher

Obligatoire. Expression **String** affichée comme message de la boîte de dialogue (longueur maximale 1 024 caractères). N'oubliez pas d'insérer un retour chariot si le texte est long, cela crée 2 lignes.

Titre

Expression **String** affichée dans la barre de titre de la boîte de dialogue. Si l'argument *Titre* est omis, le nom de l'application est placé dans la barre de titre.

TypeBouton et Icons

Expression numérique qui représente la somme des valeurs spécifiant

-le nombre et le type de boutons à afficher :

`MessageBoxButtons.OKOnly` Un seul bouton 'Ok'

`MessageBoxButtons.YesNo` Deux boutons 'Oui'
'Non'

`MessageBoxButtons.OkCancel` 'Ok' et 'Annuler'

`MessageBoxButtons.AbortRetryIgnore` 'Annule'
'Recommence' 'Ignore'

..

-le style d'icône à utiliser :

`MessageBox.Icons.Critical`

`MessageBox.Icons.Exclamation`

`MessageBox.Icons.Question`

`MessageBox.Icons.Information`

L'identité du bouton par défaut

`MessageBox.DefaultButtons.DefaultButton1`

`MessageBox.DefaultButtons.DefaultButton2`

Retour de la fonction :

Retourne une constante qui indique quel bouton à été pressé.

`DialogResult.Yes`

`DialogResult.No`

`DialogResult.Cancel`

`DialogResult.Retry`

`DialogResult.Ok`

L'ancienne syntaxe VB avec MsgBox est conservée :

`Reponse= MsgBox(TexteAAfficher, TypeBouton, Titre)`

Dans ce cas il faut utiliser `MsgBoxStyle` `MggBoxIcons` et `MsgBoxResult` pour le retour. De plus les arguments ne sont pas dans le même ordre!!

Il est conseillé d'utiliser `MessageBox.Show` (qui est VB.NET) plutôt que `MsgBox` qui est de la compatibilité avec VB

Exemple :

`Reponse=MessageBox.Show(« Bonjour »)`

'Affiche le message 'Bonjour' avec un simple bouton 'Ok'

Cela sert à fournir un message à l'utilisateur sans attendre de choix de sa part.

Autre exemple en ancienne syntaxe :

`R=MsgBox("Continuer"& chr$(13)& "l'application?",
MsgBoxStyle.YesNo, "Attention"`

Affiche une `MessageBox` avec dans la barre de titre « Attention »

'Affiche dans la boite :

« Continuer

l'application » (sur 2 lignes)

'La boite a 2 boutons : 'Oui' 'Non'

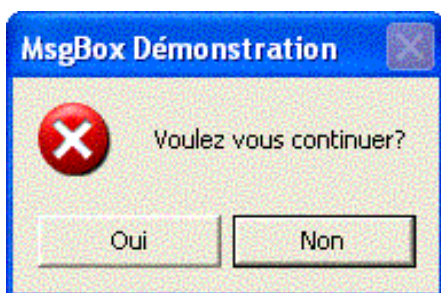
Exemple complet :

```
Dim msg As String
Dim title As String
Dim style As MsgBoxStyle
Dim response As MsgBoxResult
msg = "Voulez vous continuer?" ' Définition
                                du message à afficher.
style = MsgBoxStyle.DefaultButton2 Or _
        MsgBoxStyle.Critical Or MsgBoxStyle.YesNo ' On affiche
Oui Non
title = "MsgBox Démonstration" ' Définition
                                du titre.
' Affiche la boite MsgBox.

response = MsgBox(msg, style, title)

If response = MsgBoxResult.Yes Then ' L'utilisateur a
choisi Oui.
    ' code si l'utilisateur à cliquer sur Oui
Else
    ' code si l'utilisateur à cliquer sur No.
End If
```

Voila ce que cela donne:



'On remarque que dans l'exemple, on crée des variables dans lesquelles on met le texte ou les constantes adéquates, avant d'appeler la fonction MsgBox.

InputBox

c'st une fonction qui permet d'ouvrir une fenêtre qui pose une question :

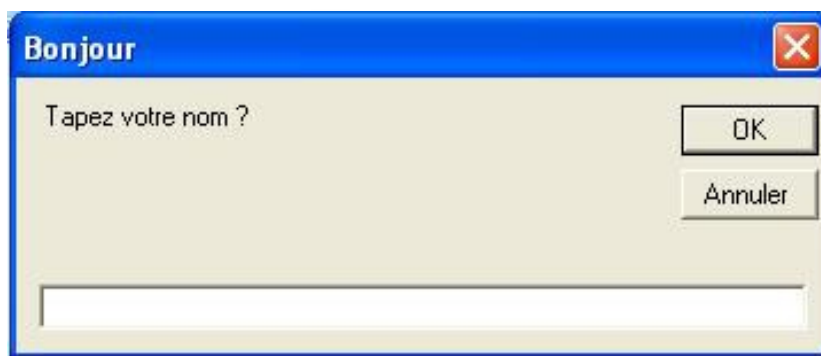
Elle retourne la réponse tapée par l'utilisateur:

Le retour est effectué dans une variable String.

```
Dim Nom As String
```

```
Nom = InputBox("Bonjour","Tapez votre nom ?")
```

Cela donne :



On pourrait rajouter un 3eme argument=la réponse par défaut.

Si l'utilisateur clique sur le bouton annuler, une chaîne vide est retournée.

OpenFileDialog

Comment afficher une boîte de dialogue permettant de sélectionner un fichier (ou des fichiers) à ouvrir par exemple ?

Dans la boîte à Outils, cliquez sur OpenFileDialog puis cliquez sur la fenêtre en cours : un contrôle OpenFileDialog1 apparaît sous le fenêtre.

Ouvre une boîte de dialogue permettant de choisir un nom et un chemin de fichier, au programmeur d'écrire le code lisant les fichiers.

Dans le code à l'endroit ou doit s'ouvrir la fenêtre, tapez :

```
OpenFileDialog1.ShowDialog()
```

C'est suffisant pour créer une fenêtre montrant l'arborescence des fichiers et répertoires et pour que l'utilisateur choisisse un fichier, mais le plus souvent on a besoin que la boîte de dialogue propose un type de fichier et un répertoire précis.

Par exemple je veux ouvrir un fichier .TXT dans le répertoire c:\MesTextes

Il faut dans ce cas, **AVANT** le ShowDialog renseigner certaines propriétés du contrôle OpenFileDialog :

```
With OpenFileDialog1
```

```
.Filter="Fichiers txt|*.txt" ' on travaille uniquement sur les .txt
```

```
par | 's'il y a plusieurs filtre les séparer
```

```
.Multiselect=False 'sélectionner 1 seul fichier
```

```
.CheckFileExists=True 'Message si nom de fichier qui n'existe pas.
```

```
qui existe 'Permet d'ouvrir uniquement un fichier
```

```
End With
```

Comment afficher la boîte et vérifier si l'utilisateur a cliqué sur ouvrir?

```
If OpenFileDialog1.ShowDialog= DialogResult.Ok Then
```

```
end if
```

Maintenant, `OpenFileDialog1.FileName` contient le nom du fichier sélectionné (avec extension et chemin)

```
Path.GetFileName(OpenFileDialog1.FileName) donne le nom du fichier sans chemin.
```

SaveFileDialog

Boîte de dialogue fonctionnant de la même manière que OpenFileDialog

mais avec quelques propriétés spécifiques.

Ouvre une boîte de dialogue permettant à l'utilisateur de choisir un nom et un chemin de fichier, au programmeur d'écrire le code enregistrant les fichiers.

<code>SaveFileDialog1.CreatePrompt= True</code>	' Message de confirmation si 'création d'un nouveau fichier
<code>SaveFileDialog1.OverwritePrompt=True</code>	'Message si le fichier existe déjà 'évite l'effacement d'ancienne données
<code>SaveFileDialog1.DefaultExt="txt"</code>	'extension par défaut

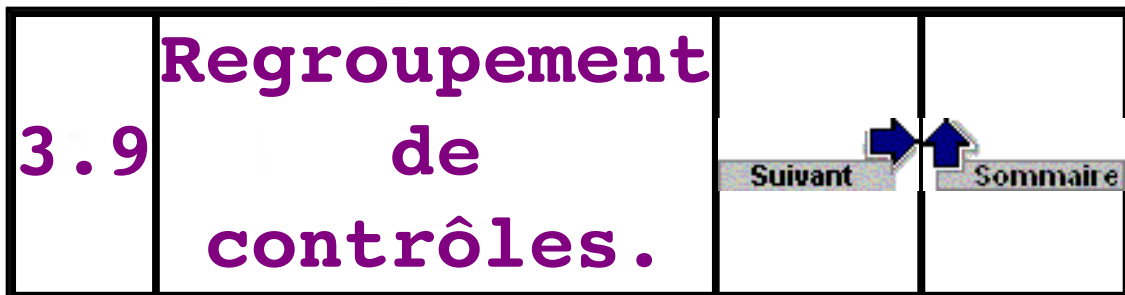
On récupère aussi dans `.FileName` le nom du fichier si la propriété `.ShowDialog` à retourné `DialogResult.Ok`.

Il existe aussi:

LoadDialog

PrintDialog





On peut regrouper des contrôles dans :

Les **GroupBox**.

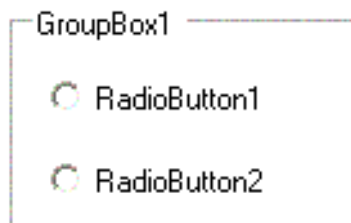
Les **Panels**.

Les **PictureBox**.

Les **TabControl**.

GroupBox et Panel

Il est possible de regrouper des contrôles dans un **container**, on peut par exemple regrouper plusieurs **RadioButton**. Le container peut être un **GroupBox** ou un **Panel**.



GroupBox
BorderStyle=Single

Panel avec AutoScroll =True et

Pour l'utilisateur, le fait que toutes les options soient regroupées dans un panneau est un indice visuel logique (Tous les RadioButton permettrons un choix dans une même catégorie de données). Au moment de la conception, tous les contrôles peuvent être déplacés facilement ; si vous déplacez le contrôle GroupBox ou Panel, tous les contrôles qu'il contient sont également déplacés. Les contrôles regroupés dans un panneau ou un GroupBox sont accessibles au moyen de la propriété Controls du panneau.

Le contrôle **Panel** est similaire au contrôle **GroupBox** ; mais, seul le contrôle Panel peut disposer de barres de défilement et seul le contrôle **GroupBox** peut afficher une légende.

La légende de la **GroupBox** est définie par la propriété **Text**.

Pour faire apparaître les barres de défilement dans le **Pannel** mettre `AutoScroll =True` et `AutoScrollMinSize =100;100`

Dans un Panel, pour afficher des barres de défilement, attribuez à la propriété **AutoScroll** la valeur `true..` La propriété **BorderStyle** détermine si la zone est entourée d'une bordure invisible (**None**), d'une simple ligne (**FixedSingle**) ou d'une ligne ombrée (**Fixed3D**).

Comment créer un contrôle Panel ?

Faites glisser un contrôle **Panel** de l'onglet **Windows Forms** de la boîte à outils jusqu'à un formulaire.

Ajoutez des contrôles au panneau en les déposant dans le panneau.

Si vous voulez mettre dans le panneau des contrôles existants, sélectionnez-les tous, coupez-les dans le Presse-papiers, sélectionnez le contrôle **Panel** et collez-les.

PictureBox

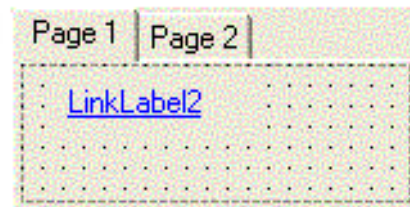
Le contrôle **PictureBox** peut afficher une image mais peu aussi servir de conteneur à d'autres contrôles.

Retenons la **notion de conteneur** qui est le contrôle **parent**.

TabControl

Ce contrôle permet de créer des onglets comme dans un classeur, onglets entièrement gérés par VB. Chaque page peut contenir d'autres contrôles.

En mode conception, en passant par la propriété **TabPage**, on ajoute des onglets dont la propriété **Text** contient le texte à afficher en haut (Ici: Page 1..). il suffit ensuite de cliquer sur chaque onglet et d'y ajouter les contrôles.



En mode run les onglets fonctionnent automatiquement: cliquez sur Page 2 affiche la page correspondante (et déclenche l'évènement **Click** de cet objet **TabPage**)..

La propriété **Alignment** permet de mettre les onglets en haut, en bas, à droite, à gauche.

Evènement commun.

Exemple: 3 cases à cocher permettent de colorer un label en vert rouge ou bleu.

Comment gérer les évènements?

On peut écrire 3 routines complètes pour chaque case à cocher.

Il est aussi toujours possible dans chacune des 3 procédures CouleurX.checkedChanged de vérifier si la case est cochée et de modifier la couleur.

C'est plus élégant d'avoir une procédure unique qui, en fonction de la case à cocher qui a déclenché l'évènement, change la couleur.

On désire donc parfois que l'évènement de plusieurs **contrôles différents soit dirigé sur une seule et même procédure.**

Mais, la notion de groupe de contrôle comme en VB6 n'existe plus!!!

Par contre par l'intermédiaire du Handles, il est possible d'associer plusieurs évènements à une seule procédure:

Private Sub CouleurCheckedChanges (ByVal sender As System.Objet, ByVal e As System.EventArgs)

Handles CouleurVert.CheckedChanged,
CouleurRouge.CheckedChanged, CouleurBleu.CheckedChanged



End Sub

Cette procédure est activée quand les case à cocher
CouleurVert CouleurBleu, CouleurRouge changent d'état.

A noter que **sender** est le contrôle ayant déclenché
l'évènement et **e** l'évènement correspondant.

Pour modifier la couleur il faut ajouter dans la procédure:

```
Select Case sender.Name  
    Case "CouleurRouge"  
        Lbl.BackColor= ..Rouge  
        .....
```

Je ne suis pas certain que cela fonctionne, il faut plutôt
mettre:

```
Select Case sender  
    Case CouleurRouge
```

Enfin la ligne suivante marche !

```
If sender Is CouleurRouge Then...
```



Retour





Index



Sommaire



Suivant

3.10	Positionnons les contrôles.	Suivant 	 Sommaire
-------------	--	--	---

On peut

Dimensionner les contrôles :

Les positionner.

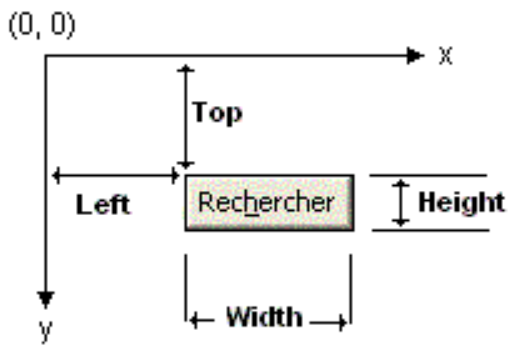
Tous les contrôles héritent donc tous de la classe Windows Forms.

Les contrôles ont tous des propriétés communes.

Unité de mesure :

L'unité de mesure est le 'Pixel' (Picture Élément).(plus de twips)

Les coordonnées d'un contrôle se définissent à partir du coin supérieur gauche du conteneur.



Pour la taille :

On peut utiliser:

Left, Top coordonnées du coin supérieur gauche et **Bottom, Right** inférieur droit.

Location : coordonnées X,Y du coin supérieur droit du contrôle en pixels.

Height, Width pour la hauteur et la largeur du contrôle en pixels.

Size : hauteur, largeur peuvent aussi être utilisées.

Exemple :

```
Button.left=188
```

```
Button.Top =300
```

Ou

```
Button.Location= New System.Drawing.Point(188,300)
```

System.Drawing.Point() positionne un point dans l'espace.



En mode conception il est bien plus simple de dimensionner les contrôles à la main dans la

fenêtre Design.

Pour le redimensionnement de fenêtre :

Pour que l'utilisateur puisse redimensionner la fenêtre(en cliquant sur les bords) il faut que la propriété **FormBorderStyle** de la fenêtre = **Sizable**.

Mais si l'utilisateur modifie la taille de la fenêtre qui contient les contrôles, la taille des contrôles ne suit pas.

(Avant cette version VB.net, il fallait dans l'événement `Form_Resize`, déclenché par la modification des dimensions de la fenêtre, écrire du code modifiant les dimensions et positions des contrôles afin qu'ils s'adaptent à la nouvelle fenêtre)

En VB.Net c'est plus simple grâce à :

Anchor :

Permet d'**ancrer** les bords.

Un bord ancré reste à égale distance du bord du conteneur quand le conteneur (la fenêtre) est redimensionné.

En mode conception il suffit de cliquer sur '. . . .' en face de Anchor pour voir s'ouvrir une fenêtre, cliquer sur les bords que vous voulez ancrer.

Par défaut les bords Top (haut) et left(gauche) sont ancrés.

Expliquons !!

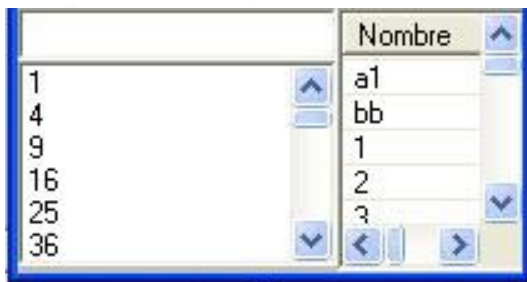
Left est ancré, si je déplace le bord droit de la fenêtre, le contrôle n'est pas déplacé car la distance bord gauche de la fenêtre et bord gauche du contrôle est fixe. Par contre si je déplace le

bord gauche de la fenêtre, le contrôle suit.

Exemple :

Prenons 2 contrôles dans une fenêtre, celui de gauche a `Anchor =left` et celui de droite à `left` et `right`.

Si je déplace le bord droit (ou le gauche d'ailleurs) : le contrôle droit est redimensionné, les 2 contrôles restent cote à cote.



Dock

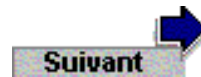
Amarre aux bords. Il y a même possibilité d'amarrer aux 4 bords (`Fill`) pour remplir le conteneur, et de modifier la propriété `DockPadding` afin de s'éloigner légèrement des bords pour faire joli.

Splitter

Le contrôle **Splitter** sert à redimensionner des contrôles **au moment de l'exécution par l'utilisateur**.

Le contrôle **Splitter** est utilisé dans les applications dont les contrôles présentent des données de longueurs variables, comme l'Explorateur Windows.

Pour permettre à un utilisateur de redimensionner un contrôle ancré au moment de l'exécution, ancrer le contrôle à redimensionner au bord d'un conteneur, puis ancrez un contrôle Splitter sur le même côté de ce conteneur.



3.11	MainMenu ContextMenu.	
-------------	----------------------------------	--

Main Menu

On peut ajouter un menu dans une fenêtre.

Beaucoup d'applications contiennent un menu.

Exemple de menu :

Fichier

Ouvrir

Fermer

Imprimer

Quitter

Edition

Couper

Copier

Coller

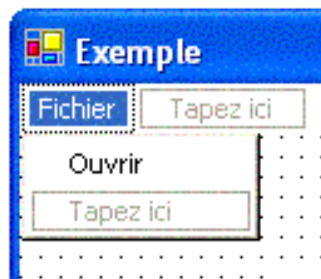
..

On remarque que le contenu des menus est **standardisé** afin que l'utilisateur s'y retrouve sans aide (L'utilisateur lit, à mon avis, rarement les aides !!)

Comment créer un menu ?

En allant dans la boîte à outils, chercher un **main menu** et en le déposant sur la fenêtre : il apparaît en dessous de la fenêtre.

Pour 'dessiner' le menu, il suffit de mettre le curseur sur le menu en haut de la fenêtre, ou est écrit 'Tapez ici' : tapez le texte du menu, ('Fichier' par exemple).



Il apparaît automatiquement un 'Tapez Ici' pour les lignes dessous ou le menu suivant.

Les lignes du menu sont nommées automatiquement MenuItem1, MenuItem2..

Quand le curseur est sur une ligne du menu, la fenêtre de propriété donne les propriétés de la ligne :

La propriété **ShortKey** permet de créer un raccourci.

La propriété **Checked** permet de cocher la ligne

La propriété **Visible** permet de faire apparaître ou non une ligne.

Si vous double-cliquez sur la ligne du menu vous voyez apparaître :

```
Private Sub MenuItem1_Click(ByVal sender As
```


System.Object, [ByVal](#) e [As System.EventArgs](#))
[Handles MenuItem1.Click](#)

[End Sub](#)

C'est la procédure événement liée à la ligne du menu.

Quand l'utilisateur clique sur une ligne du menu c'est le code contenu dans cette procédure qui est effectué.

Menu Contextuel

C'est un menu qui s'ouvre quand, le curseur de l'utilisateur est sur un objet, et qu'il clique sur le bouton droit de la souris.

En allant dans la boîte à outils, chercher un **Context menu**, on le dépose sur la fenêtre : il apparaît en dessous de la fenêtre.

Si on le sélectionne avec la souris, il apparaît en haut et comme pour le menu principal, on peut ajouter des lignes.

Il faut ensuite affecter ce Context Menu à un contrôle; pour cela donner à la propriété [ContextMenu](#) du contrôle le nom du ContextMenu.

[TextBox1.ContextMenu= ContextMenu1](#)

Si vous double-cliquez sur une ligne du menu vous voyez apparaître les procédures événement correspondantes.



Site LDF Cours VB.net



3.12	Avoir le Focus.		Suivant	Sommaire
------	-----------------	--	-------------------------	--------------------------

Nous allons étudier comment un objet de l'interface devient 'actif'.

Lorsqu'une fenêtre ou un contrôle est actif on dit qu'il a le **focus**.

Si une fenêtre prend le focus, sa barre de titre en haut prend la couleur active, si c'est un contrôle texte, le curseur apparaît dedans.

Comment donner le focus à une fenêtre ?

Si une fenêtre est visible la méthode **Activate** lui donne le focus.

```
Form1.Activate()
```

Dans ce cas l'évènement `Form1_Activated` survient.

La méthode `Deactivate` est déclenché quand la fenêtre perd le focus.

Comment donner le focus à un contrôle

?

Avec la méthode **Focus**

```
TxtNom.Focus()
```

Avec la méthode **Select**:

```
TxtNom.Select()
```

 'donne le focus à la zone de texte Txnom et met le curseur dedans.

On peut la surcharger et en plus sélectionner une portion du texte:

```
TxtNom.Select(2,3)
```

 'donne le focus et sélectionne 3 caractères à partir du second.

ou forcer à **ne rien sélectionner** (second argument à 0).

On peut interdire a un contrôle le focus en donnant la valeur False à sa propriété **CanFocus**.

Aussi avant de donner le focus il est préférable de vérifier s'il peut le prendre:

```
If TxtNom.CanFocus then
```

```
    TxtNom.Focus()
```

```
End If
```

L'évènement **GotFocus** se produit quand le contrôle **prend** le focus.

```
Private Sub TxtNom_GotFocus..
```

```
End Sub
```

Cascade d'évènement quand on prend ou on perd le focus:

Il se déclenche dans l'ordre:

Enter

Se produit quand l'utilisateur entre dans le contrôle.

GotFocus

Se produit quand le contrôle prend le focus.

Leave

Se produit quand le focus quitte le contrôle.

Validating

Se produit lorsque le contrôle est en cours de validation.

On va quitter le contrôle, il faut vérifier la validité avant. La validation c'est vous qui devez la faire!!!

Pour un bouton par exemple se produit lorsque l'on quitte le bouton, cela permet de contrôler la validité de certaines données et si nécessaire d'interdire de quitter le contrôle si certaines conditions ne sont pas remplies:

```
Private Sub Button1_Validating
  ((ByVal sender As Object, ByVal e As
  System.ComponentModel.CancelEventArgs)
  Handles Button1.Validating
  If condition then
    e.Cancel = True    'Annuler la
    perte du focus: le focus reste sur
    Button1
  End If
```

```
End Sub
```

Validated

Se produit lorsque le contrôle à terminé sa validation

LostFocus

L'évènement **LostFocus** se produit quand le contrôle **perd** le focus.

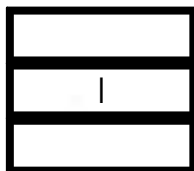
Si la propriété **CauseValidating** du contrôle a la valeur **false**, les évènements **Validating** et **Validated** sont supprimés.

Les évènements **Enter** et **Leave** sont supprimés dans les formulaire (fenêtres) Les évènements équivalents dans la classe **Form** sont les évènements **Activated** et **Deactivate**.

Certains contrôles ne peuvent pas avoir le focus, comme les labels par exemple.

Usage du clavier pour passer d'un contrôle à l'autre.

Dans une application où un utilisateur tape beaucoup de données dans de multiples contrôles, il passe souvent d'un contrôle (TextBox par exemple) au suivant avec la touche TAB.



Comment permettre cela? Chaque contrôle à une propriété **TabIndex** qui s'incrémente automatiquement de 0 à 1, 2, 3...quand en cours de conception on ajoute des contrôles sur une fenêtre.

Lorsque l'utilisateur appuie sur TAB, le focus passe au contrôle qui a le **TabIndex** immédiatement supérieur.

On peut modifier le **TabIndex** des contrôles pour modifier l'ordre de tabulation.

Quand **TabStop** a la propriété False (au niveau d'un contrôle) celui ci est exclu de l'ordre de tabulation et le focus ne s'arrête pas.

Raccourcis clavier.

Dans beaucoup d'applications certains contrôles ont un raccourci clavier:

Exemple: **Nouveau** est une ligne de menu. N étant souligné, **ALT-N** déclenche la ligne de menu, donne le focus au contrôle.

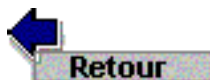
Comment faire cela: Dans la propriété Text du contrôle, quand on tape le texte en mode conception, il faut mettre un '&' avant la lettre qui servira de raccourci clavier.

'&Nouveau' dans notre exemple affichera bien **Nouveau** et **ALT N** fonctionnera.

Pour une **TextBox**, la propriété text ne peut pas être utilisée, aussi il faut mettre devant la textBox un contrôle label (qui lui ne peut pas avoir le focus), si les TabIndex du label et du TextBox se suivent, le fait de faire le raccourci clavier du label donne le focus au TextBox.

Nom

Exemple quand l'utilisateur tape Alt-N, le focus va dans le TextBox dessous.



3.13

ToolBar StatusBar

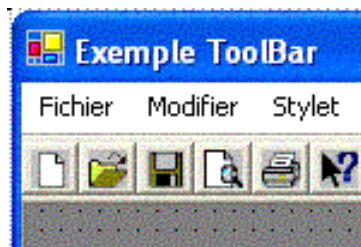
Suivant

Sommaire

Comment mettre une barre de bouton en haut et une barre d'état en bas?

La barre de bouton.

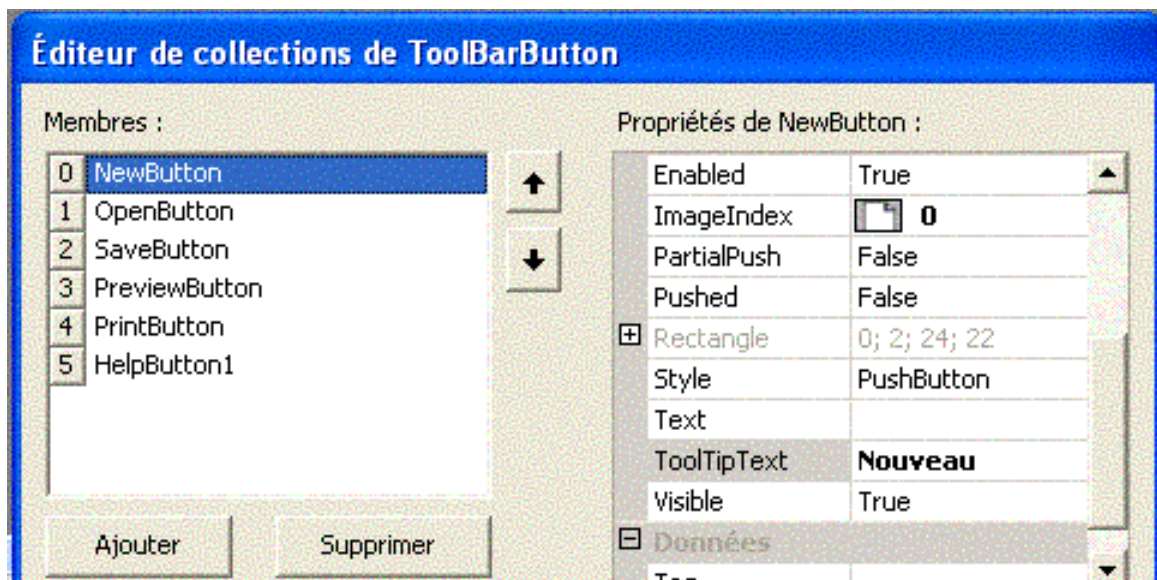
Voici un exemple classique, sous le menu il y a une barre de bouton: **Nouveau, Ouvrir, Enregistrer, Chercher, Imprimer...**



Allez chercher dans la boîte à outils un contrôle **ToolBar**, il se place en haut, sous le menu. Mettre aussi un **ImageList**. (Un contrôle ImageList est un contrôle qui stocke des images, chaque image étant chargée en mode conception et repérée par un numéro (0,1,2,3..))

Dans le ToolBar mettre dans la propriété **ImageList** le nom du contrôle ImageList qui contient les images des boutons.

Ouvrir la **collection Buttons** dans la fenêtre des propriétés de le ToolBar pour pouvoir ajouter ou modifier les boutons:



Vous pouvez ajouter ou enlever des boutons.

Chaque bouton a **ses propriétés** affichées à droite:

Name Nom du Bouton Exemple NewButton.

ImageIndex donne le numéro de l'image (contenue dans l'Imagelist) qui doit s'afficher dans le bouton.

ToolTipText donne le texte du ToolTip (Carré d'aide qui apparaît quand on est sur le bouton) Il faut aussi que la propriété ShowToolTip de la Toolbar soit à True

L'évènement déclenché par le click de l'utilisateur sur un bouton est: `ToolBar1_ButtonClick`

L'argument `e` contient les arguments de l'évènement click de la Toolbar. `e.Button` contient le nom du bouton qui a déclenché l'évènement. Pour chaque nom de bouton on appellera la procédure correspondante: `NewDoc()`, `Open()`...

Comme d'habitude il suffit de double-cliquer sur la Toolbar pour faire apparaître `ToolBar1_ButtonClick`

Voici le code complet:

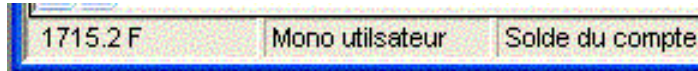
```
Private Sub ToolBar1_ButtonClick(ByVal Sender As System.Object,
ByVal e As System.Windows.Forms.ToolBarButtonClickEventArgs)
Handles toolBar1.ButtonClick
If e.Button Is NewButton Then
    NewDoc()
ElseIf e.Button Is OpenButton Then
    Open()
ElseIf e.Button Is SaveButton Then
    Save()
ElseIf e.Button Is PreviewButton Then
    PrintPreview()
```


...

```
End If
End Sub
```

Contrôle StatusBar.

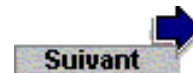
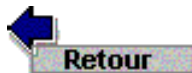
La barre d'état se trouve en bas de la fenêtre et **affiche des informations relatives aux opérations en cours.**



Ajouter un StatusBar au formulaire. Dans la fenêtre des propriétés du StatusBar, la **collection Panels** contient les zones d'affichage du StatusBar.

Dans le code, pour modifier le texte d'une zone faire:

```
StatusBar1.Panels(0).Text="1715.2F"
```



Site LDF Cours : VB.net



3.14

Image.

[Suivant](#)
[Sommaire](#)

Comment afficher des images ?

Le contrôle 'PictureBox':

Le contrôle **PictureBox** sert à afficher des graphismes au format bitmap, GIF, JPEG, métafichier ou icône (Extension **.BMP .GIF .JPG .WMF .ICO**)

L'image à affichée est déterminée par la propriété **Image**, laquelle peut être définie au moment de l'exécution ou du design. La propriété **SizeMode** détermine la façon dont l'image et le contrôle se dimensionnent l'un par rapport à l'autre.



On peut charger une image en mode conception par le fenêtre 'propriétés' et la propriété 'Image' ou dans le code:

```
PictureBox1.Image = Image.FromFile("vimage.gif")
```

(L'objet de la Classe Image charge une image d'un fichier puis l'affecte à la propriété Image.)

ou par

```
PictureBox1.Image.FromFile("vcar1.gif")
```

cette syntaxe ne marche pas!!! Pourquoi?

Ho! merveille, les GIF animés sont acceptés et s'animent sous VB.

Comment effacer une image?

```
If Not (PictureBox1.Image Is Nothing) Then
    PictureBox1.Image.Dispose()
    PictureBox1.Image = Nothing
End If
```

Les objets de la Classe Image ont comme d'habitude des propriétés et des méthodes.

La méthode **RotateFlip** permet par exemple d'effectuer une rotation de l'image; quand on tape le code, VB donne automatiquement la liste des paramètres possible.

```
PictureBox1.Image.RotateFlip(RotateFlipType.Rotate90FlipX)
```

La méthode **Save** sauvegarde l'image dans un fichier.

```
PictureBox1.Image.Save("c:\image.bmp")
```

Noter bien que le nom de l'extension suffit à imposer le format de l'image sur le disque.

On peut charger une image .GIF puis la sauvegarder en .BMP

Il y a bien d'autres propriétés gérant les dimensions, la palette de l'image.

Il est possible de définir une couleur comme 'transparente': voir le page sur les boutons.

La propriété 'Image' des contrôles:

De nombreux contrôles Windows Forms peuvent afficher des images. L'image affichée peut être une icône symbolisant la fonction du contrôle ou une image ; par exemple, l'image d'une disquette sur un bouton indique généralement une commande d'enregistrement. L'icône peut également être une image d'arrière-plan conférant au contrôle une certaine apparence.

Pour tous les contrôles affichant des images:

- l'image peut être définie à l'aide des propriétés **Image** ou **BackgroundImage** directement **en mode Design** par la fenêtre des propriétés. Il faut sélectionner Image puis cliquez sur "...", et choisir un fichier contenant une image. Dans ce cas, une fois chargée, l'image fait partie intégrante du programme. (Il n'est pas utile de fournir le fichier .BMP ou .GIF avec l'application)
- Lorsque le programme 'tourne' on peut aussi charger une Image. Le code affecte à la propriété Image ou BackgroundImage un objet de type System.Drawing.Image, en général, vous utiliserez la méthode **FromFile** de la classe Image pour charger

une Image à partir d'un fichier. (Dans ce cas le fichier contenant l'image doit être fourni)

Exemple pour un bouton:

```
button1.Image = Image.FromFile("C:\Graphics\MyBitmap.bmp")  
' Aligne l'image.  
button1.ImageAlign = ContentAlignment.MiddleRight
```

Exemple pour un label:

```
Dim Label1 As New Label()  
Dim Image1 As Image  
  
Image1 = Image.FromFile("c:\MyImage.bmp")  
  
' modifier la taille du label pour qu'il affiche l'image.  
  
Label1.Size = Image1.Size  
  
' Mettre l'image dans le label.  
  
Label1.Image = Image1
```

Si on renseigne la propriété Image, on ne peut pas utiliser en même temps la propriété ImageList décrite ci-dessous.

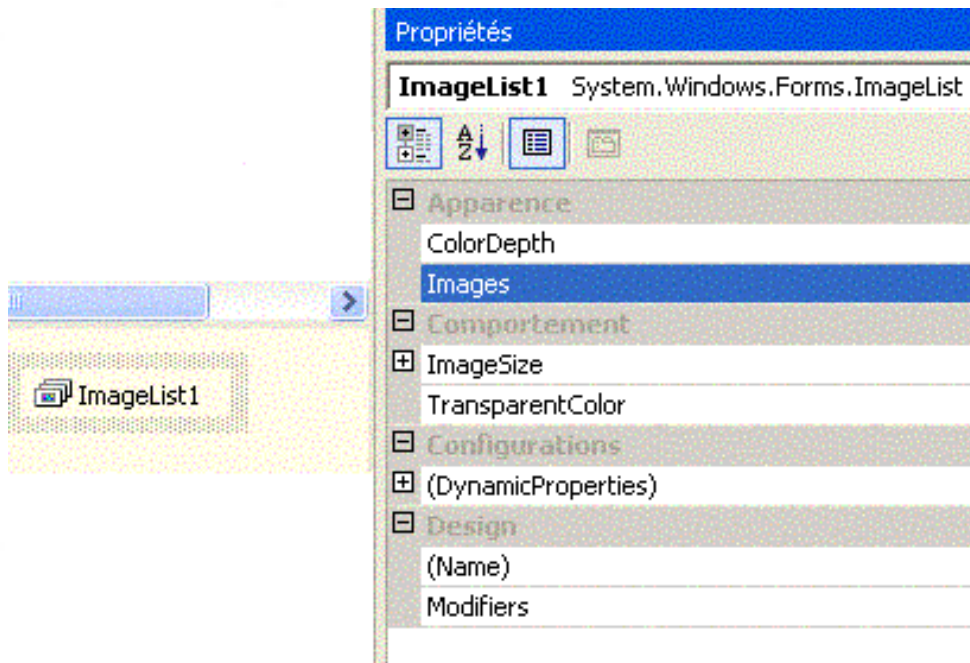
Le contrôle ImageList:

Il sert de conteneur à images, c'est une **collection d'images**. les images qu'il contient seront utilisées par d'autres contrôles (PictureBox, ListView, TreeView, Button....)

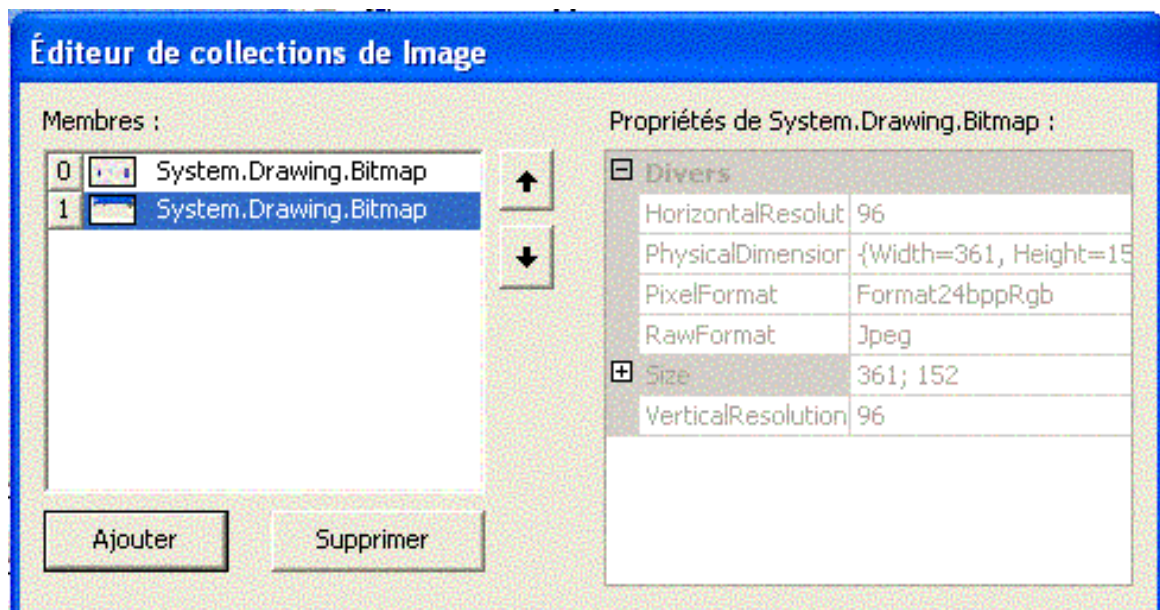
Il n'est pas visible en exécution.

En conception il apparaît en bas sous la fenêtre. A droite figurent ses propriétés, en particulier, la **collection Images** qui contient les images et la propriété **TransparentColor** qui indique la couleur qui doit être transparent, c'est à dire non visible.

Il faut l'ajouter au formulaire, il apparaît en dessous.



Si je clique sur le bouton '...' en face de **Images**, l'éditeur de collections d'image s'ouvre.



On peut ajouter des images avec le bouton 'Ajouter'.

L'ImageList est ainsi chargé.

Ensuite pour utiliser une image de l'ImageList dans un autre contrôle, il faut modifier les propriétés de cet autre contrôle(un bouton par exemple):

La propriété **ImageList** du bouton doit contenir le nom du contrôle imageList et **ImageIndex** du bouton doit contenir l'index de l'image dans l'ImageList.

```
btOk.ImageList = imagelist1
```

```
btOk.ImageIndex = 2
```

Un ImageList peut aussi être **chargé par code** :

```
imageList1.Images.Add(Image.FromFile(NomImage))
```

On ajoute à la collection Images une image venant d'un fichier nommé NomImage.

On peut surcharger la méthode Add en fournissant en plus la couleur transparente.

```
imageList1.Images.Add(Image.FromFile(imageToLoad),  
CouleurTransparente)
```

La taille des images peut aussi être modifiée par code :

```
imageList1.ImageSize = New Size(255, 255)  
imageList1.TransparentColor = Color.White
```



Retour



Index





Sommaire



Suivant

Site **LDF** Cours **VB.net**



3.30	<p>calcul de l'IMC.</p> <p>Révision++</p> <p>Structuration des programmes.</p>		
		Suivant	Sommaire

Ce chapitre permet de 'réviser' pas mal de notion.

Qu'est ce que l'IMC?

L'index de masse corporelle est très utilisé par les médecins. Il est calculé à partir du poids et de la taille:

$IMC = \text{Poids} / (\text{Taille} * \text{Taille})$ (avec Poids en Kg, Taille en mètres)

Cela permet de savoir si le sujet est

maigre ($IMC < 18.5$)

normal (IMC idéale=22)

en surpoids ($IMC > 25$)

obèse ($IMC > 30$).

On peut calculer le poids idéal par exemple $PI = 22 * T * T$

Nous allons détailler ce petit programme:

Quel est le cahier des charges du programme?

L'utilisateur doit pouvoir:

Saisir un poids, une taille, cliquer sur un bouton 'Calculer'

Les routines doivent:

Vérifier que l'utilisateur ne fait pas n'importe quoi.

Calculer et afficher les résultats: l'IMC mais aussi, en fonction de la taille, le poids idéal, les poids limites de maigreur, surpoids, obésité.

Création de l'interface

Il faut **2 zones de saisie** pour saisir le poids et la taille:

On crée 2 'TextBox' que l'on nomme

TextBoxPoids

TextBoxTaille.

On laisse la propriété **Multiline** à **False** pour n'avoir qu'une ligne de saisie.

Pour afficher les résultats , on crée 5 'label' les uns sous les autres. (Pour aller plus vite et que les labels soient de la même taille, on en crée un puis par un copier et des coller, on crée les autres)

labelImc 'pour afficher l'Imc

labelPi 'pour afficher le poids idéal

labelM 'pour afficher le poids limite de la maigreur.

labels 'pour afficher le poids limite du surpoids

labelO 'pour afficher le poids limite de l'obésité.

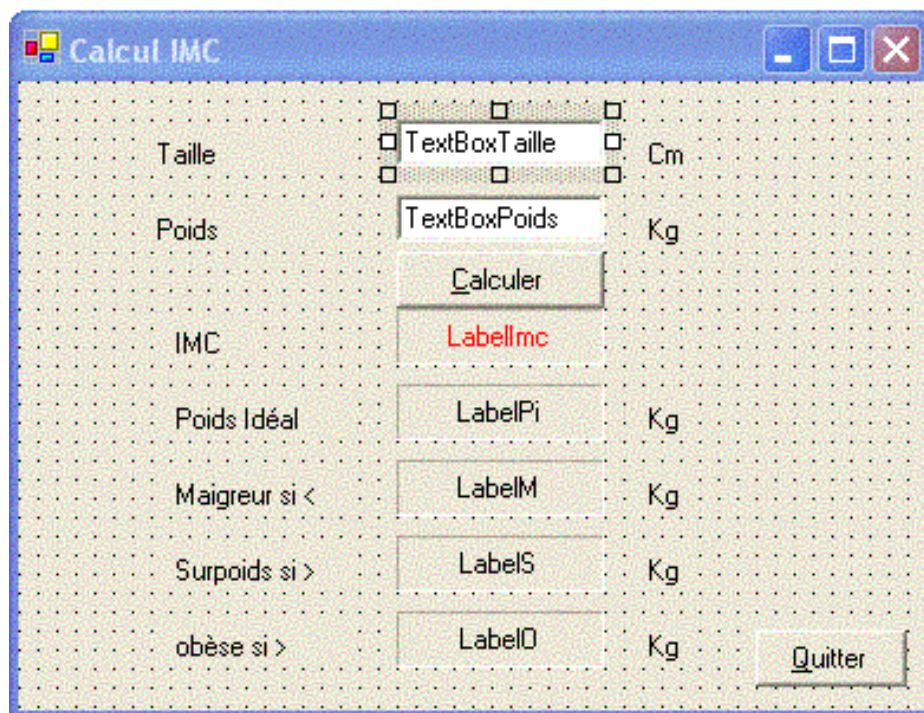
Ensuite on ajoute **des labels** devant et derrière chaque TextBox pour indiquer devant, ce qu'ils contiennent et derrière, l'unité.

On ajoute 2 boutons:

ButtonCalcul ayant pour propriété Text= "&Calculer"

ButtonQuitter ayant pour propriété Text= "&Quitter"

Cela donne :



Pour faire beau :

La propriété Text de la fenêtre contient "Calcul IMC", pour afficher cela dans la barre de titre.

La propriété ForeColor de labelImc est en rouge.

La propriété BorderStyle des labels a la valeur 'Fixed3d' ce qui rend les bords visibles.

Ajout du Code

La procédure événement `Form1_Load` qui se déclenche lorsque la fenêtre se charge **initialise** les zones d'affichage en les vidant :

```
Private Sub Form1_Load(..)
    TextBoxTaille.Text = ""
    TextBoxPoids.Text = ""
    LabelImc.Text = ""
    LabelPi.Text = ""
    LabelM.Text = ""
    LabelS.Text = ""
    LabelO.Text = ""
End Sub
```

La procédure `ButtonCalcul_Click` qui se déclenche lorsque l'utilisateur clique sur le bouton 'Calculer' contient le code principal.

Voici la totalité du code, on le détaillera dessous.

```

Private Sub ButtonCalcul_Click(..)

Dim sPoids As Single 'Variable Single contenant le poids
Dim sTaille As Single 'Variable Single contenant la
taille
'*****Contrôle de validité des entrées*****
'Les valeurs saisies sont-elles numérique?
If Not (IsNumeric(TextBoxTaille.Text)) Then
    MsgBox("Entrez une valeur numérique pour la taille")
    Exit Sub
End If
If Not (IsNumeric(TextBoxPoids.Text)) Then
    MsgBox("Entrez une valeur numérique pour le poids")
    Exit Sub
End If
'Convertir les textes saisis en single
' et les mettre dans les variables
sTaille = CType(TextBoxTaille.Text, Single) / 100
sPoids = CType(TextBoxPoids.Text, Single)
'Les valeurs saisies sont-elles cohérentes?
If sTaille < 50 Or sTaille > 250 Then
    MsgBox("Entrez une taille valide")
    Exit Sub
End If
If sPoids < 20 Or sPoids > 200 Then
    MsgBox("Entrez un poids valide")
Exit Sub
End If
'Effectuer les calculs et afficher les résultats.
LabelImc.Text = (Math.Round(sPoids / (sTaille * sTaille),
2)).ToString
LabelPi.Text = (Math.Round(22 * (sTaille * sTaille),
2)).ToString
LabelM.Text = (Math.Round(18.5 * (sTaille * sTaille),
2)).ToString
LabelS.Text = (Math.Round(25 * (sTaille * sTaille),
2)).ToString
LabelO.Text = (Math.Round(30 * (sTaille * sTaille),
2)).ToString
End Sub

```

Détaillons:

Quelles sont les différentes étapes?

On déclare les variables.

On vérifie que ce qui a été tapé est numérique.

On convertit le texte qui est dans la TextBox en Single

On teste si les valeurs de poids et taille sont
cohérentes.

On fait le calcul et on affiche.

Déclaration de variables.

```

Dim sPoids As Single 'Variable Single contenant le poids
Dim sTaille As Single 'Variable Single contenant la taille

```

Ce sont des variables 'privées' propre à la procédure (utilisation de Dim ou Private).

Contrôle de validité:

L'utilisateur est sensé taper un poids et une taille puis cliquer sur le bouton 'Calculer'. Mais **il ne faut absolument pas lui faire confiance**: il a peut-être oublié de taper le poids ou a donné une taille=0 (l'ordinateur n'aime pas diviser par 0!!), il a peut-être fait une faute de frappe et tapé du texte!!..

Donc il faut tester ce qui a été tapé, s'il y a erreur, on prévient l'utilisateur avec une 'MessageBox' puis on sort de la routine par (Exit Sub)sans effectuer de calcul.

Ici par exemple, on teste si le texte saisi dans la zone taille n'est pas numérique:

```
If Not (IsNumeric(TextBoxTaille.Text)) Then
    MsgBox("Entrez une valeur numérique pour la taille")
    Exit Sub
End If
```

Amélioration: On aurait pu automatiquement effacer la valeur erronée et placer le curseur dans la zone à ressaisir:

```
If Not (IsNumeric(TextBoxTaille.Text)) Then
    MsgBox("Entrez une valeur numérique pour la taille")
    TextBoxTaille.Text=""
    TextBoxTaille.Select()
    Exit Sub
End If
```

Conversion:

Si le texte est bien 'Numéric', on fait la conversion en réel simple précision (Single)

```
sTaille = CType(TextBoxTaille.Text, Single) / 100
```

On utilise **CType** pour convertir une String en Single.

On divise taille par 100 car l'utilisateur a saisi la taille en centimètre et les formules nécessitent une taille en mètre.

Problème du séparateur décimal dans les saisies.

Pourquoi saisir la taille en Cm? c'est pour éviter d'avoir à gérer le problème du séparateur décimal.

Si la taille était saisie en mètre, l'utilisateur aurait-il tapé "1.75" ou "1,75"?

On rappelle que pour convertir un texte en Single VB accepte le point et pas la virgule.

Pour ma part, si j'avais demandé de saisir des mètres, voici ma solution: j'ajouterais en début de routine une instruction transformant les ',' en '.':

```
TextBoxTaille.Text = Replace(TextBoxTaille.Text, ",", ".")
```

Faire les calculs et afficher les résultats.

Je fais le calcul:

```
sPoids / (sTaille * sTaille)
```

J'arrondis à 2 décimales après la virgule grâce à `Math.Round(,2)`:
`Math.Round(sPoids / (sTaille * sTaille), 2)`

Je convertis en String:

```
(Math.Round(sPoids / (sTaille * sTaille), 2)).ToString
```

J'affiche dans le label 'labelImc':

```
LabelImc.Text = (Math.Round(sPoids / (sTaille * sTaille),  
2)).ToString
```

(J'aurais pu aussi ne pas arrondir le calcul mais formater l'affichage pour que 2 décimales soient affichées)

La procédure `ButtonQuitter_Click` déclenchée quand l'utilisateur clique sur le bouton 'Quitter' ferme la seule fenêtre du projet (c'est Me , celle où on se trouve), ce qui arrête le programme.

```
Private Sub ButtonQuitter_Click()  
    Me.Close()  
End Sub
```

Structuration:

Ici on a fait simple: une procédure événement calcule et affiche les résultats.

On aurait pu, dans un but didactique '**structurer**' le programme.

On aurait pu découper le programme en procédure.

- Une procédure faisant le calcul.

- Une procédure affichant les résultats.

Si plusieurs procédures utilisent les mêmes variables il y a dans ce cas 2 possibilités:

- Mettre les variables en 'Public' dans un module Standard.

- Utiliser des variables privées et les passer en paramètres.

Première solution: Variables 'Public'

Créer dans un module standard des variables 'Public' pour stocker les variables Poids et Taille, résultats (Public sIMC A Single par exemple), créer dans ce même module standard une procédure Public nommée 'Calculer' qui fait les calculs et met les résultats dans les variables 'Public'; enfin dans le module de formulaire créer une procédure 'AfficheResultat' affichant les résultats.

Module standard:

'Déclaration de variables Public

```
Public sPoids As Single
```

```
Public sTaille As Single
```

```
Public sIMC A Single
```

```
..
```

'Procédure Public de calcul

```
Public Sub Calculer
```

```
    sIMC=Math.Round(sPoids / (sTaille * sTaille), 2)
```

```
    ...
```

End Sub

Module de formulaire Form1:

'Procédure évènement qui appelle les divers routines

Private Sub ButtonCalculer_Click

...

sTaille = CType(TextBoxTaille.Text, Single) / 100

Calculer() 'Appelle la routine de calcul

AfficheResultat() 'Appelle la routine d'affichage

End Sub

'routine d'affichage toujours dans le formulaire

Private Sub AfficheResultat()

LabelImc.Text = sIMC.ToString

...

End Sub

On voit bien que la routine de Calcul est générale et donc mise dans un module standard et d'accès 'Public', alors que la routine d'affichage affichant sur Form1 est privée et dans le module du formulaire.

Seconde solution: Variables 'Privées' et passage de paramètres

On peut ne pas créer de variables 'public' mais créer des fonctions (CalculIMC par exemple) à qui on passe en paramètre le poids et la taille et qui retourne le résultat du calcul. Une procédure AfficheResultatIMC récupère en paramètre la valeur de l'IMC à afficher.

Module standard:

'Pas de déclaration de variables Public

..

'Function Public de calcul: reçoit en paramètre le poids et la taille

'retourne l'Imc

Public Function CalculerIMC (T as Single, P As Single) As Single

Return Math.Round(P / (T*T), 2)

End Sub

Module de formulaire Form1:

'Procédure évènement qui appelle les divers routines

Private Sub ButtonCalculer_Click

...

sTaille = CType(TextBoxTaille.Text, Single) / 100

'Appelle de la routine calcul avec l'envoi de paramètres sPoids et sTaille

'Au retour on a la valeur de L'imc que l'on envoie à la routine d'affichage.

AfficheResultatIMC(CalculerIMC(sTaille, sPoids)) 'Appelle la routine d'affichage

End Sub

'routine d'affichage

```
Private Sub AfficheResultatIMC(I As Single)
```

```
    LabelImc.Text = i.ToString
```

```
End Sub
```

Remarque:

La ligne `AfficheResultatIMC(CalculerIMC(sTaille, sPoids))`

est équivalente à:

```
Dim s As single
```

```
s=(CalculerIMC(sTaille, sPoids)
```

```
AfficheResultatIMC(s)
```

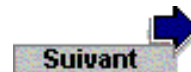
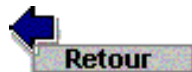
mais on se passe d'une variable temporaire.

Conclusion:

Faut-il travailler avec des variables Public ou passer des paramètres?

Réponses:

A mon avis, les 2 et "ça dépend des cas"!!!(Bien la réponse).



site **LDF** Cours : **VB.net**

3.31	Ordre des instructions.	 Suivant	 Sommaire
------	--------------------------------	--	---

Dans quel ordre écrire dans un module?

Contenu des modules.

Le code Visual Basic est stocké dans des modules (modules de formulaire, modules de classe ..), chaque module est dans un fichier ayant l'extension '.vb'. Les projets sont composés de plusieurs fichiers '.vb', lesquels sont compilés pour créer des applications.

Respecter l'ordre suivant :

1. Instructions **Option** toujours en premier. (force des contraintes de déclaration de variables, de conversion de variables, de comparaison)
2. Instructions **Imports** (charge des espaces de nom)
3. Procédure **Main** (la procédure de démarrage si nécessaire)
4. Instructions **Class**, **Module** et **Namespace**, le cas échéant

Exemple:

```
Option Explicit On
```

```
Imports System.AppDomain
```

```
Imports Microsoft.VisualBasic.Conversion
```

```
Public Class Form1
```

```
Inherits System.Windows.Forms.Form
```

```
    Dim WithEvents m As PrintDocument1
```



```
#Region " Code généré par le Concepteur Windows Form "
```

```
    Public d As Integer
```

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles End Sub
```


Site **LDF** Cours **VB.net**



E 3.1	Exemples : Conversion F/E.		
		Suivant	Sommaire

Comment créer un programme de conversion Francs=>Euros et Euros=> Francs ?

Euros

Francs:

Il y a une zone de saisie Euros, une zone Francs, si je tape dans la zone Euros '2' il s'affiche '13.12' dans la zone Francs; cela fonctionne aussi dans le sens Francs=>Euros.

Comment faire cela?

Un formulaire affichera les zones de saisie, un module standard contiendra les procédures de conversion.

On crée un formulaire contenant :

2 TextBox **BoiteF** et **BoiteE**, leurs propriétés Text=""

2 labels dont la propriété Text sera ="Euros" et "Francs",

on les positionnera comme ci-dessus.

Dans le formulaire, je dimensionne un flag (ou drapeau): **flagAffiche**, il sera donc visible dans la totalité du formulaire. Je l'initialise à True.

```
Public Class Form1
Inherits System.Windows.Forms.Form
Dim flagAffiche As Boolean = True
```

Comme la conversion doit se déclencher automatiquement lorsque le texte de BoiteF ou BoiteE change, j'utilise les évènements '**TextChanged**' de ces TextBox:

Pour la conversion Euros=>Francs, dans la procédure TextChanged de BoiteE, je récupère le texte tapé (BoiteE.Text), j'**appelle la fonction ConversionEF** en lui envoyant comme paramètre ce texte. La fonction me retourne un double que je transforme en string et que j'affiche dans l'autre TextBox(BoiteF).

```
Private Sub BoiteE_TextChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
BoiteE.TextChanged
```

```
    If flagAffiche = True Then
```

```
        flagAffiche = False
        BoiteF.Text =
        (ConversionEF(BoiteE.Text)).ToString
        flagAffiche = True
```

```
    End If
```

```
End Sub
```

Idem pour l'autre TextBox:

```
Private Sub BoiteF_TextChanged(ByVal sender As Object,
ByVal e As System.EventArgs) Handles BoiteF.TextChanged
```

```
    If flagAffiche = True Then
```

```
        flagAffiche = False
        BoiteE.Text =
        (ConversionFE(BoiteF.Text)).ToString
        flagAffiche = True
```

```
    End If
```

End Sub

End Class

A quoi sert le flag : flagAffiche?

A éviter une boucle sans fin: sans flag, BoiteF_TextChanged modifie BoiteE_Text qui déclenche BoiteE_TextChanged qui modifie BoiteF_Text qui déclenche BoiteF_TextChanged.....

Avec le flag, quand je vais modifier la propriété Text d'une TextBox, le met le flag à False, cela indique à l'autre évènement TextChanged de ne pas lui aussi convertir et afficher.

Enfin il faut **écrire les procédures** qui font la conversion: **ConversionEF** et **ConversionFE** dans un module standard. Ces procédures 'Function' appellent elles mêmes une autre fonction qui arrondi les résultats à 2 décimales.

Pour transformer des Euros en Francs, je les multiplie par 6.55957 puis j'arrondis .

On remarque que ces procédures reçoivent une string en paramètres et retourne un double.

Module Module1

```
Public Function ConversionEF(ByVal e As String) As Double
```

```
    Dim somme As Double
    Dim resultat As Double
    somme = Val(e)
    resultat = Arrondir(somme * 6.55957)
    Return resultat
```

```
End Function
```

```
Public Function ConversionFE(ByVal e As String) As Double
```

```
    Dim somme As Double
    Dim resultat As Double
    somme = Val(e)
    resultat = Arrondir(somme / 6.55957)
    Return resultat
```

```
End Function
```

Enfin la **Function Arrondir** arrondit à 2 décimales: pour cela on multiplie par 100, on arrondit à l'entier avec Round puis on divise par 100.

```
Public Function Arrondir(ByVal Valeur As Double) As Double  
    'arrondi a 2 chiffres après la virgule
```

```
        Return (Math.Round(Valeur * 100)) / 100
```

```
End Function
```

```
End Module
```

A noter que l'on aurait pu utiliser **une surcharge de Round** qui arrondit directement à 2 décimales:

```
Return (Math.Round(Valeur, 2))
```

Exercice:

Quel code mettre dans la procédure Button_Click d'un bouton nommé 'Remise à zéro' qui met les 2 zones de saisie à zéro?

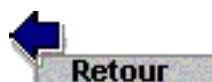
(Penser au flag)

Amélioration:

Si l'utilisateur tape une virgule il y a problème car la fonction Val utilisée pour convertir le nombre saisi en numérique reconnaît uniquement le point, il faut donc transformer les virgules en points avec

```
e = Replace(e, ",", ".")
```

On peut tester si l'utilisateur a bien tapé un nombre, avec la fonction **IsNumeric**.



Site LDF Cours VB.net



<p>E</p> <p>3.2</p>	<p>Exemples :</p> <p>Mensualités d'un prêts.</p>	<p>Suivant</p>	<p>Sommaire</p>
-----------------------------------	--	----------------	-----------------

Comment créer un programme qui calcul les mensualisées d'un prêt ?

Dans l'espace Microsoft.VisualBasic il existe des fonctions financières.

Pmt calcul les mensualités d'un prêt.

Remboursement mensuel= **Pmt**(Rate, NPer, PV, FV, Due)

Rate

Obligatoire. Donnée de type **Double** indiquant le taux d'intérêt par période. Si taux d'intérêt annuel de 10 pour cent et si vous effectuez des remboursements mensuels, le taux par échéance est de $0,1/12$, soit 0,0083.

NPer

Obligatoire. Donnée de type **Double** indiquant le nombre total d'échéances. Par exemple, si vous effectuez des remboursements mensuels dans le cadre d'un emprunt de quatre ans, il y a $4 * 12$ (soit 48) échéances.

PV

Obligatoire. **Double** indiquant la **valeur actuelle** . Par exemple, lorsque vous empruntez de l'argent pour acheter une voiture, le **montant du prêt** correspond à la valeur actuelle (pour un emprunts il est négatif).

FV

Facultatif. **Double** indiquant la valeur future ou le solde en liquide souhaité au terme du dernier remboursement. Par exemple, la valeur future d'un emprunt est de 0 F car il s'agit de sa valeur après le dernier remboursement. Par contre, si vous souhaitez économiser 70 000 F sur 15 ans, ce montant constitue la valeur future. Si cet argument est omis, 0 est utilisée par défaut.

Due

Facultatif. Objet de type Microsoft.VisualBasic.DueDate indiquant la date d'échéance des paiements. Cet argument doit être DueDate.EndOfPeriod si les paiements sont dus à terme échu ou DueDate.BegOfPeriod si les paiements sont dus à terme à échoir (remboursement en début de mois). Si cet argument est omis, DueDate.EndOfPeriod est utilisé par défaut.

Noter que si Rate est par mois NPer doit être en mois; si Rate est en année NPer doit être en année.

```
Sub CalculPret()  
Dim PVal, Taux, FVal, Mensualite, NPerVal As Double  
Dim PayType As DueDate  
  
Dim Response As MsgBoxResult  
Dim Fmt As String  
Fmt = "###,###,##0.00" ' format d'affichage.  
FVal = 0 '0 pour un prêt.  
  
PVal = CDbI(InputBox("Combien voulez-vous emprunter?"))  
Taux = CDbI(InputBox("Quel est le taux d'intérêt  
annuel?"))  
If Taux > 1 Then Taux = Taux / 100 ' Si l'utilisateur à  
tapé 4 transformer en 0.04.  
NPerVal =12* CDbI(InputBox("Durée du prêt (en années)?"))  
Response = MsgBox("Echéance en fin de mois?",  
MsgBoxStyle.YesNo)  
If Response = MsgBoxResult.No Then  
    PayType = DueDate.BegOfPeriod  
Else  
    PayType = DueDate.EndOfPeriod  
End If  
Mensualite = Pmt(Taux / 12, NPerVal, -PVal, FVal, PayType)  
MsgBox("Vos mensualités seront de " & Format(Mensualite,  
Fmt) & " par mois")  
End Sub
```

IPmt calcul les intérêts pour une période.

Calculons le total des intérêts:

```
Dim IntPmt, Total, P As Double
```

Site

```
For P = 1 To TotPmts ' Total all interest.  
IntPmt = IPmt(APR / 12, P, NPerVal, -PVal, Fval, PayType)  
Total = Total + IntPmt  
Next Period  
□
```



Retour



Index

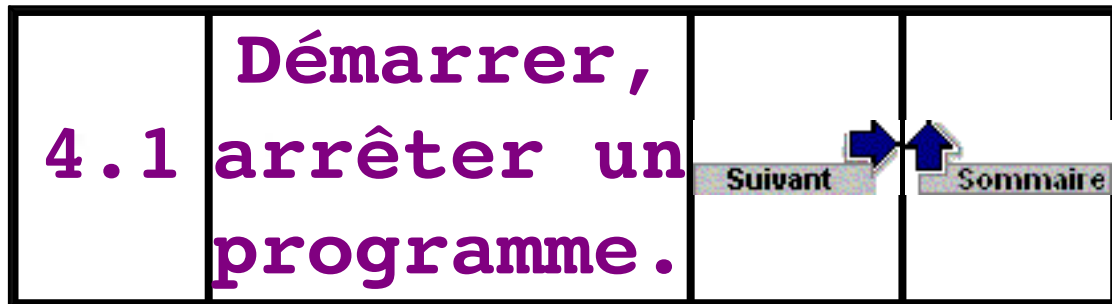


Sommaire



Suivant

Site **LDF** Cours **VB.net**



Quand vous démarrez votre programme, quelle partie du code va être exécutée en premier?

Vous pouvez le déterminer en cliquant sur le menu **Projet** puis **Propriétés de NomduProjet**, une fenêtre **Page de propriétés** du projet s'ouvre.

Sous la rubrique **Objet du démarrage**, il y a une zone de saisie avec liste déroulante permettant de choisir:

-Le nom d'une fenêtre du projet

ou

-Sub Main()

Démarrer par une fenêtre.

Si vous tapez le nom d'une fenêtre du projet, c'est celle-ci qui démarre : cette fenêtre est chargée au lancement du programme et la procédure `Form_Load` de cette fenêtre est effectuée.

Démarrer par Sub Main()

C'est cette procédure `Sub Main` qui s'exécute en premier lorsque le programme est lancé.

Dans ce cas, il faut ajouter dans un module (standard ou d'un formulaire) une Sub nommé `Main()`,

Exemple:

En mode conception `Form1` a été dessinée, C'est le modèle 'la Classe' de la fenêtre qui doit s'ouvrir au démarrage.

Dans `Sub Main()`, on crée une fenêtre de départ que l'on nomme `initForm` avec le moule, la Class `Form1` en 'instancant' la nouvelle fenêtre. Puis on affiche ce formulaire (cette fenêtre) avec `.ShowDialog`

```
Public Shared Sub Main()  
    Dim initForm As New Form1  
    initForm.ShowDialog()  
End Sub
```

Fenêtre Splash

Dans la Sub `Main` il est possible de gérer une fenêtre **Splash**. C'est une fenêtre qui s'ouvre au démarrage d'un programme, qui montre simplement une belle image, pendant ce temps le programme initialise des données, ouvre des fichiers... ensuite la fenêtre 'Splash' disparaît et la fenêtre principale apparaît.

Exemple:

Je dessine `Form1` qui est la fenêtre Spash.

Dans `Form2` qui est la fenêtre principale, j'ajoute:

```
Public Shared Sub Main()  
    Dim FrmSplash As New Form1 'instance la fenêtre Splash  
    Dim FrmPrincipal As New Form2 'instance la feuille  
    principale  
    FrmSplash.ShowDialog() 'affiche la fenêtre Splash  
    en Modale  
    FrmPrincipal.ShowDialog() 'a la fermeture de Splash,  
    affiche la fenêtre principale  
End Sub
```

Dans `Form1` (la fenêtre Splash)

Private Sub Form1_Activated

Me.Refresh() 'pour afficher totalement la fenêtre.

'ici ou on fait plein de choses on ouvre des fichiers ou on perd du temps.

Me.Close()

End Sub

On affiche FrmSplash un moment (Ho! la belle image) puis on l'efface et on affiche la fenêtre principale. Word, Excel.. font comme cela.

Comment arrêter le programme?

Me.Close() 'Ferme la fenêtre en cours

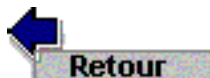
Noter bien Me désigne le formulaire, la fenêtre en cours.

Application.Exit() 'Ferme l'application

Si des fichiers sont encore ouvert, cela les ferme. (Il vaut mieux les fermer avant, intentionnellement par une procédure qui ferme tous les fichiers.)

Attention:

Si le formulaire que vous fermez est le formulaire de démarrage de votre application, votre application se termine.





	Ouvrir une autre 4.2 formulaire (une fenêtre).	Suivant →	← Sommaire
--	---	------------------	-------------------

Rappel:Formulaire=fenêtre

Comment à partir d'un formulaire 'Form1' ouvrir un second formulaire 'Form2' ?

Créer un formulaire:

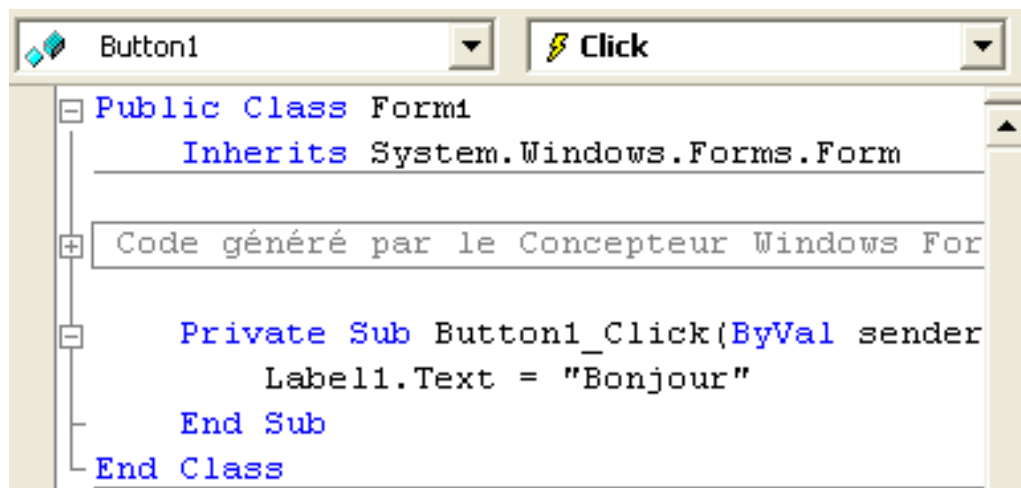
A- On va d'abord créer la Classe Form2

Ajouter un formulaire (Menu Projet, Ajouter un formulaire au projet) nommé Form2 .

On se rend compte que quand on ajoute un formulaire (Form2 par exemple), on crée une nouvelle classe' **Class Form2**' qui hérite de **System.Windows.Forms.Form** , elle hérite donc de toutes les propriétés et méthodes de la Classe Form qui est la classe 'formulaire'.

```
Public Class Form2
```

```
End Class
```



```
Button1 Click
Public Class Form1
    Inherits System.Windows.Forms.Form
    Code généré par le Concepteur Windows For
    Private Sub Button1_Click(ByVal sender
        Label1.Text = "Bonjour"
    End Sub
End Class
```

Elle contient du [code généré automatiquement par le concepteur Windows Forms](#) et les procédures liées aux évènements.

Dessinez dans Form2 les contrôles nécessaires.

B- On va créer le nouvel Objet formulaire (fenêtre):

Pour créer un nouveau formulaire dans le programme, il faut:

- **Instancer un formulaire** à partir du moule, de la Classe Form2 avec le mot `New`.
- **Ouvrir ce formulaire**, le faire apparaître, (avec `ShowDialog`, c'est un formulaire modal)

```
Dim f As New Form2()
```

```
f.ShowDialog()
```

En conclusion:

Le fait d'ajouter un formulaire et des contrôles à un projet crée une Class, (un moule) ce qui permet ensuite d'instancier un objet formulaire.

VB est tolérant : si on dessine un formulaire et ses contrôles et qu'on lance le programme, il accepte de fonctionner bien qu'on ait pas instancé le formulaire.

Dénomination des fenêtres après leur création.

Une procédure crée un formulaire par `Dim f As New Form2`

- **Dans** le formulaire `f` créé:

Utiliser `Me` pour désigner le formulaire où on se trouve. (`Form2` ou `f` ne sont pas acceptés)

Exemple:

Le formulaire `f` pourra être fermé par `Me.close()` dans le code du bouton Quitter par exemple.

- **Hors** du formulaire `f`, dans la procédure où a été instancié le formulaire:

Utiliser `f` pour désigner le formulaire.

Exemple:

Si la fenêtre appelante veut récupérer des informations dans le formulaire `f` (un texte dans `txtMessage` par exemple), il faudra écrire.

```
Text=f.txtMessage.Text
```

- Par contre, **hors de la procédure qui a créée** le formulaire, `f` n'est **pas** accessible.



En résumé: Attention donc, si vous instancez un formulaire dans une procédure, elle sera visible et accessible uniquement dans cette procédure .

Cela parait évident car **un formulaire est un objet** comme un autre et sa visibilité obéit aux règles habituelles (J'ai mis malgré tout un certains temps à le comprendre!!).

Si vous voulez créer un formulaire qui soit visible dans la totalité du programme et dont les contrôles ou propriétés soient **accessible par l'ensemble du programme**, il faut l'instancier dans un module standard avec:

```
Public f As New Form2.
```



Un formulaire est un objet et sa visibilité obéit aux règles habituelles: Il peut être instancé dans une procédure, un module, précédé de 'Public' , 'Private'.. ce qui permet de gérer son accessibilité.

Un formulaire est un objet:On peut ajouter à un formulaire des méthodes et des membres:

On a vu que, en fait, il y a création d'une Classe quand on dessine un formulaire, et bien comme dans un module de Classe (on verra cela plus loin), on peut ajouter des propriétés et des méthodes.

Pour ajouter une méthode à un formulaire, il faut créer une Sub Public dans le corps de la fenêtre:

```
Public Sub Imprime()
```

```
    Code d'impression
```

```
End Sub
```

Si une instance de la fenêtre se nomme F, **F.Imprime()** exécute la méthode Imprime (donc la sub Imprime)

De même, pour définir un membre d'un formulaire, il faut ajouter une variable 'public'.

Public Utilisateur As String

Permet d'utiliser en dehors du formulaire `F.Utilisateur`

Si le formulaire a été instancié dans un module de Classe (avec Public), les méthodes et propriétés de ce formulaire seront accessibles partout.

Exemple plus complet:

Avec récupération de données dans le formulaire créé, à partir d'une procédure:

Créer un formulaire en utilisant Form2.

L'ouvrir en formulaire modal. Quand l'utilisateur ferme cette fenêtre modale, récupérer le texte qui est dans txtMessage de cette fenêtre modale.

La ruse c'est de mettre dans le code du bouton Quitter de Form2 `Me.Hide()` pour rendre la fenêtre Form2 invisible mais accessible (et pas `Me.Close()` qui détruirait la fenêtre, le contrôle txtMessage et son contenu).

```
Dim f As New Form2()  
f.ShowDialog()  
Text=f.txtMessage.Text  
f.Close()
```

Une fois que le texte à été récupéré, on faire disparaître la fenêtre f.

En réalité, curieusement, il semble que les propriétés de f soient accessibles même après un Close!! Cela vient du fait que, bien que le formulaire soit fermé, il n'est pas encore détruit.

Autre problème : comment savoir si un formulaire existe, s'il n'existe pas le créer,

s'il existe le rendre visible et lui donner la main :

```
If f Is Nothing Then      'Si f=rien
    f = New Form2
    f.ShowDialog()
Else
    If f.Visible = False Then
        f.Visible = True
    End If
    f.Activate()
End If
```

Fenêtre modale ou non modale:

Un formulaire **modal** est un formulaire qui, une fois ouvert, prend la main, interdit l'usage des autres fenêtres. Pour poursuivre, on ne peut que sortir de ce formulaire.

Exemple typique: une MessageBox est un formulaire modal, les fenêtres d'avertissement dans Windows sont aussi modales.

Pour ouvrir un formulaire modal, il faut utiliser la méthode `.ShowDialog`

```
f.ShowDialog()
```

Noter, et c'est très important, que le code qui suit `.showDialog` est exécuté **après** la fermeture de la fenêtre modale.

Pour avoir un formulaire **non modal** faire

```
f.Show()
```

Dans ce cas le formulaire f s'ouvre, le code qui suit `.Show` est exécuté immédiatement, et il est possible de passer dans une autre fenêtre de l'application sans fermer

f.

Owner

Comment savoir quel formulaire a ouvert le formulaire en cours?(quel est le formulaire parent?)

ShowDialog possède un argument facultatif, *owner*, qu'on peut utiliser afin de spécifier une relation parent-enfant pour un formulaire. Par exemple, lorsque le code de votre formulaire principal affiche une boîte de dialogue, vous pouvez passer **Me** comme propriétaire de la boîte de dialogue, afin de désigner votre formulaire principal comme propriétaire, comme le montre le code de l'exemple suivant :

Dans Form1

```
Dim f As New Form2
```

```
f.ShowDialog(Me)
```

Dans Form2 on peut récupérer le nom du 'propriétaire', du 'parent' qui a ouvert la fenêtre (il est dans **Owner**) et l'afficher par exemple:

```
Label1.text=Me.Owner.ToString
```

Cela affiche: NomApplication.Form1,text text=est le texte de la barre supérieure.

Récupération d'information par DialogResult.

On ouvre un formulaire modal, comment, après sa fermeture, récupérer des informations sur ce qui s'est passé dans ce formulaire modale?

Par exemple, l'utilisateur a-t-il cliqué sur le bouton Ok ou le bouton Cancel pour fermer le formulaire modale?

Pour cela on va utiliser une propriété **DialogResult** des boutons, y mettre une valeur correspondant au bouton, **quand l'utilisateur clique sur un bouton, la valeur de la propriété DialogResult du bouton est assignée à la**

propriété DialogResult du formulaire, on récupère cette valeur à la fermeture du formulaire modal.

Dans le formulaire modal Form2 on met

```
ButtonOk.DialogResult= DialogResult.ok
```

```
ButtonCancel.DialogResult= DialogResult.Cancel
```

Dans le formulaire qui appelle:

```
Form2.ShowDialog()
```

```
If form2.DialogResult= DialogResult.ok then
```

```
    'l'utilisateur a cliqué sur le bouton ok
```

```
End if
```

Remarque:

1. On utilise comme valeur de DialogResult les constantes de l'énumération DialogResult: DialogResult.ok .Cancel .No .Yes .Retry .None
2. Si l'utilisateur clique sur la fermeture du formulaire modal (bouton avec X) cela retourne DialogResult.cancel
3. on peut aussi utiliser la syntaxe:

```
If form2.ShowDialog(Me) = System.Windows.Forms.DialogResult.OK Then
```

 qui permet en une seule ligne d'ouvrir form2 et de tester si l'utilisateur a cliqué sur le bouton ok de form2.
4. La fermeture du formulaire modal par le bouton de fermeture ou l'appel de la méthode Close ne détruit pas toujours le formulaire modal, il faut dans ce cas utiliser la méthode **Dispose pour le détruire.**



Mon truc: De manière générale s'il y a des informations à faire passer d'un formulaire à un autre, j'utilise une variable Publique (nommée BAL comme 'Boite aux lettres' par exemple) dans laquelle je mets l'information à faire passer.

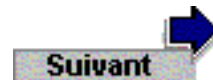
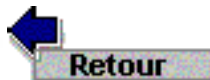
Bouton par défaut.

Parfois dans un formulaire, l'utilisateur doit pouvoir, valider (taper sur la touche 'Entrée') pour accepter et quitter rapidement le formulaire (c'est l'équivalent du bouton 'Ok') ou taper 'Echap' pour sortir du formulaire sans accepter (c'est l'équivalent du bouton 'Cancel').

Il suffit pour cela de donner aux propriétés [AcceptButton](#) et [CancelButton](#) du formulaire, le nom des boutons ok et cancel qui sont sur la feuille.

```
form1.AcceptButton = buttonOk  
form1.CancelButton = buttonCancel
```

Si l'utilisateur tape la touche 'Echap' la procédure [buttonCancel_Click](#) est exécutée.



Site **LDF** Cours **VB.net**



4.3	Traiter les erreurs.	Suivant →	← Sommaire
------------	-----------------------------	------------------	-------------------

Il y a plusieurs types d'erreurs.

Les erreurs de syntaxe.

Les erreurs d'exécution.

Les erreurs de logique.

Les erreurs de syntaxe:

Elle surviennent **en mode conception** quand on tape le code:

Exemple:

A+1=B 'Erreur dans l'affectation

f.ShowDialog 'Faute de frappe, il fallait taper ShowDialog

2 For... et un seul Next

Dans ces cas VB souligne en **ondulé bleue** le code. Il faut mettre le curseur sur le mot souligné, l'explication de l'erreur apparaît.

Exemple: Propriété Text d'un label mal orthographiée:

Site
Label1.Texte() = "12"

'Texte' n'est pas un membre de 'System.Windows.Forms.Label'.

Il faut les corriger immédiatement en tapant le bon code.

Les erreurs d'exécution:

Elle surviennent en mode Run ou lors de l'utilisation de l'exécutable, une instruction ne peut pas être effectuée.

Le logiciel s'arrête brutalement, c'est très gênant!! Pour

l'utilisateur c'est un 'BUG' 

L'erreur est:

- Soit une erreur de conception.

Exemple:

Ouvrir un fichier qui n'existe pas (On aurait du vérifier qu'il existe avant de l'ouvrir!).

Division par zéro.

Utiliser un index d'élément de tableau supérieur au plus grand possible:

```
Dim A(3) As String: A(5)="Toto"
```

- Soit une erreur de l'utilisateur.

Exemple: On lui demande de taper un chiffre, il tape une lettre ou rien puis valide.

Il faut toujours vérifier ce que fait l'utilisateur et prévoir toutes les possibilités.

Exemple: si je demande à l'utilisateur de taper un nombre entre 1 et 10, il faut:

Vérifier qu'il a tapé quelque chose.

Que c'est bien un chiffre (pas des lettres).

Que le chiffre est bien entre 1 et 10.

Sinon il faudra reposer la question.

On voit bien que pour éviter les erreurs d'exécution il est possible:

-D'écrire du code gérant ces problèmes, contrôlant les actions de l'utilisateur..

-Une autre alternative est de capter l'erreur.

A-Capter les erreurs avec Try Catch Finally:

Avant l'instruction supposée provoquer une erreur indiquez: **Essayer (Try)**, si une erreur se produit **Interceptor l'erreur (Catch)** puis poursuivre (après **Finally**)

Try

Instruction susceptible de provoquer une erreur.

Catch

Traitement de l'erreur

Finally

Code toujours exécuté

End Try

Il faut pour que cela fonctionne avoir tapé au préalable **Imports System.IO**

Il est possible d'utiliser Catch pour **recupérer l'objet 'Exception'** qui est généré par l'erreur.

Catch ex As Exception

Cet **objet Exception** à des propriétés:

Message qui contient le descriptif de l'erreur.

Source qui contient l'objet qui a provoqué l'erreur....

`ex.Message` contient donc le message de l'erreur.

Cet **objet Exception** (de l'espace IO) à aussi des **classes dérivées**: **StackOverflowException**; **FileNotFoundException**; **EndOfStreamException**; **FileLoadException**; **PathTooLongException**. Enfin une exception peut provenir de l'espace System: **ArgumentExceptions**; **ArithmeticException**; **DivideByZeroException**.....

Il est possible d'écrire plusieurs instructions Catch avec pour chacune le type de l'erreur à intercepter. (Faisant partie de la classe Exceptions)

Exemple:

On ouvre un fichier par StreamReader , comment intercepter les exceptions suivantes?

Répertoire non valide

Fichier non valide

Autre.

```
Try
  sr= New StreamReader (NomFichier)
Catch ex As DirectoryNotFoundException
  MsgBox("Répertoire invalide")
Catch ex As FileNotFoundException
  MsgBox("Fichier invalide")
Catch ex As Exception
  MsgBox(ex.Message)
End Try
```

Noter que le dernier Catch intercepte toutes les autres exceptions.

On peut encore affiner la gestion par le mot clé **When** qui permet une condition.

Catch ex As FileNotFoundException

```
                When ex.Message.IndexOf ("Mon Fichier.txt")  
>0  
  
                MsgBox ("Impossible d'ouvrir Mon  
Fichier.txt")
```

Si le texte "Mon Fichier.txt" est dans le message, affichez que c'est lui qui ne peut pas être ouvert.

`Exit Try` permet de sortir prématurément.

B-Capter les erreurs avec On error :

On peut aussi utiliser en VB.Net la méthode VB6:

`On Error Goto` permet en cas d'erreur de sauter à une portion de code traitant l'erreur.

On peut lire le numéro de l'erreur qui s'est produite, ce numéro est dans `Err.Number`.

`Err.Description` contient le texte décrivant l'erreur.
`Err.Source` donne le nom de l'objet ou de l'application qui a créé l'erreur.

Quand l'erreur est corrigée, on peut revenir de nouveau à la ligne qui a provoqué l'erreur grâce à `Resume` ou poursuivre à la ligne suivante grâce à `Resume Next`

Exemple:

```
On Error GoTo RoutinedErreur 'Si une erreur se produit se  
rendre à 'RoutineErreur'
```

```
Dim x As Integer = 33
```

```
Dim y As Integer = 0
```

```
Dim z As Integer
```

```
z = x / y ' Crée une division par 0 !!
```

```
RoutinedErreur: ' La Routine d'erreur est ici (remarquer  
' :').
```

```
Select Case Err.Number ' On regarde le numéro de l'erreur.
```

```
Case 6 ' Cas : Division par zéro interdite
```

```
    y = 1 ' corrige l'erreur.
```

```
Case Else
```

```
    ' autres erreurs....
```

```
End Select
```

```
Resume ' Retour à la ligne qui a provoqué l'erreur.
```


Pour arrêter la gestion des erreurs il faut utiliser:

`On Error Goto 0`

Parfois on utilise une gestion hyper simplifiée des erreurs:

Si une instruction 'plante', la sauter et passez à l'instruction suivante, pour cela on utilise:

`On Error Resume Next`

Exemple: On veut effacer un fichier

`On Error Resume Next`

`Kill (MonFichier)`

`On Error goto 0`

Ainsi , si le fichier n'existe pas , cela ne plante pas (on aurait pu aussi vérifier qu'il existe avant de l'effacer).

`On Error Gosub` n'existe plus.

Les erreurs de logique:



Le programme fonctionne, pas d'erreurs apparentes, mais **les résultats sont erronés, faux.**



Il faut donc toujours tester le fonctionnement du programme de multiples fois dans les conditions réelles avec des données courantes, mais aussi avec des données remarquables (limites supérieures, inférieures, cas particuliers..) pour voir si les résultats sont cohérents et exacts.



Et avoir une armée de Bêta-testeurs.

Une fois l'erreur trouvée, il faut en déterminer la cause et la corriger.

Ou bien elle est évidente à la lecture du code ou bien elle n'est pas évidente et c'est l'horreur.

Dans ce dernier cas il faut analyser le fonctionnement du programme pas à pas, instruction par instruction en surveillant la valeur des variables.(voir la rubrique [débogage](#))

Les erreurs les plus communes sont:

Utilisation d'un mauvais nom de variable (La déclaration obligatoire des variables évite cela)

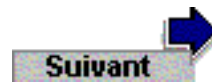
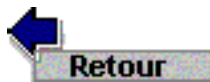
Erreur dans la **portée d'une variable**.

Erreur dans le **passage de paramètres** (Attention au By Val et By Ref)

Erreur dans la **conception de l'algorithme**.



...

Quelques règles permettent de les éviter: voir [Règles de bonne programmation](#).



Site **LDF** Cours : **VB.net**



4.4	Travailler sur une fenêtre multi document.	Suivant 	 Sommaire
-----	---	---	--

Comment créer un programme MDI (Multi Document Interface) ?

Comprendre les programmes MDI :

L'exemple de Word : la fenêtre principale (fenêtres MDI) contient les menus en haut, on peut ouvrir plusieurs documents dans des **fenêtres filles**.

Ci dessous l'exemple de LDF (Programme de comptabilité écrit par l'auteur):



On a une fenêtre MDI (conteneur) contenant 2 fenêtres filles affichant chacune une année de comptabilité.

Dans VB.NET, un **MDIForm** (fenêtre principale MDI) est une fenêtre quelconque dont la propriété **IsMdiContainer** = **True**.

Dans la fenêtre fille, la propriété **MDIParent** indique le conteneur (C'est à dire le nom de la fenêtre MDI) .

Les applications MDI peuvent avoir plusieurs conteneurs MDI.

Exemple d'un programme MDI.

On va créer une Form1 qui est le conteneur.

Une Form2 qui est la fenêtre fille.

Dans Form1 le menu principal contient la ligne '&Nouvelle' qui crée une nouvelle instance de la fenêtre fille.

Création de la fenêtre conteneur parent :

Créer la fenêtre Form1 :

Dans la fenêtre **Propriétés**, affectez la valeur **True** à la propriété **IsMdiContainer**. Ce faisant, vous désignez la fenêtre comme le **conteneur MDI** des fenêtres enfants.

Remarque Affecter la valeur **Maximized** à la propriété **WindowState**, car il est plus facile de manipuler des fenêtres MDI enfants lorsque le formulaire parent est agrandi. Sachez par ailleurs que le formulaire MDI parent prend la couleur système (définie dans le Panneau de configuration Windows).

Ajouter les menus du conteneur :

A partir de la **boîte à outils**, faire glisser un contrôle **MainMenu** sur le formulaire. Créer un élément de menu de niveau supérieur en définissant la propriété **Text** avec la valeur **&File** et des éléments de sous-menu appelés **&Nouvelle** et **&Close**. Créer également un élément de menu de niveau supérieur appelé **&Fenêtre**.

Dans la liste déroulante située en haut de la fenêtre **Propriétés**, sélectionnez l'élément de menu correspondant à l'élément **&Fenêtre** et affectez la valeur **true** à la propriété **MdiList**. Vous activez ainsi le menu **Fenêtre** qui permet de tenir à jour une liste des fenêtres MDI enfants ouvertes et indique à l'utilisateur par une coche la fenêtre enfant active.

Il est conseillé de créer un **module standard** qui instance la **fenêtre principale** et qui contient une procédure **Main** qui affiche la fenêtre principale:

```
Module StandartGénéral
Public FrmMDI as Form1
Sub Main()
    FrmMDI.ShowDialog()
End sub
End Module
```

Noter bien que **FrmMDI** est donc la fenêtre conteneur et est **Public** donc accessible à tous.

Création des fenêtres filles :

Pour créer une fenêtre fille, il suffit de donner à la propriété **MDIParent** d'une **fenêtre** le nom de la fenêtre conteneur.

Dessiner dans **Form2** les objets nécessaire dans la fenêtre fille.

Comment créer une instance de la fenêtre fille à chaque fois que l'utilisateur clique sur le menu '&Nouvelle'?

En premier lieu, déclarez dans le haut du formulaire **Form1** une variable nommée **MDIFilleActive** qui contiendra la fenêtre fille active.

```
Dim MDIFilleActive As Form2
```

La routine correspondant au **MenuItem &Nouvelle** (dans la fenêtre MDI) doit créer une instance de la

fenêtre fille :

```
Protected Sub MDIChildNouvelle_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MenuItem2.Click
```

```
    MDIFilleActive = New Form2()  
    'Indique à la fenêtre fille son 'parent'.  
    MDIFilleActive.MdiParent = Me  
    'Affiche la fenêtre fille  
    MDIFilleActive.Show()
```

```
End Sub
```

Comment connaître la fenêtre fille active?

Quand on en a ouvert plusieurs?

La fenêtre fille active est dans `Me.ActiveMdiChild` du conteneur

Comment voir s'il existe une fenêtre active?

```
If Not (ActiveMdiChild=Nothing) then 'elle existe
```

En mettant dans la variable **MDIFilleActive** la **fenêtre active**, on est sûr de l'avoir toujours à disposition: pour cela dans la procédure **Form1_MdiActivate** de la **fenêtre MDI** (qui se produit à chaque fois que l'on change de fenêtre fille) je récupère **Me.ActiveMdiChild** qui retourne la fenêtre fille active.

Dans Form1

```
Private Sub Form1_MdiChildActivate..  
    MDIFilleActive=Me.ActiveMdiChild  
End Sub
```



Il faut comprendre que peut importe le nom de la fenêtre fille active, on sait simplement que la fenêtre fille active est dans **MDIFilleActive**, variable que l'on utilise pour travailler sur cette fenêtre fille.

Comment avoir accès aux objets de la fenêtre fille à partir du conteneur?

De la fenêtre conteneur j'ai accès aux objets de la fenêtre fille par l'intermédiaire de la variable `MDIFilleActive` précédemment mise à jour; par exemple le texte d'un label

```
MDIFilleActive.labell1.text
```

Comment parcourir toutes les fenêtres filles?

La collection `MdiChildren` contient toutes les fenêtres filles, on peut les parcourir:

```
Dim ff As Form2
```

```
For Each ff In Me.MdiChildren
```

```
...
```

```
Next
```

Comment avoir accès aux objets du conteneur à partir de la fenêtre fille?

En utilisant `Me.MdiParent` qui contient le nom du conteneur.

Dans la fenêtre fille le code `Me.MdiParent.text = "Document 1"` affichera 'Document 1' dans la barre de titre du conteneur.

Comment une routine du module conteneur appelle une routine dans la fenêtre fille active?

Si une routine public de la fenêtre fille se nomme `Affiche`, on peut l'appeler par:

```
MDIFilleActive.Affiche()
```

Il n'est pas possible d'appeler les évènements liés aux objets.

Agencement des fenêtres filles :

La propriété `LayoutMdi` de la fenêtre conteneur modifie l'agencement des fenêtres filles.

0 - `MdiLayout.Cascade`

- 1 - `MdiLayout.TileHorizontal`
- 2 - `MdiLayout.TileVertical`
- 3 - `MdiLayout.ArrangeIcons`

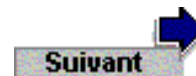
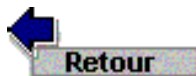
Exemple:

Le menu Item Cascade met les fenêtres filles en cascade.

```
Protected Sub CascadeWindows_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
```

```
Me.LayoutMdi(System.Windows.Forms.MdiLayout.Cascade)
```

```
End Sub
```





4.5	<h2>Travailler sur le temps: dates, heure, Timers.</h2>		
-----	---	--	--

On a vu qu'il existe un type de variable 'DateTime' pour gérer les dates et heures, comment l'utiliser ?

Nous verrons aussi comment utiliser les Timers pour déclencher des événements à intervalle régulier,

Enfin comment perdre du temps?

Une variable **DateTime** Contient une date plus l'heure.

Elle occupe 8 octets.(64 bits)

Elle peut contenir une date comprise entre le 1^{er} janvier de l'année 1 et le 31 décembre 9999 et une heure comprise entre 0:00:00 (minuit) et 23:59:59.

En fait ce qui est codé dans la variable DateTime est le nombre de graduations (Une graduation= 100 nanosecondes.) écoulées à compter de minuit, le 1er janvier de l'année 1 jusqu'a la date codée.

Nb: DateTime fait partie d'une Classe .Net , il existe aussi un type nommé Date qui contient aussi une date et l'heure et qui fait partie de VB mais qui n'est pas une classe.

Saisir une date, une heure

Pour saisir une valeur **DateTime** en utilisant un littéral: elle doit être placée entre des signes (#) et son format doit être de type d/m/yyyy, par exemple #31/5/1998#.

```
Dim DateNaissance As DateTime
```

```
DateNaissance= #02/12/1951#
```

Autre manière de saisir une date, une heure:

```
Dim date1 As New System.DateTime(1996, 6, 3, 22, 15, 0)
'Année, mois, jour, heure, minute, seconde, et
éventuellement millisecondes)
```

Afficher une date, une heure.

Pour afficher les dates et heures simplement, il suffit d'utiliser **.ToString**

```
MsgBox(DateNaissance.ToString) 'Affichera 02/12/1951
11:00:00
```

C'est le format utilisé par l'ordinateur (en fonction du pays)

ToString peut comporter des arguments qui formatent l'affichage:

Voici quelques codes de formatage:

d	affiche le jour	2
dd	affiche le jour sur 2 chiffres	02
ddd	affiche le jour abrégé	Dim.
dddd	affiche le jour complet	Dimanche

M affiche le mois 12

MM affiche le mois sur 2 chiffres 12

MMM affiche le mois abrégé déc

MMMM affiche le mois complet
décembre

y, yy, yyyy affiche 1 à 2 chiffres, deux chiffres ou quatre chiffre 51, 51, 1951

H affiche l'heure sur un ou deux chiffres (format 24h)

HH affiche l'heure sur 2 chiffres

h et hh font de même mais avec un format 12 h.

t, tt affiche l'heure en format 12h plus A ou P (pour matin, après midi)

m, mm, s, ss, f, ff font de même pour les minutes, secondes et millisecondes.

: et / sont les séparateurs heure et date.

Exemple:

```
MsgBox(DateNaissance.ToString("dddd d MMMM yyyy"))  
'Affichera Dimanche 2 décembre 1951
```

```
MsgBox(DateNaissance.ToString("hh:mm")) 'Affichera  
11:00
```

```
MsgBox(DateNaissance.ToString("d/MM/yy")) 'Affichera  
02/12/51
```

```
MsgBox(DateNaissance.ToString("%h")) 'Affichera 11 le  
caractère % est utilisé quand on affiche une seule donnée.
```

On peut enfin utiliser les méthodes de la **classe DateTime!!**

```
DateNaissance.ToLongDateString 'dimanche 02  
décembre 1951
```

```
DateNaissance.ToShortDateString '02/12/1951
```

DateNaissance.ToLongTimeString '11:00:00

DateNaissance.ToShortTimeString '11:00

Variable "temps"

Un **TimeSpan** est une unité de temps (un intervalle de temps) exprimée en **jours, heures, minutes, secondes;**

Un TimeSpan initialisé avec 1.0e+13 graduations représente "11.13:46:40", ce qui correspond à 11 jours, 13 heures, 46 minutes et 40 secondes.

L'espace de nom **System.DateTime**. contient une multitude de membre:

Add, Substrat

On peut **ajouter ou soustraire** un TimeSpan à un DateTime, on obtient un DateTime.

En clair on peut ajouter à une date une durée, on obtient une date.

' Quel sera la date dans 36 jours?.

```
Dim today As System.DateTime
Dim duration As System.TimeSpan
Dim answer As System.DateTime
```

```
today = System.DateTime.Now
duration = New System.TimeSpan(36, 0, 0, 0)
answer = today.Add(duration)
```

On peut ajouter ou soustraire 2 dates, on obtient une TimeSpan

```
Dim diff1 As System.TimeSpan
diff1 = date2.Subtract(date1)
```

AddDay, AddMonths, AddHours, AddSeconds, AddMilliseconds

Permet d'ajouter des jours, des mois, des heures, des secondes, ou des millisecondes à une date, on obtient une

date.

```
Answer=today.AddDays(36)
```

Year, Mouth, Day, Hour, Minute, Seconde, Millisecond

Permettent d'extraire l'année, le mois, le jour, l'heure, les minutes, les secondes, les millisecondes d'une date:

```
I=DateNaissance.Year ' => I=1951
```

```
I=System.DateTime.Now.Day 'donne le jour d'aujourd'hui (1 à 31)
```

DayOfWeek

Retourne le jour de la semaine (0 pour dimanche à 6 pour samedi)

```
I=DateNaissance.DayOfWeek 'I=0 car le 02/12/1951 est un dimanche.
```

`DayForYear` existe aussi.

Now, Today, TimeOfDay

`Now` est la date et l'heure du système. (Là, maintenant)

`Today` est la date du système avec l'heure à 0.

`TimeOfDay` est l'heure actuelle.

Ticks

Donne le nombre de graduations d'un `DateTime`.

`AddTicks` peut être utilisé.

Comparaison de DateTime

On utilise `Compare: DateTime.Compare(t1, t2)` retourne 0 si $t1=t2$, une valeur positive si $t1>t2$ négative si $t1<t2$.

```

Dim t1 As New DateTime(100)
Dim t2 As New DateTime(20)

If DateTime.Compare(t1, t2) > 0 Then
    Console.WriteLine("t1 > t2")
End If
If DateTime.Compare(t1, t2) = 0 Then
    Console.WriteLine("t1 = t2")
End If
If DateTime.Compare(t1, t2) < 0 Then
    Console.WriteLine("t1 < t2")
End If

```

On peut aussi utiliser la méthode `op_Equality` de l'espace de nom pour voir si 2 dates sont égales:

```
areEqual = System.DateTime.op_Equality(april19, otherDate)
```

Il existe aussi `op_GreaterThan` et beaucoup d'autres.

Comment saisir rapidement une date dans un programme?

En ajoutant à une fenêtre un contrôle **DateTimePicker**

En mode Run , il apparaît une zone rectangulaire avec la date système dedans:



Si l'utilisateur clique sur la flèche déroulante, il apparaît une fenêtre calendrier.



Il suffit pour l'utilisateur de cliquer sur la bonne date.

Le programmeur récupère la date dans
[DateTimePicker1.value](#)

Il existe, bien sur, de multiples propriétés et plusieurs évènements, le plus remarquable étant: [ValueChanged](#).

MonthCalendar est un contrôle similaire mais qui reste toujours ouvert.

De plus grâce à **CalendarDimension** on peut afficher plusieurs mois.

Les Timers.

Pour déclencher un évènement à intervalle régulier, il faut utiliser les **minuteries** ou '**Timer**'.

Prendre le contrôle **Timer** dans le Boite à outils, l'ajouter à la fenêtre. Il apparaît en bas sous la fenêtre dans la barre d'état des composants.

Il n'apparaît pas à l'utilisateur dans la fenêtre en mode Run.

Il est très simple à utiliser.

La propriété [Interval](#) contient la périodicité de l'évènement **Ticks**, évènement qui se déclenche régulièrement.

[Interval](#) est en millisecondes. Pour [Interval=500](#) l'évènement Ticks se déclenche toutes les 1/2 secondes.

Start et **Stop** déclenche et arrête la minuterie. (De même [Enabled](#) active ou non)

Exemple:

Faire clignoter un label toutes les 1/2 secondes.

Créer le label1

Ajouter un Timer1 (qui se place en bas sous la fenêtre)

```
Private Sub Form3_Load(...)
    Timer1.Interval = 500
    Timer1.Start()
End Sub
```

```
Private Sub Timer1_Tick(..)
    Label1.Visible = Not (Label1.Visible)
End Sub
```

Un évènement Timer_Tick se produit toutes les 1/2 secondes et inverse la valeur de la propriété visible du label. (Si elle était égale à True, elle devient égale à False et vice versa.)

Mais attention: Timer à des restrictions de taille:

- Si votre application ou une autre demande beaucoup au système (boucles longues, calculs complexes, accès intensifs à un périphérique, un réseau ou un port, par exemple), les événements de minuterie peuvent être moins fréquents que spécifié dans la propriété **Interval**. Il n'est pas garanti que l'intervalle s'écoule dans le temps exact!!
- L'intervalle peut être compris entre 1 et 64 767 millisecondes: l'intervalle le plus long ne dépasse pas de beaucoup la minute (64,8 secondes).
- Le système génère 18 graduations à la seconde (même si la valeur de la propriété **Interval** est mesurée en millisecondes, la véritable précision d'un intervalle ne dépassera pas un dix-huitième de seconde).

Donc pour faire clignoter un label :OUI

Pour compter précisément un intervalle de temps:NON

Mais il y a d'[autres méthodes](#).

Perdre du temps:

Parfois on a besoin de perdre du temps:

Exemple ne rien faire pendant 3 secondes puis poursuivre..

- **Il est exclu de faire des boucles vides:**

```
For i=0 to 100000 ' le temps écoulé est variable en fonction des machines..
```

```
Next i
```

- Autre méthode : on boucle tant que l'heure courante est inférieure à l'heure du départ+3s

```
Dim t As DateTime=DateTime.Now
```

```
Do While DateTime.Now <t.AddSeconds(3)
```


Loop

Mais cela accapare le processeur.

- On peut utiliser un Timer et vérifier dans la procédure Tick si le temps est écoulé.
- On peut utiliser Thread.Sleep

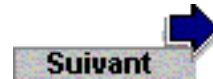
```
System.Threading.Thread.Sleep(3000)
```

Chronométrer:

Parfois on a besoin de chronométrer un évènement:

Voir la rubrique [Chronométrer 7-4](#)

L'exemple sur [l'horloge](#) est aussi didactique.





4.6

Les fichiers.

Suivant

Sommaire

Comment lire et écrire dans des fichiers du texte, des octets, du XML du Rtf ?

Généralités et rappels:

Le mot '**fichier**' est à prendre au sens informatique: ce n'est pas un ensemble de fiches mais plutôt un ensemble d'octets. Un fichier peut être un programme (Extension .EXE), du texte (Extension .TXT ou .DOC....), une image (Extension .BMP .GIF .JPG...), une base de données (.MDB..) du son, de la vidéo....

Pour travailler avec du texte, des octets, des données très simple (sans nécessité d'index, de classement..), on utilise les méthodes décrites dans cette page: travail direct dans **les fichiers séquentiels, aléatoires, binaires**. Mais dès que les informations sont plus structurées, il faut utiliser les **bases de données** (Il y a plusieurs chapitre plus loin traitant des Base de données).

Un fichier a un **nom**: 'Image.GIF' , une **extension**: '.GIF' qui en indique généralement le type de contenu , des **attributs** (Longueur, Date de création, de modification, Fichier en lecture seule ou non..).

On voit cela dans l'explorer Windows:

Nom	Taille	Type	Date de modification
2035.gif	10 Ko	Image GIF	25/11/2002 18:55
accueil.gif	1 Ko	Image GIF	27/12/2001 23:42
ampoule.gif	2 Ko	Image GIF	11/04/2002 22:34

Un fichier est composé d'**enregistrements** qui sont des 'paquets' de données; suivant le type de fichier un enregistrement peut correspondre à une ligne, un octet, un groupe d'octets..

Comment utiliser les fichiers? Voici le plan de cet article:

A- Il est conseillé de travailler avec les Classes du Frameworks

Avec la Classe FileInfo, on obtient des **renseignements** sur le fichier.

Pour **lire écrire** dans un fichier (en dehors des bases de données), il y a plusieurs méthodes:

**Avec la Classe System.IO on a a notre disposition
StreamReader StreamWriter BinaryReader BinaryWriter
FileStream:**

Pour lire ou écrire dans un fichier, il faut l'**ouvrir (Open)**, **lire ou écrire en utilisant un flux de données (Stream)** puis le **refermer (Close)**.

Le **Stream** (flux, torrent, courant) est une notion générale, c'est donc un flux de données provenant ou allant vers un fichier, un port, une connexion TCP/IP...

L'accès est séquentiel: les données sont traitées du début à la fin du fichier.

B- Il existe toujours la méthode classique du FileOpen:

On ouvre le fichier en mode séquentiel, aléatoire, binaire, on lit X enregistrements, on referme le fichier.

C- Avec certains objets, on gèrent automatiquement les lectures écritures sur disque.

Comme avec le RichTextBox par exemple.

En résumé, pour travailler sur les fichiers, on dispose:

- des instructions VB runtime traditionnelles: FileOpen WriteLine..
- des instructions du FSO (FileObjetSystem) pour la compatibilité avec les langages de script.
- de l'espace de nom System.IO avec les Class et objets .NET

Les 2 premiers font appel au troisième; donc pourquoi ne pas utiliser directement les Classe .NET?

A- Classe FileInfo et File, Stream.

Pour travailler sur les fichiers, il faut au préalable taper:

```
Imports System.IO
```

La classe **File** est utilisée pour travailler sur un ensemble de fichier ou un fichier (sans instanciation préalable), la Classe **FileInfo** donne des renseignements sur un fichier particulier (Il faut instancer au préalable un objet FileInfo).

La Classe **File** possède les méthodes suivantes.

- | | |
|--------------------------------------|-------------------------------|
| Exists | Teste si le fichier existe. |
| Create | Crée le fichier |
| Copy | Copie le fichier |
| Delete | Efface le fichier |
| GetAttributes , SetAttributes | Lire ou écrire les attributs. |
| Move | Déplacement de fichier |

Toutes les méthodes **Open** (pour un FileStream) **OpenRead**, **OpenWrite**, **OpenText**.

Exemple:

Un fichier existe-t-il? Afficher True s'il existe:

```
Label1.Text = File.Exists("vessaggi.gif").ToString
```

La Classe **FileInfo** possède les propriétés suivantes.

Name	Nom du fichier (sans extension)
FullName	Nom complet avec extension
Extension	Extension (.txt par exemple)
Length	Longueur du fichier.
Directory	Répertoire parent
DirectoryName	Répertoire ou se trouve le fichier
Exists	
LastAccessTime	Date du dernier accès, LastWriteTime existe aussi.
Attributes	Attributs

Il faut faire un **AND** entre **Attributes** et une valeur de l'énumération **FileAttributes** (Archive, Compressed, Directory, Encrypted, Hidden, Normal, ReadOnly, System, Temporal).

Pour tester **ReadOnly** par exemple:

Fi.Attributes And FileAttributes.ReadOnly Retourne True si le fichier est **ReadOnly**

Et les méthodes suivantes:

Create, Delete, MoveTo

AppendText, CopyTo Open, OpenRead, OpenWrite, OpenText..

On voit que toutes les informations sont accessibles.

Exemple:

Pour un fichier, afficher successivement le nom, le nom avec répertoire, le répertoire, la longueur, la date de dernière écriture et si le fichier est en **ReadOnly.**

```
Dim sNom As String = "c:\monfichier.txt"  
Dim Fi As FileInfo  
Fi=New FileInfo( sNom)
```

```
MsgBox("Nom=" & Fi.Name)
MsgBox("Nom complet =" & Fi.FullName)
MsgBox("Répertoire=" & Fi.DirectoryName)
MsgBox("Longueur=" & Fi.Length.ToString)
MsgBox("Date der modification=" &
Fi.LastWriteTime.ToShortDateString)
MsgBox("ReadOnly=" & (Fi.Attributes And
FileAttributes.ReadOnly).ToString)
```

Utiliser les "Stream".

Le **Stream** (flux, torrent, courant) est une notion générale, c'est donc un flux de données provenant ou allant vers un fichier, un port, une connexion TCP/IP...

Ici on utilise un **Stream** pour lire ou écrire dans un fichier.

L'accès est séquentiel: les données sont traitées du début à la fin du fichier.

Pour écrire dans un fichier texte:

Il faut instancer un objet de la classe **StreamWriter** . On écrit avec **Write** ou **WriteLine**.(ajoute un saut de ligne) Enfin on ferme avec **Close**.

On peut instancer avec le constructeur de la classe StreamWriter et avec New, ou par la Classe File.

```
Dim SW As New StreamWriter ("MonFichier.txt") ' crée ou si existe écrase
```

Il existe une surcharge permettant d'ajouter à la fin du fichier:

```
Dim SW As New StreamWriter ("MonFichier.txt", True) ' crée ou si existe ajoute
```

Avec la classe File:

```
Dim SW As StreamWriter=File.CreateText ("MonFichier.txt") ' crée ou si existe écrase
```

```
Dim SW As StreamWriter = File.AppendText("MonFichier.txt") ' crée ou si existe ajoute
```

Ensuite pour écrire 2 lignes:

```
SW.WriteLine ("Bonjour")
```

SW.WriteLine ("Monsieur")

Enfin on ferme:

SW.Close()

Pour lire dans un fichier Texte:

Il faut instancier un objet de la classe **StreamReader**. On lit avec **Read** (un nombre d'octet) **ReadLine** (une ligne) **ReadToEnd** (de la position courante jusqu'à la fin). Enfin on ferme avec **Close**.

Avec le constructeur de la Classe Stream Reader:

```
Dim SR As New StreamReader ("MonFichier.txt")
```

Avec la Classe File:

```
Dim SR As StreamReader=File.OpenText ("MonFichier.txt")
```

Comment lire chaque ligne du fichier et s'arrêter à la fin?

En effet on ne sait pas habituellement combien le fichier contient de ligne, si le fichier contient 2 lignes il faut en lire 2 et s'arrêter sinon on tente de lire après la fin du fichier et cela déclenche une erreur.

3 solutions:

1-Utiliser **ReadToEnd** qui lit en bloc jusqu'à la fin.

2-Avant **ReadLine** mettre un **Try**: quand l'erreur 'fin de fichier' survient elle est interceptée par **Catch** qui sort du cycle de lecture et ferme le fichier.

3-Utiliser **Peek** qui lit dans le fichier un caractère mais sans modifier la position courante de lecture.

La particularité de **Peek** est de retourner -1 s'il n'y a plus de caractère à lire sans déclencher d'erreur, d'exception.

La troisième solution est la plus générale et la plus élégante:

```
Do Until SR.Peek=-1
```

```
    Ligne=SR.ReadLine()
```

```
Loop
```

Enfin on ferme:

SR.Close()

Notion de 'Buffer', utilisation de Flush.

En fait quand on écrit des informations sur le disque, le logiciel travaille sur un buffer ou **mémoire tampon** qui est en mémoire vive. Si on écrit des lignes dans le fichier, elles sont 'écrites' dans le buffer en mémoire vive. Quand le buffer est plein, (ou que l'on ferme le fichier) l'enregistrement du contenu du buffer est effectué effectivement sur le disque.

Ce procédé est général à l'écriture et à la lecture de fichier mais totalement transparent car le programmeur ne se préoccupe pas des buffers.

Parfois, par contre, même si on a enregistré peu d'information, on veut être sûr qu'elle est sur le disque, il faut donc forcer l'enregistrement sur disque même si le buffer n'est pas plein, on utilise alors la méthode **Flush**.

SW.Flush()

Le fait de fermer un fichier par Close, appelle automatiquement Flush() ce qui enregistre des données du buffer.

B- Utiliser "FileOpen".

Visual Basic fournit **trois types d'accès au fichier** :

- **l'accès séquentiel**, pour lire et écrire des fichiers 'texte' de manière continue, chaque donnée est enregistrée successivement du début à la fin ; les enregistrements n'ont pas la même longueur, ils sont séparés par un séparateur (des virgules ou des retours à la ligne).

Philippe

Jean-

François

Louis

On ne peut qu'écrire le premier enregistrement puis le second, le troisième, le quatrième...

Pour lire c'est pareil: on ouvre , on lit le premier, le second, le troisième, le quatrième....

Pour lire le troisième enregistrement , il faut lire avant les 2 premiers.

- q **l'accès aléatoire (Random)**, (on le nomme parfois **accès direct**) pour lire et écrire des fichiers texte ou binaire constitués d'enregistrements *de longueur fixe* ; on peut avoir directement accès à un enregistrement à partir de son numéro.

Philippe 1 place de la gare
Jean 35 rue du cloître
Pierre 14 impasse du musée
Louis sdf

Les enregistrements ont une longueur fixe: il faut prévoir!! si on décide de 20 caractères pour le prénom, on ne pourra pas en mettre 21, le 21ème sera tronqué, à l'inverse l'enregistrement de 15 caractères sera complété par des blancs.

Il n'y a pas de séparateur entre les enregistrements.

Les enregistrements peuvent être constitués d'un ensemble de variables: une structure, ici prénom et adresse.

Ensuite on peut lire directement n'importe quel enregistrement, le second enregistrement par exemple, ou écrire sur le 3ème.

- q **l'accès binaire**, pour lire et écrire dans tous les fichiers , on lit ou écrit un nombre d'octet désiré..

En pratique:

Les fichiers séquentiels sont bien pratique pour charger une série de ligne, (toujours la même) dans une ListBox par exemple.

Faut-il utiliser les fichiers séquentiels ou random (à accès

aléatoire) pour créer par exemple un petit carnet d'adresse?

Il y a 2 manières de faire:

- Créer un fichier random et lire ou écrire dans un enregistrement pour lire ou modifier une adresse.
- Créer un fichier séquentiel. A l'ouverture du logiciel lire séquentiellement toutes les adresses et les mettre dans un tableau (de structure). Pour lire ou modifier une adresse: lire ou modifier un élément du tableau. En sortant du programme enregistrer tous les éléments du tableau séquentiellement. (Enregistrer dans un nouveau fichier, effacer l'ancien, renommer le nouveau avec le nom de l'ancien).
- Bien sur s'il y a de nombreux éléments dans une adresse, un grand nombre d'adresse, il faut utiliser une base de données.



Si on ouvre un fichier en écriture et qu'il n'existe pas sur le disque, il est créé.

Si on ouvre un fichier en lecture et qu'il n'existe pas, une exception est déclenchée (une erreur). On utilisait cela pour voir si un fichier existait: on l'ouvrait, s'il n'y avait pas d'erreur c'est qu'il existait. Mais maintenant il y a plus simple pour voir si un fichier existe.

Si on ouvre un fichier et que celui-ci est déjà ouvert par un autre programme, il se déclenche généralement une erreur (sauf si on l'ouvre en Binaire, c'était le cas en VB6, c'est à vérifier en VB.NET).

Pour ouvrir un fichier on utilise **FileOpen**

FileOpen (FileNumber, FileName, Mode, Access, Share, RecordLength)

Paramètres de FileOpen

FileNumber

A tous fichier est affecté un numéro unique, c'est ce numéro que l'on utilisera pour indiquer sur quel fichier pratiquer une opération.. Utilisez la fonction **FreeFile** pour obtenir le prochain numéro de fichier disponible.

FileName

Obligatoire. Expression de type **String** spécifiant un nom de fichier. Peut comprendre un nom de répertoire ou de dossier, et un nom de lecteur.

Mode

Obligatoire. Énumération **OpenMode** spécifiant le mode d'accès au fichier : **Append**, **Binary**, **Input (séquentiel en lecture)**, **Output (séquentiel en écriture)** ou **Random (accès aléatoire)**.

Access

Facultatif. Mot clé spécifiant les opérations autorisées sur le fichier ouvert : **Read**, **Write** ou **ReadWrite**. Par défaut, la valeur est **OpenAccess.ReadWrite**.

Share

Facultatif. Spécifiant si un autre programme peut avoir en même temps accès au même fichier : **Shared (permet l'accès aux autres programmes)**, **Lock Read (interdit l'accès en lecture)**, **Lock Write (interdit l'accès en écriture)** et **Lock Read Write (interdit totalement l'accès)**. Le processus **OpenShare.Lock Read Write** est paramétré par défaut.

RecordLength

Facultatif. Nombre inférieur ou égal à 32 767 (octets). Pour les fichiers ouverts en mode **Random**, cette valeur représente la longueur de l'enregistrement. Pour les fichiers séquentiels, elle représente le nombre de caractères contenus dans la mémoire tampon.

Pour **écrire** dans un fichier on utilise

Print , **Write**, **WriteLine**. dans les fichiers séquentiels

FilePut dans les fichiers aléatoires

Pour **lire** dans un fichier on utilise:

Input, **LineInput** dans les fichiers séquentiels

FileGet dans les fichiers aléatoires.

Pour **fermer** le fichier on utilise **FileClose()**

Numéro de fichier:

Pour repérer chaque fichier, on lui donne un numéro unique (de type Integer).

La fonction **FreeFile** retourne le premier numéro libre.

Dim No as integer

No= Freefile()

Ensuite on peut utiliser **No**

FileOpen(No, "MonFichier", OpenMode.Output)

Print(**No**, "toto")

FileClose (**No**)

Fichier séquentiel:

Vous devez spécifier si vous voulez lire (entrer) des caractères issus du fichier (mode **Input**), écrire (sortir) des caractères vers le fichier (mode **Output**) ou ajouter des caractères au fichier (mode **Append**).

Ouvrir le fichier 'MonFichier' en mode séquentiel pour y écrire:

Dim No as integer

No= Freefile

FileOpen(No, "MonFichier", OpenMode.Output)

Pour écrire dans le fichier séquentiel: on utilise **Write** ou **WriteLine** **Print** ou **PrintLine**:

- La fonction **Print** écrit dans le fichier sans aucun caractère de séparation.

```
Print(1,"toto")
```

```
Print(1,"tata")
```

```
Print(1, 1.2)
```

Donne le fichier 'totototal.2'

- La fonction **Write** insère des points-virgules entre les éléments et des guillemets de part et d'autre des chaînes au moment de leur écriture dans le fichier, les valeurs booléens et les variables DateTime sont écrites sans problèmes.

```
Write(1,"toto")
```

```
Write(1,"tata")
```

```
Write(1, 1.2)
```

Donne le fichier '"toto";"tata";1.2"

Attention s'il y a des points-virgules dans les chaînes , elles seront considérées comme séparateurs!! ce qui entraîne des erreurs à la lecture; il faut mettre la chaîne entre "" ou bien remplacer le point-virgule par un caractère non utilisé (# par exemple) avant de l'enregistrer puis après la lecture remplacer '#' par ';'.

Il faut utiliser **Input** pour relire ces données (Input utilise aussi le point-virgule comme séparateur.

- ¶ La fonction **WriteLine** insère un caractère de passage à la ligne, c'est-à-dire un retour chariot+ saut de ligne (Chr(13) + Chr(10)), On lira les données par **LineInput**.

```
WriteLine(1,"toto")
```

```
WriteLine(1,"tata")
```

```
WriteLine(1, 1.2)
```

Donne le fichier

```
"toto"
```

```
"tata"
```

```
1.2
```

Il faut utiliser **LineInput** pour relire ces données car il lit jusqu'au retour Chariot, saut de ligne.

Toutes les données écrites dans le fichier à l'aide de la fonction Print respectent les conventions internationales ; autrement dit, les données sont mises en forme à l'aide du séparateur décimal approprié. Si l'utilisateur souhaite produire des données en vue d'une utilisation par plusieurs paramètres régionaux, il convient d'utiliser la fonction **Write**

EOF (NuméroFichier) veut dire 'End Of File', (Fin de Fichier) il prend la valeur True si on est à la fin du fichier et qu'il n'y a plus rien à lire.

LOF (NuméroFichier) veut dire 'Lenght Of File', il retourne la longueur du fichier.

Exemple: Lire chaque ligne d'un fichier texte.

```
Dim Line As String
```

```
FileOpen(1, "MonFichier.txt", OpenMode.Input) 
```

```
Ouvre en lecture.
```

```
While Not EOF(1) ' Boucler jusqu'à la fin du  
fichier
```

```
Line = LineInput(1) ' Lire chaque ligne  
Debug.WriteLine(Line) ' Afficher chaque  
ligne sur la console.
```

```
End While
FileClose(1) Fermer.
```

Ici on a utilisé une boucle While.. End While qui tourne tant que EOF est Faux. Quand on est à la fin du fichier EOF (End of File) devient égal à True et on sort de la boucle.

Fichier à accès aléatoire:

On ouvre le fichier avec **FileOpen** et le mode **OpenMode.Random**, ensuite on peut écrire un enregistrement grâce à **FilePut()** ou en lire un grâce à **FileGet()**. On peut se positionner sur un enregistrement précis (le 2eme, le 15ème) avec **Seek**.

Le premier enregistrement est l'enregistrement numéro 1

Exemple:

Fichier des adresses

Créer une structure Adresse, on utilise <VBFixedString()> pour fixer la longueur.

```
Public Structure Adresse
    <VBFixedString(20)>Dim Nom           As String
    <VBFixedString(20)>Dim Rue           As String
    <VBFixedString(20)>Dim Ville        As String
End Structure
```

'Ouvrir le fichier, comme il n'existe pas, cela entraîne sa création

```
Dim FileNum As Integer, RecLength As Long, UneAdresse As Adresse
```

' Calcul de la longueur de l'enregistrement

```
RecLength = Len(UneAdresse)
```

' Récupérer le premier numéro de fichier libre.

```
FileNum = FreeFile
```

' Ouvrir le fichier.

```
FileOpen(FileNum, "MONFICHIER.DAT", OpenMode.Random, , , RecLength)
```

Pour **écrire** des données sur le second enregistrement par exemple:

```
UneAdresse.Nom = "Philippe"
```

```
UneAdresse.Rue = "Grande rue"
```

```
UneAdresse.Ville = "Lyon"
```

```
FilePut(FileNum, UneAdresse, 2 )
```

Dans cette ligne de code, `FileNum` contient le numéro utilisé par la fonction `FileOpen` pour ouvrir le fichier, `2` est le numéro de l'enregistrement ou sera copié la variable `'UneAdresse'` (c'est un long si on utilise une variable) et `UneAdresse`, déclaré en tant que type `Adresse` défini par l'utilisateur, reçoit le contenu de l'enregistrement. Cela écrase l'enregistrement 2 s'il contenait quelque chose.

Pour **écrire à la fin** du fichier, **ajouter un enregistrement** il faut connaître le nombre d'enregistrement et écrire l'enregistrement suivant.

```
Dim last as long 'noter que le numéro d'enregistrement est un long
```

Pour connaître le nombre d'enregistrement, il faut diviser la longueur du fichier par la longueur d'un enregistrement.

```
last = FileLen("MONFICHIER.DAT") / RecLength
```

On ajoute 1 pour créer un nouvel enregistrement.

```
FilePut(FileNum, UneAdresse, last+1 )
```

Pour **lire** un enregistrement (le premier par exemple):

```
FileGet(FileNum, UneAdresse, 1)
```

Attention Option Strict doit être à `false` .

Si option Strict est à `True`, la ligne qui précède génère une erreur car le second argument attendu ne peut pas être une variable `'structure'`. Pour que le second argument de `FileGet` (Une adresse) soit converti dans une variable `Structure` automatiquement Option Strict doit donc être à `false`. (Il doit bien y avoir un moyen de travailler avec Option Strict On et de convertir explicitement mais je ne l'ai pas trouvé)

Remarque: si le fichier contient 4 enregistrements, on peut écrire le 10ème enregistrement, VB ajoute entre le 4ème et le 10ème, 5 enregistrements vides. On peut lire un enregistrement qui n'existe pas, cela ne déclenche pas d'erreur.

Le numéro d'enregistrement peut être omis dans ce cas c'est l'enregistrement courant qui est utilisé.

On positionne l'enregistrement courant avec **Seek:**

Exemple: Lire le 8ème enregistrement:

`Seek(FileNum,8)`

`FileGet(FileNum,Une Adresse)`

Suppression d'enregistrements

Vous pouvez supprimer le contenu d'un enregistrement en effaçant ses champs (enregistrer à la même position des variables vides), mais l'enregistrement existe toujours dans le fichier.

Pour enlever un enregistrement supprimé

1. Créez un nouveau fichier.
2. Copiez tous les enregistrements valides du fichier d'origine dans le nouveau fichier (pas ceux qui sont vides).
3. Fermez le fichier d'origine et utilisez la fonction **Kill** pour le supprimer.
4. Utilisez la fonction **Rename** pour renommer le nouveau fichier en lui attribuant le nom du fichier d'origine.

Fichier binaire:

Dans les fichiers binaires on travaille sur les octets.

La syntaxe est la même que pour les fichiers Random, sauf qu'on travaille sur la position d'un octet et non sur un numéro d'enregistrement.

Pour ouvrir un fichier binaire:

`FileOpen(FileNumber, FileName, OpenMode.Binary)`

FileGet et **FilePut** permettent de lire ou d'écrire des octets .

```
FileOpen(iFr, ReadString, OpenMode.Binary)
MyString = New String(" "c, 15)           'Créer une chaîne de
15 espaces
FileGet(iFr, MyString)                     ' Lire 15 caractères
dans MyString
FileClose(iFr)
MsgBox(MyString)
```

Le fait de créer une variable de 15 caractères et de l'utiliser dans FileGet permet de lire 15 caractères.

C-Utilisation du Contrôle RichTextBox.

On rappelle que du texte présent dans un contrôle **RichTextBox** peut être enregistré ou lu très simplement avec les méthodes **.SaveFile** et **.LoadFile**.

Le texte peut être du **texte brut** ou du **RTF**.

```
richTextBox1.SaveFile(FileName,  
RichTextBoxStreamType.PlainText)
```

Si on remplace `.PlainText` par **.RichText** c'est le texte enrichi et non le texte brut qui est enregistré

Pour lire un fichier il faut employer **.LoadFile** avec la même syntaxe.

Simple, non!!!

Lire ou écrire des octets ou du XML:

`BinaryWriter` et `BinaryReader` permettent d'écrire ou de lire des données binaires.

`XMLTextWriter` et `XMLTextReader` écrit et lit du Xml.



Retour



Index



Sommaire



Suivant

Site **LDF** Cours **VB.net**



4.7	<h2>Travailler sur les répertoires.</h2>		Suivant	Sommaire
-----	--	--	-------------------------	--------------------------

Comment créer, copier effacer des répertoires (ou dossiers)?

Classe DirectoryInfo et la Classe Directory

Pour travailler sur les dossiers (ou répertoires), il faut au préalable taper:

```
Imports System.IO
```

La classe **Directory** est utilisée pour travailler sur un ensemble de dossier, la Classe **directoryInfo** donne des renseignements sur un dossier particulier (Après instantiation).

La Classe **Directory** possède les méthodes suivantes.

Exists	Teste si le dossier existe.
CreateDirectory	Crée le dossier
Delete	Efface le dossier
Move	Déplacement de dossier
GetCurrentDirectory	Retourne le dossier de travail de l'application en cours
SetCurrentDirectory	Définit le dossier de travail de l'application.
GetDirectoryRoot	Retourne le dossier racine du chemin spécifié.
GetDirectories	Retourne le tableau des sous dossiers du dossier spécifié.
GetFiles	Retourne les fichiers du dossier spécifié.

GetFileSystemEntries Retourne fichier et sous dossier avec possibilité d'un filtre.

GetLogicalDrives Retourne les disques

GetParent Retourne le dossier parent du dossier spécifié.

La Classe Directory est **statique** : on l'utilise directement.

Exemple:

Afficher dans une listBox les sous dossiers du répertoire de l'application:

```
Dim SousDos() As String=  
Directory.GetDirectories(Directory.GetCurrentDirectory)  
Dim Dossier As String  
For Each Dossier In SousDos  
    List1.Items.Add(Dossier)  
Next
```

La Classe **DirectoryInfo** possède les propriétés suivantes.

Name Nom du dossier (sans extension)

Full Name Chemin et nom du dossier

Exists

Parents Dossier parent

Root Racine du dossier

La Classe DirectoryInfo n'est **pas statique** : il faut instancer un dossier avant de l'utiliser.

Il y a aussi les méthodes suivantes:

Create, Delete, MoveTo

CreateSubdirectory

GetDirectories Retourne les sous-dossiers

GetFiles Retourne des fichiers

GetFileSystemInfos

Exemple:

Afficher le répertoire parent d'un dossier:

```
Dim D As DirectoryInfo  
D= New DirectoryInfo( MonDossier)  
MsgBox(D.Parent.ToString)
```

Classe Path

La Classe statique **Path** a des méthodes simplifiant la manipulation des répertoires :

Exemple :

Si `C = "C:\Windows\MonFichier.txt"`

`Path.GetDirectoryName(C)` retourne `"C:\Windows"`

`Path.GetFileName(C)` retourne `"Monfichier.txt"`

`Path.GetExtension(C)` retourne `".txt"`

`Path.GetFileNameWithoutExtension(C)` retourne `"MonFichier"`

`Path.PathRoot(C)` retourne `"c:\"`

Il y a aussi les méthodes **ChangeExtension, Combine, HasExtension ...**

Classe Environment

Donne des informations concernant l'environnement et la plate-forme en cours ainsi que des moyens pour les manipuler. Par exemple: les arguments de la ligne de commande, le code de sortie, les paramètres des variables d'environnement, le contenu de la pile des appels, le temps écoulé depuis le dernier démarrage du système ou le numéro de version du Common Language Runtime **mais aussi certains répertoires** .

`Environment.CurrentDirectory` 'donne le répertoire courant : ou le processus en cours démarre.

`Environment.MachineName` 'Obtient le nom NetBIOS de l'ordinateur local.

`Environment.OsVersion` 'Obtient un objet contenant l'identificateur et le numéro de version de la plate-forme en cours.

`Environment.SystemDirectory` 'Obtient le chemin qualifié complet du répertoire du système

`Environment.UserName` 'Obtient le nom d'utilisateur de la personne qui a lancé le thread en cours.

La fonction `GetFolderPath` avec un argument faisant partie de l'énumération `SpecialFolder` retourne le répertoire d'un tas de choses :

Exemple: Quel est le répertoire Système?

`Environment.GetFolderPath(Environment.SpecialFolder.System)`

Comment récupérer le nom des disques?

```
Dim drives As String() = Environment.GetLogicalDrives()
```

Comment récupérer la ligne de commande?

```
Dim arguments As String() = Environment.GetCommandLineArgs()
```

On peut aussi utiliser les anciennes méthodes

VB:

CurDir() retourne le chemin d'accès en cours.

```
MyPath = CurDir()  
MyPath = CurDir("C:c")
```

Dir()

Retourne une chaîne représentant le nom d'un fichier, d'un répertoire ou d'un dossier qui correspond à un modèle ou un attribut de fichier spécifié ou à l'étiquette de volume d'un lecteur.

'vérifier si un fichier existe:

' Retourne "WIN.INI" si il existe.

```
MyFile = Dir("C:\WINDOWS\WIN.INI")
```

' Retourne le fichier spécifié par l'extension .

```
MyFile = Dir("C:\WINDOWS\*.INI")
```

'Un nouveau Dir retourne le fichier suivant

```
MyFile = Dir()
```

' On peut surcharger avec un attribut qui sert de filtre .

```
MyFile = Dir("*.TXT", vbHidden) ' affiche les fichiers cachés
```

' Recherche les sous répertoires.

```
MyPath = "c:\" ' Set the path.
```

```
MyName = Dir(MyPath, vbDirectory)
```

```
□
```

ChDrive change le lecteur actif. La fonction lève une exception si le lecteur n'existe pas.

```
ChDrive("D") □
```

Mkdir crée un répertoire ou un dossier. Si aucun lecteur n'est spécifié, le nouveau répertoire ou dossier est créé sur le lecteur actif.

[MkDir\("C:\MYDIR"\)](#)

Rmdir enlève un répertoire ou un dossier existant.
' Vérifier que le répertoire est vide sinon effacer les
fichier avec **Kill**.

[Rmdir \("MYDIR"\)](#)

ChDir change le répertoire par défaut mais pas le lecteur par
défaut.

[ChDir\("D:\TMP"\)](#)

L'exécution de changements relatifs de répertoire s'effectue
à l'aide de "..", comme suit :

[ChDir\(".."\)](#) ' Remonte au répertoire parent.

FileCopy

Copier un fichier.

[FileCopy\(SourceFile, DestinationFile\)](#)

Rename

Renommer un fichier, un répertoire ou un dossier.

[Rename \(OldName, NewName\)](#)

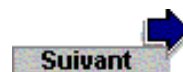
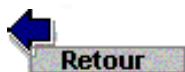
FileLen donne la longueur du fichier, **SetAttr** et **GetAttr**
modifie ou lit les attributs du fichier

[Result = GetAttr\(FName\)](#)

Result est une combinaison des attributs. Pour déterminer les attributs
définis, utilisez l'opérateur And pour effectuer une comparaison
d'opérations de bits entre la valeur retournée par la fonction GetAttr
et la valeur de l'attribut. Si le résultat est différent de zéro, cet
attribut est défini pour le fichier désigné. Par exemple, la valeur de
retour de l'expression And suivante est zéro si l'attribut Archive
n'est pas défini :

[Result = GetAttr\(FName\) And vbArchive](#)

□





4.8	Afficher correctement.	Suivant	Sommaire
------------	-----------------------------------	----------------	-----------------

Comment afficher du texte , du numérique suivant le format désiré ?

On a vu que pour afficher du texte il fallait l'affecter à la propriété 'Text' d'un label ou d'un textBox (ou pour des tests l'afficher sur la fenêtre 'console').

Pas de problème pour afficher des chaînes de caractères, par contre, pour les valeurs numériques, il faut d'abord les transformer en 'String' et les formater (définir les séparateurs, le nombre de décimales...) .

ToString

On a déjà vu que pour afficher une variable numérique, il fallait la transformer en 'string' de la manière suivant:

```
MyDouble.ToString
```

Mais ToString peut être surchargé par un paramètre appelé **chaîne de format**. Cette chaîne de format peut être standard ou personnalisée.

• Chaîne de format standard:

Cette chaîne est de la forme 'Axx' ou A donne le type de format et xx le nombre de chiffre après la virgule. Le format est défini par la 'culture' en cours sur le thread courant.

```
Imports System
```

```
Imports System.Globalization
```

```
Imports System.Threading
```

```
Module Module1
```

```
Sub Main()
```

```
Thread.CurrentThread.CurrentCulture = New CultureInfo("en-us")'changement  
de culture
```

```
Dim UnDouble As Double = 123456789
```

```
Console.WriteLine("Cet exemple est en-US culture:")
```

```
Console.WriteLine(UnDouble.ToString("C")) 'format monétaire (C) affiche  
$123,456,789.00
```

```
Console.WriteLine(UnDouble.ToString("E")) 'format scientifique (E)  
affiche 1.234568E+008
```

<code>Console.WriteLine(UnDouble.ToString("P"))</code>	'format % (P)
affiche 12,345,678,900.00%	
<code>Console.WriteLine(UnDouble.ToString("N"))</code>	'format nombre (N)
affiche 123,456,789.00	
<code>Console.WriteLine(UnDouble.ToString("F"))</code>	'format virgule fixe (F)
affiche 123456789.00	
<code>End Sub</code>	
<code>End Module</code>	

Autre exemple:

`S=(1.2).ToString("C")` retourne en CurrentCulture Français (par défaut sur mon ordinateur):1,2

Il existe aussi **D** pour décimal, **G** pour général **X** pour hexadécimal.

¶ **Chaîne de format personnalisé:**

On peut créer de toute pièce un format, on utilise pour cela les caractères suivants:

0 indique une espace réservé de 0

Chaque '0' est réservé à un chiffre. Affiche un chiffre ou un zéro. Si le nombre contient moins de chiffres que de zéros, affiche des zéros non significatifs. Si le nombre contient davantage de chiffres à droite du séparateur décimal qu'il n'y a de zéros à droite du séparateur décimal dans l'expression de format, arrondit le nombre à autant de positions décimales qu'il y a de zéros. Si le nombre contient davantage de chiffres à gauche du séparateur décimal qu'il n'y a de zéros à gauche du séparateur décimal dans l'expression de format, affiche les chiffres supplémentaires sans modification.

indique un espace réservé de chiffre.

Chaque '#' est réservé à un chiffre. Affiche un chiffre ou rien. Affiche un chiffre si l'expression a un chiffre dans la position où le caractère# apparaît dans la chaîne de format ; sinon, n'affiche rien dans cette position.

Ce symbole fonctionne comme l'espace réservé au **0**, sauf que les zéros non significatifs et à droite ne s'affichent pas si le nombre contient moins de chiffres qu'il n'y a de caractères # de part et d'autre du séparateur décimal dans l'expression de format.

. (**point**) indique l'emplacement du séparateur décimal (celui affiché sera celui du pays)

Vous devriez donc utiliser le point comme espace réservé à la décimale, même si vos paramètres régionaux utilisent la virgule à cette fin. La chaîne mise en forme apparaîtra dans le format correct pour les paramètres régionaux.

, (**virgule**) indique l'emplacement du séparateur de millier.

Séparateur de milliers. Il sépare les milliers des centaines dans un nombre de quatre chiffres ou plus à gauche du séparateur décimal.

"Littéral" la chaîne sera affichée telle quelle.

% affichera en pour cent.

Multiplie l'expression par 100. Le caractère du pourcentage (%) est inséré

E0 affiche en notation scientifique.

: **et** / sont séparateur d'heure et de date.

; est le séparateur de section : on peut donner 3 formats (un pour les positifs, un pour les négatifs, un pour zéro) séparés par ;

Exemple:

Chaîne de format '0000', le chiffre 145 cela affiche '0145'

Chaîne de format '####', le chiffre 145 cela affiche '145'

Chaîne de format '000.00', le chiffre 45.2 cela affiche '045.20'

Chaîne de format '#,#', le chiffre 12345678 cela affiche '12,345,678'

Chaîne de format '#,,' le chiffre 12345678 cela affiche '12'

La chaîne de formatage ' #,##0.00 ' veut dire obligatoirement 2 chiffres après le séparateur décimal et un avant:

Si on affiche avec ce format

1.1 cela donne 1,10

.5 cela donne 0,50

4563 cela donne 4 563,00

Exemple:

```
Dim N As Double = 19.95
```

```
Dim MyString As String = N.ToString("$#,##0.00;($#,##0.00);Zero")
```

' En page U.S. English culture, MyString = "\$19.95".

' En page Française , MyString = "19,95".

Exemple 2:

```
Dim UnEntier As Integer = 42
```

```
MyString = UnEntier.ToString( "Mon nombre " + ControlChars.Lf + "= #" )
```

Affiche:

Mon nombre
= 42

Str() est toujours accepté

Il permet de transformer une variable numérique et String, qui peut ensuite être affichée.

```
MyString=Str( LeNombre)
```

```
Label1.Text=MyString
```

Pas de formatage..

String.Format

Permet de combiner des informations littérales à afficher sans modification et des zones formatées.

Les arguments de String.Format se décomposent en 2 parties séparées d'une virgule.

- Chaîne de formatage entre guillemets: Exemple "{0} + {1} = {2}": les numéros indiquent l'ordre des valeurs.
- Valeurs à afficher dans l'ordre, la première étant d'indice zéro. Exemple= A, B, A+B

Exemple:

Si A=3 et B=5

```
MsgBox(String.Format("{0} + {1} = {2}",A, B, A+B)) affiche  
3+5=8
```

Autre exemple:

```
Dim MonNom As String = "Phil"  
String.Format("Nom = {0}, heure = {hh}", MonNom,  
DateTime.Now)
```

Le texte fixe est "Nom =" et ", heure = ", les éléments de format sont « {0} » et « {hh} » et la liste de valeurs est MonNom et DateTime.Now.

Cela affiche: **Nom = Phil Heure= 10**

Là aussi on peut utiliser les formats:

- **Prédéfinis:** Ils utilisent là aussi les paramètres régionaux, . Ils utilisent C, D, E, F,G,N,P,R,X comme ToString.

```
MsgBox(String.Format("{0:C}",-456.45))   Affiche  
-456,45
```

```
MsgBox(String.Format("{0:D8}", 456))   Affiche
```

00000456 Décimal 8 chiffres

```
MsgBox(String.Format("{0:P}", 0.14))      Affiche  
14%      Pourcent
```

```
MsgBox(String.Format("{0:X}", 65535))      Affiche  
FFFF      Hexadécimal
```

¶ **Personnalisés:** avec des # et des 0

```
MsgBox(String.Format("{0:##,##0.00}",  
6553.23))
```

La fonction Format (pas la classe `String.Format`) fournit des fonctions similaires mais les arguments sont dans l'ordre inverse (valeur, chaîne de formatage) et il n'y a pas de numéro d'ordre et de `{}`!! C'est pratique pour afficher une seule valeur.

```
MyStr = Format(5459.4, "##,##0.00") ' Returns "5,459.40".
```

```
MyStr = Format(334.9, "###0.00") ' Returns "334.90".
```

```
MyStr = Format(5, "0.00%") ' Returns "500.00%".
```

CultureInfo

On se rend compte que l'affichage est dépendant de la `CurrentCulture` du Thread en cours.

Exemple:

Si la `CurrentCulture` est la 'CultureInfo Us' et que j'affiche avec le format 'C' (monétaire) cela affiche un \$ avant, si je suis en 'CultureInfo Français' cela affiche un après.

Par défaut la `CultureInfo` est celle définie dans Windows.

On peut modifier la `CurrentCulture` par code (voir exemple plus haut).

En français par défaut:

Le séparateur de décimal numérique est le '.'

Exemple : 1.20

Le séparateur décimal monétaire est la ','

Exemple : 1,20



Retour



Index



Sommaire



Suivant

4.9

Le curseur.

Suivant

Sommaire

Comment modifier l'apparence du curseur?

Un curseur est une petite image dont l'emplacement à l'écran est contrôlé par la souris, un stylet ou un trackball. Quand l'utilisateur déplace la souris, le système d'exploitation déplace le curseur.

Différentes formes de curseur sont utilisées pour informer l'utilisateur de l'action que va avoir la souris.

Apparence du curseur

Pour modifier l'aspect du curseur il faut modifier l'objet `Cursor.Current`; l'énumération `Cursors` contient les différents curseurs disponibles:

```
System.Windows.Forms.Cursor.Current =  
System.Windows.Forms.Cursors.WaitCursor
```

ou plus simplement pour afficher le sablier:

```
Cursor.Current = Cursors.WaitCursor
```

Pour revenir au curseur normal:

```
Cursor.Current = Cursors.Default
```

Comme d'habitude il suffit de taper `Cursors.` pour voir la liste des curseurs.

Le curseur peut disparaître et être de nouveau affiché par `Hide` et `Show`.

Curseur sur un controle:

Un contrôle dans une fenêtre possède une propriété `Cursor`; en mode design, si je donne une valeur autre que celle par défaut, `CursorWait` par exemple, cela modifie le curseur quand la souris passe au dessus de l'objet (cela met un sablier dans notre exemple).





4.10	<h2>Lancer une application, une page Web.</h2>		Suivant Sommaire
------	--	--	--

Comment lancer une autre application, un autre programme ?

L'ancienne méthode toujours valable: Shell

Shell lance un programme exécutable.

```
Id=Shell (NomdeProgramme) 'lance l'application  
NomdeProgramme
```

on peut utiliser aussi:

```
Id=Shell( NomdeProgramme, TypedeFenetre, Wait, Timeout)
```

TypedeFenêtre utilise l'énumération **AppWinStyle** pour définir le type de fenêtre de l'application lancée:
AppWinStyle.MaximizedFocus ouvre par exemple l'application en plein écran.

Si vous souhaitez attendre la fin du programme avant de continuer, vous devez définir *Wait* à **True**.

Timeout est le nombre de millisecondes à attendre pour la fin du programme si *Wait* est **True**.

Exemple:

```
ID = Shell("""C:\Program Files\MonFichier.exe" -a -q", , True,  
10000)
```

Dans une chaîne une paire de guillemets doubles adjacents (""") est interprétée comme un caractère de guillemet double dans la chaîne. Ainsi, l'exemple précédent présente la chaîne suivante à la fonction Shell :

```
"C:\Program Files\MonFichier.exe" -a -q
```

La fonction **AppActivate** rend active l'application ou la fenêtre définie par son nom ou son Id (Numéro indentificateur).

```
Dim ID As Integer
```

On peut utiliser:

```
AppActivate("Untitled - Notepad")
```

ou

```
ID = Shell("NOTEPAD.EXE", AppWinStyle.MinimizedNoFocus)  
AppActivate(ID)
```

Avec la Classe Process

La Classe Process fournit l'accès à des processus locaux ainsi que distants, et vous permet de démarrer et d'arrêter des processus système locaux.

Classe de nom à importer : `Imports System.Diagnostics`

On peut **instancier** un Process

```
Dim monProcess As New Process()
```

Ensuite il faut fournir à la classe fille StartInfo **les informations nécessaires au démarrage.**

```
monProcess.StartInfo.FileName = "MyFile.doc"  
monProcess.StartInfo.Verb = "Print"  
monProcess.StartInfo.CreateNoWindow = True
```

Enfin **on lance** le process

```
monProcess.Start()
```

Noter la puissance de cette classe: on donne le nom du document et VB lance l'exécutable correspondant, charge le document, effectue certaines actions.

Dans l'exemple du dessus on ouvre Word on y charge MyFile , on l'imprime, cela sans ouvrir de fenêtre.

¶ On peut utiliser la classe Process en statique (sans instantiation)

```
Process.Start("IExplore.exe")
```

```
Process.Start(MonPathFavori)
```

ou en une ligne:

```
Process.Start("IExplore.exe", "http//:developpez.com")
```

En local on peut afficher un fichier html ou asp

```
Process.Start("IExplore.exe", "C:\monPath\Fichier.htm")
```

```
Process.Start("IExplore.exe", "C:\monPath\Fichier.asp")
```

```
□
```

¶ On peut enfin **utiliser un objet StartInfo**

```
Dim startInfo As New ProcessStartInfo("IExplore.exe")
```

```
startInfo.WindowStyle = ProcessWindowStyle.Minimized
```

```
Process.Start(startInfo)
```

```
startInfo.Arguments = "http//:developpez.com"
```

```
Process.Start(startInfo)
```

```
□
```

Des propriétés du processus en cours permettent de connaître l'Id du processus (**Id**) les **threads**, les **modules**, les **Dll**, la **mémoire**, de connaître le texte de la barre de titre (**MainWindowsTitle**)..

On peut fermer le processus par **Close** ou **CloseMainWindows**

On peut instancer un 'Process' sur une application déjà en cours d'exécution avec **GetProcessByName** et **GetProcessById**:

```
Dim P As Process() = Process.GetProcessesByName("notepad")
```

```
□
```

' **On peut récupérer le processus courant.**

```
Dim ProcessusCourant As Process = Process.GetCurrentProcess()
```

' **Récupérer toutes les instances de Notepad qui tourne en local.**

```
Dim localByName As Process() =
```

```
Process.GetProcessesByName("notepad")
```


' **Récupérer tous les processus en cours** d'exécution grâce à [GetProcesses](#):

```
Dim localAll As Process() = Process.GetProcesses()
```

Processus sur ordinateur distant.

Vous pouvez afficher des données statistiques et des informations sur les processus en cours d'exécution sur des ordinateurs distants, mais vous ne pouvez pas appeler Kill, Start, CloseMainWindows sur ceux-ci.



[Retour](#)



[Index](#)



[Sommaire](#)



[Suivant](#)



4.11	Imprimer.	Suivant	Sommaire
------	-----------	---------	----------

Comment Imprimer ?

Prévoir une longue soirée, au calme, un bon siège, 1 g de paracétamol et un gros thermo de café!!

On devra que l'on peut utiliser pour imprimer:

Soit un composant 'PrintDocument'.

Soit une instance de 'la Class PrintDocument'.

Imprimer 'Hello' avec le composant 'PrintDocument'

L'utilisateur clique sur un bouton, cela imprime 'Hello'

Cet exemple utilise un 'composant PrintDocument'

Comment faire en théorie?

C'est le composant `PrintDocument` qui imprime.

En prendre un dans la boîte à outils, le mettre dans un formulaire. Il apparaît sous le formulaire et se nomme `PrintDocument1`.

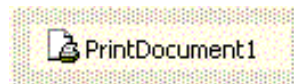
Pour imprimer il faut utiliser la méthode `Print` de ce composant `PrintDocument`; Il faut donc écrire l'instruction suivante:

```
PrintDocument1.Print
```

Cette instruction appelle la procédure événement `PrintDocument1_PrintPage` du composant `PrintDocument` et qui contient la logique d'impression. Un paramètre de cet événement `PrintPage` est l'objet graphique envoyé à l'imprimante (nommé `e`). C'est à vous de dessiner dans l'objet graphique (`e`) ce que vous voulez imprimer. En fin de routine, l'objet graphique sera imprimé (automatiquement).

En pratique:

- Je prends un PrintDocument dans la boîte à outils, je le met dans un formulaire. Il apparaît sous le formulaire et se nomme PrintDocument1.



- Si je double-clique sur PrintDocument1 je vois apparaître la procédure PrintDocument1_PrintPage (qui a été générée automatiquement):

```
Private Sub PrintDocument1_PrintPage(ByVal sender
As System.Object, ByVal e As
System.Drawing.Printing.PrintPageEventArgs)
Handles PrintDocument1.PrintPage
```

```
End Sub
```

C'est cette procédure qui est fondamentale et qui contient les routines d'impression écrites par le programmeur. Les routines d'impression agissent sur l'objet graphique qui sera utilisé pour imprimer, cet objet graphique est fourni dans les paramètres de la procédure (ici c'est e qui est de type PrintPageEventArgs)

- Dans cette routine PrintPage, j'ajoute donc le code dessinant une texte (DrawString) sur l'objet graphique 'e':

```
e.Graphics.DrawString("Hello", New Font("Arial",
80, FontStyle.Bold), Brushes.Black, 150, 125)
```

- Enfin je dessine un bouton nommé 'ButtonPrint' avec une propriété Text contenant "Imprimer Hello" et dans la procédure ButtonPrint_Click j'appelle la méthode Print

```
PrintDocument1.Print()
```

Voici le code complet:

```
Private Sub PrintDocument1_PrintPage(ByVal sender As System.Object,
ByVal e As System.Drawing.Printing.PrintPageEventArgs) Handles
PrintDocument1.PrintPage
```

```
e.Graphics.DrawString("Hello", New Font("Arial", 80,
FontStyle.Bold), Brushes.Black, 150, 125)
```

```
End Sub
```

```
Private Sub ButtonPrint_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ButtonPrint.Click
```

```
PrintDocument1.Print()
```

```
End Sub
```

Si je clique sur le bouton 'ImprimerHello' cela affiche un gros 'Hello'.



La méthode **Print** d'un **PrintDocument** déclenche l'évènement **PrintPage** de ce PrintDocument qui contient le code dessinant sur le graphique de la page à imprimer. En fin de routine PrintPage le graphique est imprimé sur la feuille de l'imprimante.

Toutes les méthodes graphiques écrivant, dessinant, traçant des lignes... sur un graphique permettent donc d'imprimer.

Imprimer un dessin:

Créons une ellipse bleue à l'intérieur d'un rectangle avec la position et les dimensions suivantes : début à 100, 150 avec une largeur de 250 et une hauteur de 250.

```
Private Sub PrintDocument1_PrintPage(ByVal sender As Object,
ByVal e As System.Drawing.Printing.PrintPageEventArgs)
Handles PrintDocument1.PrintPage
    e.Graphics.FillEllipse(Brushes.Blue, New Rectangle(100,
150, 250, 250))
End Sub
```

Imprimer un Message Box indiquant 'Fin d'impression'.

On a étudié l'évènement **PrintPage**, mais il existe aussi les évènements:

BeginPrint et **EndPrint** respectivement déclenchés en début et fin d'impression

Il suffit d'utiliser l'évènement **EndPrint** pour prévenir que l'impression est terminée:

```
Private Sub PrintDocument1_EndPrint(ByVal sender As Object, ByVal e As
System.Drawing.Printing.PrintEventArgs) Handles PrintDocument1.EndPrint
    MessageBox.Show("Fin d'impression")
End Sub
```

On peut même figoler et afficher "Fin d'impression de Nom du document"

Il faut avoir renseigné le **DocumentName**:

```
PrintDocument1.DocumentName = "MyTextFile"
```

Puis écrire:

```
Private Sub PrintDocument1_EndPrint(ByVal sender As Object, ByVal e As
```

```
System.Drawing.Printing.PrintEventArgs) Handles PrintDocument1.EndPrint
    MessageBox.Show( "Fin d'impression de "+PrintDocument1.DocumentName)
End Sub
```

Même programme: Imprimer 'Hello' mais avec la Classe PrintDocument

L'utilisateur clique sur un bouton, cela imprime 'Hello'

Cet exemple utilise 'une instance de la Classe PrintDocument'. On ne met pas de composant 'PrintDocument' dans le formulaire.

Comment faire en théorie?

Il faut importer l'espace de nom 'Printing' par :

```
Imports System.Drawing.Printing
```

Il faut créer une instance de la Classe PrintDocument dans le module.

```
Dim pd as PrintDocument = new PrintDocument()
```

Il faut créer soi même, une routine `pd_PrintPage` .

```
Private Sub pd_PrintPage(sender As object, ev As
System.Drawing.Printing.PrintPageEventArgs)
End sub
```

Il faut indiquer le "lien" entre l'objet `pd` et la routine événement `PrintPage`

```
AddHandler pd.PrintPage, AddressOf Me.pd_PrintPage
```

Dans la procédure `Button_Click` d'un bouton "Imprimer" il faut appeler la méthode `Print` du `PrintDocument` pour effectuer l'impression du document .

```
pd.Print
```

Cela déclenche la procédure `Private Sub pd_PrintPage` précédemment écrite, dans laquelle on a ajouté:

```
ev.Graphics.DrawString ("Hello", printFont, Brushes.Black,
leftMargin, yPos, new StringFormat()).
```

Cela donne le code complet:

```
Imports System.Drawing.Printing
```

```
Public Class Form1
Inherits System.Windows.Forms.Form
```

```
Dim pd As PrintDocument = New PrintDocument 'Assumes the default printer
```

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
```

```
    AddHandler pd.PrintPage, AddressOf Me.Pd_PrintPage
```

```
End Sub
```

```
Private Sub Pd_PrintPage(ByVal sender As System.Object, ByVal e As System.Drawing.Printing.PrintPageEventArgs)
```

```
    e.Graphics.DrawString("Hello", New Font("Arial", 80, FontStyle.Bold), Brushes.Black, 150, 125)
```

```
End Sub
```

```
Private Sub ButtonPrint_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ButtonPrint.Click
```

```
    pd.Print()
```

```
End Sub
```

```
End Class
```

Comment choisir l'imprimante?

Le composant `PrintDialog` permet **le choix de l'imprimante**, de la zone à imprimer (tout, la sélection..) et donne accès aux caractéristiques de l'imprimante.

Comment l'utiliser?

Il faut créer une instance de `PrintDialog`:

```
Dim dlg As New PrintDialog
```

Il faut indiquer au `PrintDialog` sur quel `PrintDocument` travailler:

```
dlg.Document = pd
```

Puis ouvrir la fenêtre `PrintDialog` avec la méthode `ShowDialog`.

L'utilisateur choisit son imprimante puis clique sur 'Ok'.

Si elle retourne Ok, on imprime.

Quand l'utilisateur clique sur le bouton `ButtonPrint` ('Imprimer') la fenêtre `PrintDialog` s'ouvre:

Voici le code complet:

```
Private Sub ButtonPrint_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ButtonPrint.Click
```

```
Dim dlg As New PrintDialog
```

```
dlg.Document = pd
```

```
Dim result As DialogResult = dlg.ShowDialog()
```

```
If (result = System.Windows.Forms.DialogResult.OK) Then
```

```
    pd.Print()
```

```
End If
```

End Sub

Comment modifier la page à imprimer?

Comment choisir d'imprimer en **portrait ou paysage? modifier les marges..**

Il faut utiliser un composant [PageSetUpDialog](#).

Pour stocker les informations sur la page (marges...) il faut un [PageSetting](#)

Je lie le PageSetting au PageSetUpDialog en donnant à la propriété [PageSettings](#) du PageSetUpDialog le nom du PageSetting.

puis j'ouvre le PageSetUpDialog.

Au retour le PageSetting contient les modifications, je les 'passe' au PrintDocument avant d'imprimer.

Cela donne:

```
Dim psDlg As New PageSetupDialog

Dim LePageSettings As New PageSettings

psDlg. PageSettings = LePageSettings

psDlg.ShowDialog()

pd.DefaultPageSettings = LePageSettings
```

Prévisualisation de la page à imprimer?

On utilise pour cela un [PrintPreviewDialog](#), on lui indique quel PrintDocument pré visualiser en l'assignant à sa méthode [document](#) puis on l'affiche par [ShowDialog\(\)](#).

```
Dim dllg As New PrintPreviewDialog

dllg.Document = pd

dllg.ShowDialog()
```

Construction d'une application d'impression complexe:

Comment imprimer le contenu d'un fichier texte?

Tous les didacticiels (Microsoft compris) donnent cet exemple.

La première chose que vous devez faire est d'écrire votre logique d'impression. Pour cela, quand la méthode **PrintDocument.Print()** est appelée, les événements suivants sont déclenchés.

- **BeginPrint**
- **PagePrint** (un ou plusieurs s'il y a plusieurs pages à imprimer)
- **EndPrint**

L'argument d'événement de **PagePrint** (**PagePrintEventArgs**) comprend une propriété **HasMorePages**. Si celle-ci a la valeur **True** lors du retour de votre gestionnaire d'événements, **PrintDocument** définit une nouvelle page et déclenche de nouveau l'événement **PagePrint**.

Voyons la logique dans votre gestionnaire d'événements **PagePrint** :

- Imprimez le contenu de la page en utilisant les informations des arguments d'événement. Les arguments d'événement contiennent l'objet **Graphics** pour l'imprimante, le **PageSettings** pour cette page, les limites de la page, et la taille des marges.

Il faut dans la **procédure PagePrint** imprimer ligne par ligne en se déplaçant à chaque fois vers le bas d'une hauteur de ligne.

Pour 'simplifier', on considère que chaque ligne ne déborde pas à droite!!

- Déterminer s'il reste des pages à imprimer.
- Si c'est le cas, **HasMorePages** doit être égal à **True**.
- S'il n'y a pas d'autres pages, **HasMorePages** doit être égal à **False**.

```
Public Class ExampleImpression
    Inherits System.Windows.Forms.Form

    ....

    private printFont As Font
    private streamToPrint As StreamReader

    Public Sub New ()
        MyBase.New
        InitializeComponent()
    End Sub

    'Événement survenant lorsque l'utilisateur clique sur le bouton
    'Imprimer'
    Private Sub printButton_Click(sender As object, e As
System.EventArgs)

        Try
            streamToPrint = new StreamReader ("PrintMe.Txt")
            Try
                printFont = new Font("Arial", 10)
                Dim pd as PrintDocument = new PrintDocument()
                'déclaration du PrintDocument
                AddHandler pd.PrintPage, AddressOf Me.pd_PrintPage
                pd.Print()
            End Try
        End Try
    End Sub
End Class
```



```

        Finally
            streamToPrint.Close()
        End Try

    Catch ex As Exception
        MessageBox.Show("Une erreur est survenue: - " + ex.Message)
    End Try

End Sub

'Evènement survenant pour chaque page imprimer
Private Sub pd_PrintPage(sender As object, ev As
System.Drawing.Printing.PrintPageEventArgs)

    Dim lpp As Single = 0    'nombre de ligne par page
    Dim yPos As Single = 0  'ordonnée
    Dim count As Integer = 0 'numéro de ligne
    Dim leftMargin As Single = ev.MarginBounds.Left
    Dim topMargin As Single = ev.MarginBounds.Top
    Dim line as String

    'calcul le nombre de ligne par page
    ' hauteur de la page/hauteur de la police de caractère
    lpp = ev.MarginBounds.Height / printFont.GetHeight(ev.Graphics)

    'lit une ligne dans le fichier
    line=streamToPrint.ReadLine()

    'Boucle affichant chaque ligne

    while (count < lpp AND line <> Nothing)

        yPos = topMargin + (count *
printFont.GetHeight(ev.Graphics))

        'Ecrit le texte dans l'objet graphique
        ev.Graphics.DrawString (line, printFont, Brushes.Black,
leftMargin, _
                                yPos, new StringFormat())

        count = count + 1

        if (count < lpp) then
            line=streamToPrint.ReadLine()
        end if

    End While

    'S'il y a encore des lignes, on réimprime une page
    If (line <> Nothing) Then
        ev.HasMorePages = True
    Else
        ev.HasMorePages = False
    End If

End Sub

....

End Class

```

On a vu que pour 'simplifier', on considère que chaque ligne ne déborde pas à droite.

Dans la pratique, pour gérer les retours à la ligne automatiques on peut dessiner dans un rectangle.

(Voir la page sur les graphiques.)

Propriétés du 'PrintDocument':

On peut sans passer par une 'boite de dialog' gérer directement l'imprimante, les marges, le nombre de copies..

Si pd est le PrintDocument:

`pd.PrinterSetting` désigne l'imprimante en cours

`pd.PrinterSetting.PrinterName` retourne ou définit le nom de cet imprimante

`pd.PrinterSetting.Printerresolution` donne la résolution de cette imprimante.

`pd.PrinterSetting.installedPrinted` donne toutes les imprimantes installées.

La propriété `DefaultPageSetting` est en rapport avec les caractéristiques de la page.

`pd.PrinterSetting.DefaultPageSetting.Margins` donne les marges

`pd.PrinterSetting.PrintToFile` permettrait d'imprimer dans un fichier (non testé)

Imprime le formulaire en cours.

Exemple fournit par Microsoft.

```
Private Declare Function BitBlt Lib "gdi32.dll" Alias
"BitBlt" (ByVal _
    hdcDest As IntPtr, ByVal nXDest As Integer, ByVal nYDest
As _
    Integer, ByVal nWidth As Integer, ByVal nHeight As
Integer, ByVal _
    hdcSrc As IntPtr, ByVal nXSrc As Integer, ByVal nYSrc As
Integer, _
    ByVal dwRop As System.Int32) As Long
Dim memoryImage As Bitmap
Private Sub CaptureScreen()
    Dim mygraphics As Graphics = Me.CreateGraphics()
    Dim s As Size = Me.Size
    memoryImage = New Bitmap(s.Width, s.Height, mygraphics)
```

```

Dim memoryGraphics As Graphics =
Graphics.FromImage(memoryImage)
Dim dc1 As IntPtr = mygraphics.GetHdc
Dim dc2 As IntPtr = memoryGraphics.GetHdc
BitBlt(dc2, 0, 0, Me.ClientRectangle.Width, _
Me.ClientRectangle.Height, dc1, 0, 0, 13369376)
mygraphics.ReleaseHdc(dc1)
memoryGraphics.ReleaseHdc(dc2)
End Sub
Private Sub PrintDocument1_PrintPage(ByVal sender As
System.Object, _
ByVal e As System.Drawing.Printing.PrintPageEventArgs)
Handles _
PrintDocument1.PrintPage
e.Graphics.DrawImage(memoryImage, 0, 0)
End Sub
Private Sub PrintButton_Click(ByVal sender As System.Object,
ByVal e As _
System.EventArgs) Handles PrintButton.Click
CaptureScreen()
PrintDocument1.Print()
End Sub

```

Imprime un contrôle DataGridView.

Exemple fournit par Microsoft.

Cet exemple nécessite :

- un contrôle **Button**, nommé ImprimerGrid, dans le formulaire ;
- un contrôle **DataGridView** nommé DataGridView1 ;
- un composant **PrintDocument** nommé PrintDocument1.

Comme d'habitude **PrintPage** imprime e.Graphics.

D'après ce que j'ai compris, l'évènement **Paint** redessine un contrôle

mais on peut choisir le contrôle et l'endroit où le redessiner,

Je redessine donc grâce à **Paint**, le DataGridView dans e.graphics.

PaintEventArgs Fournit les données pour l'évènement Paint:

PaintEventArgs spécifie l'objet **graphics** à utiliser pour peindre le contrôle, ainsi que le **ClipRectangle** dans lequel le peindre.

InvokePaint déclenche l'évènement Paint

```

Private Sub ImprimerGrid_Click(ByVal sender As System.Object,
ByVal e As _

```

```
System.EventArgs) Handles PrintGrid.Click
PrintDocument1.Print()
End Sub

Private Sub PrintDocument1_PrintPage(ByVal sender As
System.Object, _
    ByVal e As System.Drawing.Printing.PrintPageEventArgs)
Handles _
    PrintDocument1.PrintPage
    Dim myPaintArgs As New PaintEventArgs(e.Graphics, New
Rectangle(New _
    Point(0, 0), Me.Size))
    Me.InvokePaint(DataGrid1, myPaintArgs)
End Sub
```

Simple, non!! Je plaisante!!



4.12 Dessiner.

Suivant

Sommaire

Avec **GDI+** utilisé par VB.NET, on utilise des objets **Graphics** qui sont des zones de dessin.

Image (BitMap ou MetaFile) contient une image.

Rectangle permet de définir une zone.

Pen correspond à un Stilet.

Font correspond à une police de caractères

Brush , c'est une brosse.

Sur quoi dessiner?

Il faut définir une zone de dessin, **un objet Graphics**. On peut y inclure des objets **Image** (des BitMap ou des MetaFile)

Pour obtenir un objet **Graphics**, il y a plusieurs façons:

- Soit on instance un objet Graphics.

Dim g as Graphics 'Graphics contient
Nothing, je ne peux rien en faire.

Il faut donc y mettre un **objet Image** (un
BitMap ou un MetaFile) pour pouvoir travailler
dessus.

Pour obtenir un BitMap par exemple, on peut:

Soit créer un objet BitMap vide:

```
Dim newBitmap As Bitmap  
= New Bitmap(600, 400)
```

```
Dim g as Graphics =  
Graphics.FromImage(newBitmap)
```

Paramètres= taille du
BitMap mais il y a plein
de surcharges

Soit créer un BitMap à partir d'un
fichier sur disque

C'est pratique si on
veut **modifier une image**
qui est dans un fichier:
on la lit dans un BitMap
puis on la passe dans
l'objet Graphics.

```
Dim myBitmap as New  
Bitmap("maPhoto.bmp")  
'Charge maPhoto dans le  
BitMap  
Dim g as Graphics =  
Graphics.FromImage(myBitmap)  
'Crée un Graphics et y  
met le BitMap
```

g est un Graphics
contenant l'image
'maPhoto.bmp' et je peux
la modifier.

Attention: le Graphics n'est pas
'visible', pour le voir il faut le
mettre dans un composant (un
PictureBox par exemple) qui lui

sera visible. On verra cela plus bas.

¶ Soit on appelle la méthode CreateGraphics d'un contrôle ou d'un formulaire

On appelle la méthode **CreateGraphics** d'un contrôle ou d'un formulaire afin d'obtenir une référence à un objet Graphics représentant la surface de dessin de ce contrôle ou formulaire.

Cette méthode est utilisée si vous voulez dessiner sur un formulaire ou un contrôle existant ;

```
Dim g as Graphics
g = Me.CreateGraphics      'Pour un formulaire
```

```
Dim g as Graphics
g = Panel1.CreateGraphics  'Pour un contrôle
Panel
```

On peut ensuite dessiner sur **g**, cela sera immédiatement visible sur le contrôle ou le formulaire.

Il faut quand on n'utilise plus l'objet graphics, utiliser la méthode **Dispose** pour le libérer.

¶ Soit on récupère l'objet Graphics argument de l'évènement Paint d'un contrôle.

L'évènements **Paint** des contrôles se déclenche lorsque le contrôle est redessiné, un objet **Graphics** est fourni comme une valeur de **PaintEventArgs**.

Pour obtenir une référence à un objet Graphics

à partir des PaintEventArgs de l'événement Paint et dessiner:

1. Déclarez l'objet `Graphics`
2. Assignez la variable pour qu'elle référence l'objet `Graphics` passé dans les `PaintEventArgs`.
3. Dessinez dans l'objet `Graphics`.

```
Private Sub Form1_Paint(sender As Object, pe  
As PaintEventArgs) Handles _  
MyBase.Paint
```

```
    Dim g As Graphics = pe.Graphics  
    ' Dessiner dans pe ici..  
End Sub
```

Noter bien que `pe` est visible uniquement dans `Form1_Paint`

Pour forcer le déclenchement de l'évènement `Paint` et dessiner, on utilise la méthode `OnPaint`

Comment dessiner?

La classe `Graphics` fournit des méthodes permettant de dessiner

`DrawImage` 'Ajoute une image (Bitmap ou MetaFile)

`DrawLine` 'Trace une ligne

`DrawString` 'Ecrit un texte

`DrawPolygon` 'Dessine un polygone

...

En GDI+ on envoie des paramètres à la méthode pour dessiner:

Exemple:

`MonGraphique.DrawEllipse(New Pen(Couleur),r) 'cela`
dessine une ellipse

Les 2 paramètres sont: la couleur et le rectangle dans lequel on dessine l'ellipse.

Pour travailler on utilise les objets:

Brush (Brosse)	<p>Utilisé pour remplir des surfaces fermées avec des motifs, des couleurs ou des bitmaps.</p> <p>Il y a plusieurs types de brush:</p> <p>La brosse peut être pleine et ne contenir qu'une couleur.</p> <pre>Dim SB= New SolidBrush(Color.Red)</pre> <p>TextureBrush utilise une image pour remplir.</p> <pre>Dim SB= New TextureBrush(MonImage)</pre> <p>LinearGradientBrush permet des dégradés (passage progressive d'une couleur à une autre).</p> <pre>Dim SB= New LinearGradientBrush(PointDébut, PointFin,Color1, Color2)</pre> <p>Les HatchBrush sont des brosse hachurées</p> <pre>Dim HatchBrush hb = new HatchBrush(HatchStyle.ForwardDiagonal, Color.Green,Color.FromArgb(100, Color.Yellow))</pre> <p>Les PathGradient sont des brosses plus complexes.</p>

Pen (Stylet)	<p>Utilisé pour dessiner des lignes et des polygones, tels que des rectangles, des arcs et des secteurs.</p> <p>Comment créer un Stylet?</p> <p><code>Dim blackPen As New Pen(Color.Black)</code> on donne la couleur</p> <p><code>Dim blackPen As New Pen(Color.Black, 3)</code> on donne couleur et l'épaisseur</p> <p><code>Dim blackPen As New Pen(MyBrush)</code> on peut même créer un stylet avec une brosse</p> <p>Propriétés de ce Stylet:</p> <p><code>DashStyle</code> permet de faire des pointillés.</p> <p><code>StartCap</code> et <code>EndCap</code> définissent la forme du début et de la fin du dessin (rond, carré, flèche..)</p>
Font (Police)	<p>Utilisé pour décrire la police utilisée pour afficher le texte.</p> <p><code>Dim Ft= New Font("Lucida sans unicode",60)</code> 'paramètres=nom de la font et taille</p> <p>Il y a de nombreuses surcharges.</p> <p><code>Dim Ft= New Font("Lucida sans unicode",60, FontStyle.Bold)</code>'pour écrire en gras</p>
Color (Couleur)	<p>Utilisé pour décrire la couleur utilisée pour afficher un objet particulier. Dans GDI+, la couleur peut être à contrôle alpha.</p> <p><code>System.Drawing.Color.Red</code> pour le rouge</p>

Point	<p>Ils ont des coordonnées x, y</p> <pre>Dim point1 As New Point(120, 120) ' Point avec des Integer</pre> <p>ou <code>des PointF</code> 'avec des Singles</p>
Rectangle	<p>Permet de définir un rectangle.</p> <pre>Dim r As New RectangleF(0, 0, 100, 100)</pre> <p>On remarque que le F après Point ou Rectangle veut dire 'Float', et nécessite l'usage de Single.</p>

Comment faire?

Dessiner une ligne sur le graphique:

Pour dessiner une ligne, on utilise `DrawLine`.

```
Dim blackPen As New Pen(Color.Black, 3) 'créer un
stylet noir d'épaisseur 3
' Créer des points
Dim point1 As New Point(120, 120) 'créer des points
Dim point2 As New Point(600, 100)
' Dessine la ligne
e.Graphics.DrawLine(blackPen, point1, point2)
```

On aurait pu utiliser une surcharge de `Drawline` en spécifiant directement les **coordonnées** des points.

```
Dim x1 As Integer = 120
Dim y1 As Integer = 120
Dim x2 As Integer = 600
Dim y2 As Integer = 100
e.Graphics.DrawLine(blackPen, x1, y1, x2, y2)
```

Dessiner une ellipse:

Définir un rectangle dans lequel sera dessiné l'ellipse.

```
Dim r As New RectangleF(0, 0, 100, 100)
```

```
g.DrawEllipse(New Pen(Color.Red), r)' Dessinons l'  
ellipse
```

Dessiner une rectangle:

```
myGraphics.DrawRectangle(myPen, 100, 50, 80, 40)
```

Comme d'habitude on peut fournir après le stylet des coordonnées(4), des points (2) ou un rectangle.

Dessiner un polygone:

```
Dim MyPen As New Pen(Color.Black, 3)  
' Créons les points qui définissent le polygone  
Dim point1 As New Point(150, 150)  
Dim point2 As New Point(100, 25)  
Dim point3 As New Point(200, 5)  
Dim point4 As New Point(250, 50)  
Dim point5 As New Point(300, 100)  
Dim point6 As New Point(350, 200)  
Dim point7 As New Point(250, 250)  
Dim curvePoints As Point() = {point1, point2, point3,  
point4, _  
point5, point6, point7}  
' Dessinons le Polygone.  
e.Graphics.DrawPolygon(MyPen, curvePoints
```

Dessiner un rectangle plein:

```
e.FillRectangle(new SolidBrush(Color.red), 300,15,50,50)
```

Il existe aussi DrawArc, DrawCurve, DrawBezier DrawPie..

Ecrire du texte sur le graphique:

Pour cela on utilise la méthode DrawString de l'objet graphique:

```
g.DrawString ("Salut", Me.Font, New SolidBrush  
(ColorBlack), 10, 10)
```

Paramètres:

Texte à afficher.

Police de caractères

Brosse , cela permet d'écrire avec des textures.

Coordonnées.

Si on spécifie un rectangle à la place des 2 derniers paramètres, le texte sera affiché dans le rectangle **avec passage à la ligne** si nécessaire:

```
Dim rectangle As New RectangleF (100, 100, 150, 150 )
```

```
Dim T as String= "Chaîne de caractères très longue"
```

```
g.DrawString (T, Me.Font, New SolidBrush (ColorBlack),  
Rectangle)
```

On peut même **imposer un format** au texte:

Exemple: centrer le texte.

```
Dim Format As New StringFormat()
```

```
Format.Alignment=StringAlignment.Center
```

```
g.DrawString (T, Me.Font, New SolidBrush (ColorBlack),  
Rectangle, Format)
```

On peut **mesurer la longueur (ou le nombre de lignes)** d'une chaîne:

Avec `MeasureString`

Exemple 1: centrer le texte: pour cela, calculer la longueur de la chaîne, puis calculer le milieu de l'écran moins la 1/2 longueur de la chaîne:

```
Dim W As Double= Me.DisplayRectangle.Width/2
```

```
Dim L As SizeF= e.Graphics.MeasureString (Texte, TextFont)
```

```
Dim StartPos As Double = W - (L.Width/2)
```

```
g.Graphics.MeasureString (T, Me.Font, New SolidBrush  
(ColorBlack), Rectangle, StartPos, 10)
```

Exemple 2: Calculer le nombre de ligne et le nombre de caractères d'une chaîne:

```
g.Graphics.MeasureString (T, Me.Font, New SolidBrush  
(ColorBlack), Rectangle, Nex StringFormat()  
NombredeCaractères, NombredeLignes)
```

Ajouter une image sur le graphique:

Pour cela on utilise la méthode `DrawImage` de l'objet graphique:

```
g.Graphics.DrawImage(New Bitmap("sample.jpg"), 29, 20,  
283, 212)
```

On peut travailler avec des images .jpeg .png .bmp .Gif .icon .tiff .exif

Travailler sur un Objet Image

Charger une image:

Si on veut afficher une image bitmap ou vectoriel, il faut fournir à l'objet Graphics un objet bitmap ou vectoriel. C'est la méthode `DrawImage` qui reçoit l'objet Metafile ou Bitmap comme argument. L'objet Bitmap, si on le désire peut charger un fichier qui sera affiché.

```
Dim myBMP As New BitMap ("MonImage.bmp")
```

```
myGraphics.DrawImage(myBMP, 10, 10)
```

Le point de destination du coin supérieur gauche de l'image, (10, 10), est spécifié par les deuxième et troisième paramètres.

```
myGraphics.FromImage(myBMP) est aussi possible
```

On peut utiliser plusieurs formats de fichier graphique : BMP, GIF, JPEG, EXIF, PNG, TIFF et ICON.

Cloner une image:

La classe Bitmap fournit une méthode `Clone` qui permet de créer une copie d'un objet existant. La méthode `Clone`

admet comme paramètre un rectangle source qui vous permet de spécifier la portion de la Bitmap d'origine à copier. L'exemple suivant crée un objet Bitmap en clonant la moitié supérieure d'un objet Bitmap existant. Il dessine ensuite les deux images.

```
Dim originalBitmap As New Bitmap("Spiral.png") 'on charge un fichier png dans un BitMap
Dim sourceRectangle As New Rectangle(0, 0, originalBitmap.Width, _ originalBitmap.Height / 2) 'on définit un rectangle

Dim secondBitmap As Bitmap = originalBitmap.Clone(sourceRectangle, _ PixelFormat.DontCare) 'on définit un second BitMap Clonant une partie du 1ere BitMap avec le rectangle

'On met les 2 BitMap dans un Graphics

myGraphics.DrawImage(originalBitmap, 10, 10)
myGraphics.DrawImage(secondBitmap, 150, 10)
```

Enregistrer une image sur disque:

On utilise pour cela la méthode Save.

Exemple: enregistrer le BitMap newBitMap dans 'Image1.jpg'

```
newBitmap.Save("Image1.jpg", ImageFormat.Jpeg)
```

Comment voir un Graphics?

Si on a instance un objet Graphics, on ne le voit pas. Pour le voir il faut le mettre dans un PictureBox par exemple:

Exemple:

```
Dim newBitmap As Bitmap = New Bitmap(200, 200) 'créons un BitMap
```

```
Dim g As Graphics = Graphics.FromImage(newBitmap) 'créons un Graphics et y mettre le BitMap
```

```
Dim r As New RectangleF(0, 0, 100, 100) ' Dessinons une
```

ellipse

```
g.DrawEllipse(New Pen(Color.Red), r)
```

Comment voir l'ellipse?

Ajoutons un `PictureBox` au projet, et donnons à la propriété `Image` de ce `PictureBox` le nom du `Bitmap` du `Graphics`:

```
PictureBox1.Image = newBitmap
```

L'ellipse rouge apparaît!! Si ,Si!!

Paint déclenché par Resize.

Par défaut `Paint` n'est pas déclenché quand un contrôle ou formulaire est redimensionné, pour forcer à redessiner en cas de redimensionnement, il faut mettre le style `Style.Resizedraw` du formulaire ou du contrôle à `true`.

```
SetStyle (Style.Resizedraw, true)
```

Cette syntaxe marche, la suivante aussi (pour le formulaire)

```
Me.SetStyle (Style.Resizedraw, true) 'pour tous les objets du formulaire?
```

Mais `PictureBox1.SetStyle (Style.Resizedraw, true)` n'est pas accepté!!

Afficher un texte en 3D

Afficher un texte en 3d.

```
PrivateSub TextEn3D(ByVal g As Graphics, ByVal position As PointF, ByVal text As String, ByVal ft As Font, ByVal c1 As Color, ByVal c2 As Color)
```

```
    Dim rect AsNew RectangleF(position, g.MeasureString(text, ft))
```




```
Dim bOmbre AsNew LinearGradientBrush(rect,
Color.Black, Color.Gray, 90.0F)
g.DrawString(text, ft, bOmbre, position)
position.X -= 2.0F
position.Y -= 6.0F
rect = New RectangleF(position,
g.MeasureString(text, ft))
Dim bDegrade AsNew
LinearGradientBrush(rect, c1, c2, 90.0F)
g.DrawString(text, ft, bDegrade, position)
EndSub
```

Espace de nom

Pour utiliser les graphiques il faut que `System.Drawing` soit importé (ce qui est fait par défaut).

(System.Drawing.DLL comme références de l'assembly)



4.13	Ajouter une aide.		Sommaire
-------------	----------------------------------	---	-----------------

Quand l'utilisateur utilise votre logiciel, il se pose parfois des questions, comment l'aider?

Avec des aides que le programmeur doit créer et ajouter au programme.

Généralisées sur les 4 sortes d'aide.

La Class [Help](#) permet d'ouvrir un fichier d'aide.

Le composant [HelpProvider](#) offre 2 types d'aide.

- Le premier consiste à **ouvrir un fichier d'aide grâce à F1.**
- Quant au second, il peut afficher une **aide brève** pour chacun des contrôles en utilisant **le bouton d'aide (?)**. Il s'avère particulièrement utile dans les boîtes de dialogue modale.

Le composant [ToolTip](#) offre lui:

- une aide propre à chaque contrôle des Windows Forms.

Les fichiers d'aide:

On peut utiliser les formats:

HTML Fichier .htm

HTMLHelp 1.x ou version ultérieure) Fichier .chm

HLP Fichier .hlp les plus anciens.

Comment créer ces fichiers:

Pour les fichiers HTM:

Utiliser Word, ou FontPage, ou Netscape Compositeur...

Pour les fichiers HLP:

Utiliser Microsoft HelpWorkshop livré avec VB6

Pour les fichiers CHM:

Thierry AIM fournit sur le site developpez.com un excellent:

[Cours pour créer un fichier CHM](#)

On conseille d'utiliser les fichiers chm.

Utilisation des fichiers d'aide:

Appel direct:

La classe `Help` permet d'ouvrir directement par code un fichier d'aide.

C'est ce qu'on utilise dans le menu '?' d'un programme (sous menu 'Aide'); dans la procédure correspondante (Sub Aide_Click) on écrit:

```
Help.ShowHelp (Me, "MonAide.html")
```

MonAide.html doit être dans le répertoire de l'application (répertoire Bin)

Cela peut être un URL , l'adresse d'une page sur Internet!!

Il peut y avoir un 3ème paramètre: on verra cela plus bas (C'est le même paramètre que la propriété HelpNagigator

de HelpProvider).

Appel par F1:

Vous pouvez utiliser le composant [HelpProvider](#) pour **attacher des rubriques d'aide** figurant dans un fichier d'aide (au format HTML, HTMLHelp 1.x ou ultérieur) à **des contrôles** de l'interface utilisateur.

Quand on met un composant dans un formulaire (avec dans la propriété [HelpNamespace](#), le nom de fichier d'aide), cela ajoute aux contrôles de ce formulaire les propriétés:

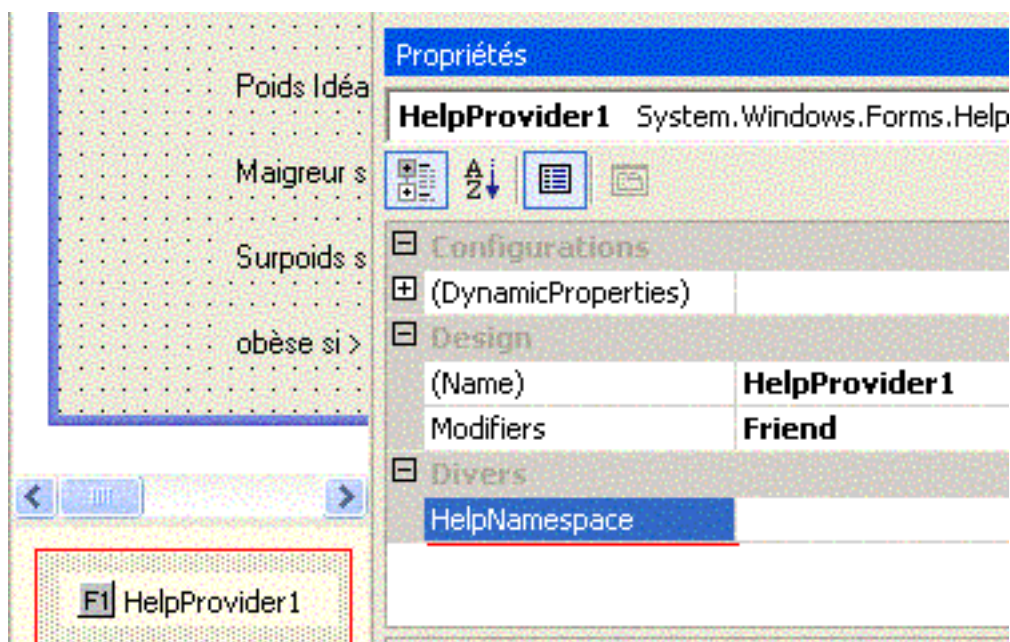
- [HelpNavigator](#) qui déterminent le **type** d'appel (par numéro de rubrique, mot clé..)
- [HelpKeyword](#) qui contient le **paramètre de recherche** (le numéro de rubrique, le mot clé..)

Quand l'utilisateur est sur le contrôle et qu'il clique sur **F1** la rubrique d'aide s'ouvre.

Pour créer cette aide:

Faites glisser un composant **HelpProvider** de la boîte à outils vers votre formulaire.

Le composant se place dans la barre d'état située au bas de la fenêtre.



Dans la fenêtre Propriétés du HelpProvider , donner à la propriété [HelpNamespace](#), un nom de fichier d'aide .chm, ou .htm.

Dans la fenêtre Propriétés du contrôle de l'interface qui doit déclencher l'aide, donner à la propriété [HelpNavigator](#) une valeur de l'énumération HelpNavigator.

Cette valeur détermine la façon dont la propriété **HelpKeyword** est passée au système d'aide. HelpNagigator peut prendre la valeur:

AssociateIndex	Indique que l'index d'une rubrique spécifiée est exécuté dans l'URL spécifiée.
Find	Indique que la page de recherche d'une URL spécifiée est affichée.
Index	Indique que l'index d'une URL spécifiée est affiché.
KeywordIndex	Spécifie un mot clé à rechercher et l'action à effectuer dans l'URL spécifiée.
TableOfContents	Indique que le sommaire du fichier d'aide HTML 1.0 est affiché.
Topic	Indique que la rubrique à laquelle l'URL spécifiée fait référence est affichée.

Définissez la propriété [HelpKeyword](#) dans la fenêtre Propriétés. (la valeur de cette propriété sera passé au fichier d'aide afin de déterminer la rubrique d'aide à afficher)

Au moment de l'exécution, le fait d'appuyer sur **F1** lorsque le contrôle (dont vous avez défini les propriétés **HelpKeyword** et **HelpNavigator**) a le focus ouvre le fichier d'aide associé à ce composant **HelpProvider**.

Remarque Vous pouvez définir, pour la propriété **HelpNamespace**, une adresse <http://> (telle qu'une page Web). Cela permet d'ouvrir le navigateur par défaut sur la page Web avec la chaîne indiquée dans la propriété

HelpKeyword utilisée comme ancre (pour accéder à une section spécifique d'une page HTML).

Dans le code il faut utiliser la syntaxe `HelpProvider.SetHelpKeyword=".."`

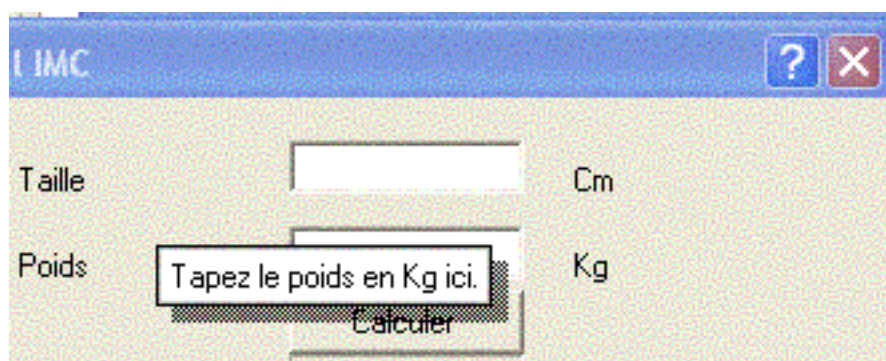
Exemple:

Pour afficher la page d'aide sur les formes ovales, sélectionnez la valeur `HelpNavigator.KeyWordIndex` dans la liste déroulante **Help Navigator** ; dans la zone de texte **HelpKeyword**, 'tapez « ovales » (sans chevrons).

Utilisation du bouton d'aide:

Vous pouvez afficher l'aide pour un contrôle via le **bouton Aide (?)** situé dans la partie droite de la barre de titre .

Il faut que l'utilisateur clique sur le bouton d'aide (?) puis sur le contrôle qui nécessite une aide, ce qui entraîne l'ouverture d'un carré blanc contenant un message d'aide.



L'affichage de l'aide de cette façon convient particulièrement aux boîtes de dialogue. En effet, avec un affichage modal des boîtes de dialogue, il n'est pas facile d'ouvrir des systèmes d'aide externes, dans la mesure où les boîtes de dialogue modales doivent être fermées avant que le focus puisse passer à une autre fenêtre. Le bouton Réduire ou Agrandir ne doit pas être affiché dans la barre de titre. Il s'agit d'une convention pour les boîtes de dialogue alors que les formulaires disposent généralement de boutons Réduire et Agrandir.

Pour afficher l'aide contextuelle

Faites glisser un composant [HelpProvider](#) de la boîte à outils vers votre formulaire.

Le contrôle est placé dans la barre d'état des composants située au bas de la fenêtre.

Attribuer aux propriétés Minimize et Maximize de la fenêtre la valeur **false**.

Puis,

Dans la fenêtre Propriétés **de la fenêtre**, donner à la propriété **HelpButton** la valeur **true**. Cette configuration permet d'afficher dans la partie droite de la barre de titre du formulaire un bouton contenant un point d'interrogation.

Sélectionnez le contrôle pour lequel vous souhaitez afficher l'aide dans votre formulaire et mettre dans la propriété [HelpString](#) la chaîne de texte qui sera affichée dans une fenêtre de type [ToolTip](#).

Test:

Appuyez sur **F5**.

Appuyez sur le bouton Aide (?) de la barre de titre et cliquez sur le contrôle dont vous avez défini la propriété **HelpString**. Le tooltip apparaît.

Utilisation des info bulles.

Le composant [ToolTip](#) peut servir à afficher des messages d'aide courts et spécialisés relatifs à des contrôles individuels.

Cela ouvre une petite fenêtre indépendante rectangulaire dans laquelle s'affiche une brève description de la raison d'être d'un contrôle lorsque le curseur de la souris pointe sur celui-ci.

Il fournit une propriété qui précise le texte affiché

pour chaque contrôle du formulaire.

En outre, il est possible de configurer, pour le composant **ToolTip**, le délai qui doit s'écouler avant qu'il ne s'affiche.

Comment faire:

Ajoutez le contrôle **ToolTip** au formulaire.

Chaque contrôle à maintenant une propriété **ToolTip** ou on peut mettre le texte à afficher dans l'info bulle

Utilisez la méthode **SetToolTip** du composant **ToolTip**.

On peut aussi le faire par code:

```
ToolTip1.SetToolTip(Button1, "Save changes")
```

Par code créons de toute pièce un ToolTip.

```
Dim toolTip1 As New ToolTip()  
  
' modifions les délais du ToolTip.  
toolTip1.AutoPopDelay = 6000  
toolTip1.InitialDelay = 2000  
toolTip1.ReshowDelay = 500  
' Force le ToolTip à être visible que la fenêtre soit  
active ou non .  
toolTip1.ShowAlways = True  
  
' donne le texte de l'info bulle à 2 contrôles.  
toolTip1.SetToolTip(Me.button1, "My button1")  
toolTip1.SetToolTip(Me.checkBox1, "My checkBox1")
```



Retour



Index



Sommaire



Suivant

LDF Cours : VB.net



4.14	Appel d'une API.		Suivant	Sommaire
------	-----------------------------	--	-------------------------	--------------------------

Les Api (Application Programming Interface) sont des bibliothèques de liaisons dynamiques (DLL, *Dynamic-Link Libraries*), se sont des fonctions (généralement écrites en C) et qui sont compilées dans une DLL.

Elles font :

- soit partie intégrante du système d'exploitation Windows. (API Windows)

Ce sont ces Api (Kernel32.Dll=coeur du système, User32Dll= fonctionnement des applications, gdi32..dll=interface graphique) que Windows utilise pour fonctionner.

Les fonctions sont donc écrites pour Windows; parfois on n'a pas d'équivalent VB, aussi, plutôt que de les réécrire quand on en a besoin, on appelle celles de Windows.

Elles permettent d'effectuer des tâches lorsqu'il s'avère difficile d'écrire des procédures équivalentes. Par exemple, Windows propose une fonction nommée FlashWindowEx qui vous permet de varier l'aspect de la barre de titre d'une application entre des tons clairs et foncés.

Il faut avouer que, le Framework fournissant des milliers de classes permettant de faire pratiquement tout ce que font les Api Windows, on a très peu à utiliser les Api Windows. Chaque fois que cela est possible, vous devez utiliser du code managé à partir du .NET Framework plutôt que les appels API Windows pour effectuer des tâches.

- soit partie de dll spécifiques fournies par des tiers pour permettre d'appeler des fonctions n'existant pas dans VB ni Windows.

Par exemple , il existe des Api MySql qui donnent accès

aux divers fonctions permettant d'utiliser une base de données MySQL. (Ces Api contiennent 'le moteur' de la base de données.)

Les Api sont en code non managé. De plus elles n'utilisent souvent pas les mêmes types de données que VB . L'appel des Api se faisant avec des passages de paramètres, il y a des précautions à prendre!! Sinon cela plante!!!cela plante vraiment.

Les Api Windows

L'avantage de l'utilisation d'API Windows dans votre code réside dans le gain de temps de développement, car elles contiennent des centaines de fonctions utiles déjà écrites et prêtes à être utilisées. L'inconvénient des API Windows est qu'elles peuvent être complexes à utiliser et implacables lorsqu'une opération se déroule mal.

Pour plus d'informations sur les API Windows, consultez la documentation du kit de développement Win32 SDK dans les API Windows du kit de développement Platform SDK. Pour plus d'informations sur les constantes utilisées par les API Windows, examinez les fichiers d'en-tête, tels que Windows.h, fournis avec le kit de développement Platform SDK. [MSDN donne aussi une description des Api](#)

Appels API avec Declare

La façon la plus courante d'appeler les API Windows consiste à utiliser l'instruction **Declare**.

Exemple (repris de chez Microsoft): appel de la fonction Windows 'MessageBox' qui est dans user32.dll et qui affiche une MessageBox.

• Rechercher de la documentation de la fonction:

[MSDN et les Api](#) donne la définition de la fonction MessageBox:

```
int MessageBox(  
    HWND hWnd,  
    LPCTSTR lpText,  
    LPCTSTR lpCaption,  
    UINT uType  
);
```

Parameters

hWnd

[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText

[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption

[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title **Error** is used.

uType

[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

Constantes API Windows : Vous pouvez déterminer la valeur numérique de des constantes utiliser dans les Api par l'examen des instructions **#define** dans le fichier WinUser.h. Les valeurs numériques sont généralement affichées au format hexadécimal. Par conséquent, vous pouvez les convertir au format décimal. Par exemple, si vous voulez combiner les constantes pour le style exclamation **MB_ICONEXCLAMATION** 0x00000030 et le style Oui/Non **MB_YESNO** 0x00000004, vous pouvez ajouter les nombres et obtenir un résultat de 0x00000034, ou 52 décimales.

Return Value

IDABORT	Abort button was selected.
IDCANCEL	Cancel button was selected.
IDCONTINUE	Continue button was selected.
IDIGNORE	Ignore button was selected.
IDNO	No button was selected.
IDOK	OK button was selected.
IDRETRY	Retry button was selected.

IDTRYAGAIN Try Again button was selected.
IDYES Yes button was selected.

¶ Il faut déclarer la procédure DLL

Ajoutez la fonction **Declare** suivante à la **section de déclaration** du formulaire de départ de votre projet ou à celle de la classe ou du module où vous voulez utiliser la DLL :

```
Declare Auto Function MBox Lib "user32.dll" _  
Alias "MessageBox" (ByVal hWnd As Integer, _  
ByVal txt As String, ByVal caption As String, _  
ByVal Typ As Integer) As Integer
```

Declare comprend les éléments suivants.

Le modificateur **Auto** indique de suivre les règles du Common Language Runtime.

Le nom qui suit **Function** est celui que votre programme utilise pour accéder à la fonction importée.

Le mot clé **Alias** indique le nom réel de cette fonction.

Lib suivi du nom et de l'emplacement de la DLL qui contient la fonction que vous appelez. Vous n'avez pas besoin d'indiquer le chemin d'accès des fichiers situés dans les répertoires système Windows.

Utilisez le mot clé **Alias** si le nom de la fonction que vous appelez n'est pas un nom de procédure Visual Basic valide ou est en conflit avec le nom d'autres éléments de votre application. **Alias** indique le nom réel de la fonction appelée.

Les types de données que Windows utilise ne correspondent pas à ceux de Visual Studio. Visual Basic effectue la plupart des tâches à votre place en convertissant les arguments en types de données compatibles, processus appelé *marshaling*. Vous pouvez contrôler de manière explicite la façon dont les arguments sont marshalés en utilisant l'attribut **MarshalAs** défini dans l'espace de noms **System.Runtime.InteropServices**.

Remarque Les versions antérieures de Visual Basic vous autorisaient à déclarer des paramètres **As Any** (tout type). Visual Basic .NET ne le permet pas.

Ajoutez des instructions Const à la section des déclarations de votre classe ou module pour rendre ces constantes disponibles pour l'application. Par exemple :

```
Const MB_ICONQUESTION = &H20L
Const MB_YESNO = &H4
Const IDYES = 6
Const IDNO = 7
```

Pour appeler la procédure DLL

```
DimRetVal As Integer ' Valeur de retour.

RetVal = MsgBox(0, "Test DLL", "Windows API MessageBox", _
                MB_ICONQUESTION Or MB_YESNO)
If RetVal = IDYES Then
    MsgBox("Vous avez cliqué sur OUI")
Else
    MsgBox("Vous avez cliqué sur NON")
End If
```

Visual Basic .NET convertit automatiquement les types de données des paramètres et valeurs de retour pour les appels API Windows, mais vous pouvez utiliser l'attribut **MarshalAs** pour indiquer de façon explicite les types de données non managés attendus par une API.

On peut aussi appeler une API Windows à l'aide de l'attribut DllImport mais c'est compliqué.

Autre exemple classique.

Utilisation de la routine BitBlt qui déplace des octets.

La documentation donne les renseignements suivants:

```
Declare Function BitBlt Lib "gdi32" ( _
    ByVal hDestDC As Long, _
    ByVal x As Long, _
```

```
ByVal y As Long, _
ByVal nWidth As Long, _
ByVal nHeight As Long, _
ByVal hSrcDC As Long, _
ByVal xSrc As Long, _
ByVal ySrc As Long, _
ByVal dwRop As RasterOps _
) As Long
```

Parameter Information

- `hdcDest`
Identifies the destination device context.
- `nXDest`
Specifies the logical x-coordinate of the upper-left corner of the destination rectangle.
- `nYDest`
Specifies the logical y-coordinate of the upper-left corner of the destination rectangle.
- `nWidth`
Specifies the logical width of the source and destination rectangles.
- `nHeight`
Specifies the logical height of the source and the destination rectangles.
- `hdcSrc`
Identifies the source device context.
- `nXSrc`
Specifies the logical x-coordinate of the upper-left corner of the source rectangle.
- `nYSrc`
Specifies the logical y-coordinate of the upper-left corner of the source rectangle.
- `dwRop`
Specifies a raster-operation code.

Les Constantes dwRop

```
' Copies the source bitmap to destination bitmap
SRCCOPY = &HCC0020
'
' Combines pixels of the destination with source bitmap using the Boolean
AND operator.
SRCAND = &H8800C6
'
' Combines pixels of the destination with source bitmap using the Boolean
XOR operator.
SRCINVERT = &H660046
'
```

' Combines pixels of the destination with source bitmap using the Boolean OR operator.

```
SRCPAINT = &HEE0086
```

' Inverts the destination bitmap and then combines the results with the source bitmap

' using the Boolean AND operator.

```
SRCERASE = &H4400328
```

' Turns all output white.

```
WHITENESS = &HFF0062
```

' Turn output black.

```
BLACKNESS = &H42
```

Return Values

If the function succeeds, the return value is nonzero.

Ici on va utiliser cette routine pour copier l'image de l'écran dans un graphics.

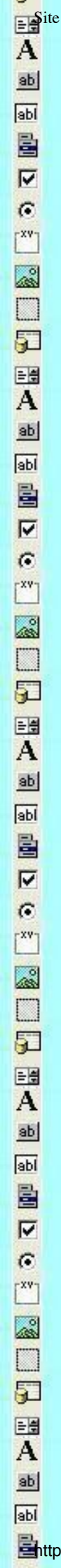
```
Private Declare Function BitBlt Lib "gdi32.dll" Alias  
"BitBlt" (ByVal _  
    hdcDest As IntPtr, ByVal nXDest As Integer, ByVal nYDest  
As _  
    Integer, ByVal nWidth As Integer, ByVal nHeight As  
Integer, ByVal _  
    hdcSrc As IntPtr, ByVal nXSrc As Integer, ByVal nYSrc As  
Integer, _  
    ByVal dwRop As System.Int32) As Long  
Dim memoryImage As Bitmap
```

```
Private Sub CaptureScreen()  
    Dim mygraphics As Graphics = Me.CreateGraphics()  
    Dim s As Size = Me.Size  
    memoryImage = New Bitmap(s.Width, s.Height, mygraphics)  
    Dim memoryGraphics As Graphics =  
Graphics.FromImage(memoryImage)  
    Dim dc1 As IntPtr = mygraphics.GetHdc  
    Dim dc2 As IntPtr = memoryGraphics.GetHdc
```

```
    BitBlt(dc2, 0, 0, Me.ClientRectangle.Width,  
Me.ClientRectangle.Height, dc1, 0, 0, 13369376)
```

```
    mygraphics.ReleaseHdc(dc1)  
    memoryGraphics.ReleaseHdc(dc2)  
End Sub
```

Le dernier paramètre a pour valeur= 13369376= SRCCOPY = &HCC0020 et correspond à 'Copies the source bitmap to destination bitmap'.



LDF Cours VB.net



4.15	Drag and Drop.	
------	----------------	--

L'exécution d'opérations **glisser-déplacer** (Drag and Drop) peut être ajoutée dans un programme.

La méthode [DoDragDrop](#) du contrôle de départ autorise la collecte des données au début de l'opération.

Les événements [DragEnter](#), [DragLeave](#) et [DragDrop](#) permettent de 'poser' les données dans le contrôle d'arrivée.

Exemple No 1 (simple)

Exemple: Le contrôle de départ est un contrôle [Button](#), les données à faire glisser sont la chaîne représentant la propriété `Text` du contrôle **Button**, et les effets autorisés sont la copie ou le déplacement. Le texte sera déposé dans un `textBox`:

Le contrôle de départ.

La fonctionnalité qui autorise la collecte des données au début de l'opération glisser dans la méthode [DoDragDrop](#).

L'événement [MouseDown](#) du **contrôle de départ** est généralement utilisé pour démarrer l'opération glisser parce qu'il est le plus intuitif (la plupart des glisser-déplacer commencent par un appui sur le bouton de la souris). Mais, souvenez-vous que n'importe quel événement peut servir à initialiser une procédure glisser-déplacer.

Remarque Les contrôles `ListView` et `TreeView`, ont un événement **ItemDrag** qui est spécifique.

```
Private Sub Button1_MouseDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles Button1.MouseDown
    Button1.DoDragDrop(Button1.Text, DragDropEffects.Copy Or
DragDropEffects.Move)
End Sub
```

Le premier argument indique les données à déplacer.

Le second les effets permis = copier ou déplacer.

Le contrôle d'arrivée

Toute zone d'un Windows Form ou d'un contrôle peut être configurée pour accepter les données déplacées en définissant la propriété AllowDrop et en gérant les événements DragEnter et DragDrop.

Dans notre exemple, c'est un contrôle TextBox1 qui est le contrôle d'arrivée.

```
TextBox1.AllowDrop =True. 'autorise le contrôle TextBox à recevoir
```

Dans l'événement **DragEnter** du contrôle qui doit recevoir les données déplacées

Vérifier que le type des données est compatible avec le contrôle d'arrivée (ici, vérifier que c'est bien du texte).

Définir ensuite l'effet produit lorsque le déplacement a lieu en lui attribuant une valeur de l'énumération DragDropEffects.(ici il faut copier)

```
Private Sub TextBox1_DragEnter(ByVal sender As Object, ByVal e As System.Windows.Forms.DragEventArgs) Handles TextBox1.DragEnter
    If (e.Data.GetDataPresent(DataFormats.Text)) Then
        e.Effect = DragDropEffects.Copy
    Else
        e.Effect = DragDropEffects.None
    End If
End Sub
```

Dans l'événement **DragDrop** du contrôle d'arrivée, utilisez la méthode **GetData** pour extraire les données que vous faites glisser.

```
Private Sub TextBox1_DragDrop(ByVal sender As Object, ByVal e As System.Windows.Forms.DragEventArgs) Handles TextBox1.DragDrop
    TextBox1.Text = e.Data.GetData(DataFormats.Text).ToString
End Sub
```

Exemple No 2 plus complexe:

Glisser déplacer **une ligne** d'une **listBox** 'ListBox1' vers une **listBox** 'ListBox2'.

Créer une ListBox1

Créer une listBox2 avec sa propriété AllowDrop=True 'listBox2 accepte le 'lâcher'

Dans l'en-tête du module ajouter:

```
Public IndexdInsertion As Integer ' Variable contenant l'index ou  
sera inséré la ligne
```

'Eventuellement pour l'exemple charger les 2 ListBox avec des chiffres pour pouvoir tester.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles MyBase  
Dim i As Integer  
For i = 1 To 100  
ListBox1.Items.Add(i.ToString)  
Next  
For i = 1 To 100  
ListBox2.Items.Add(i.ToString)  
Next  
End Sub
```

'Dans le listBox de départ, l'évènement MouseDown déclenche le glisser déplacer par DoDragDrop.

```
Private Sub ListBox1_MouseDown(ByVal sender As Object, ByVal e As  
System.Windows.Forms.MouseEventArgs) Handles ListBox1.MouseDown  
  
ListBox1.DoDragDrop(ListBox1.Items(ListBox1.IndexFromPoint(e.X, e.Y)),  
DragDropEffects.Copy Or DragDropEffects.Move)  
  
End Sub
```

'ListBox1.IndexFromPoint(e.X, e.Y) retourne l'Index de l'item ou se trouve la souris à partir des coordonnées e.x et e.y du pointeur)

'DoDragDrop a 2 arguments: l'élément à draguer et le mode

'DragOver qui survient quand la souris se balade sur le contrôle d'arrivé, vérifie si le Drop reçoit bien du texte et met dans IndexdInsertion le listItem qui est sous la souris.

'Noter que e.x et e.y sont les coordonnées écran , il faut les transformer en coordonnées client (du contrôle) par PointToClient afin d'obtenir l'index de l'item ou se trouve la souris (en utilisant IndexFromPoint.

```
Private Sub ListBox2_DragOver(ByVal sender As Object, ByVal e As  
System.Windows.Forms.DragEventArgs) Handles ListBox2.DragOver  
If Not (e.Data.GetDataPresent(GetType(System.String))) Then  
e.Effect = DragDropEffects.None  
Else  
IndexdInsertion = ListBox2.IndexFromPoint(ListBox2.PointToClient(New  
Point(e.X, e.Y)))  
e.Effect = DragDropEffects.Copy  
End If  
End Sub
```

'Enfin dans DragDrop, on récupère le texte dans Item et on ajoute un item après l'item pointé.

```
Private Sub ListBox2_DragDrop(ByVal sender As Object, ByVal e As
```

```
System.Windows.Forms.DragEventArgs) Handles ListBox2.DragDrop
Dim item As Object = CType(e.Data.GetData(GetType(System.String)),
System.Object)
ListBox2.Items.Insert(IndexdInsertion + 1, item)
End Sub
```



Site LDF Cours VB.net



Le débogage est la recherche des bugs. (voir 4-3 Traiter les erreurs)

Pour déboguer, il faut lancer l'exécution du programme, suspendre l'exécution à certains endroits du code et voir ce qui se passe puis faire avancer le programme pas à pas :

Pour démarrer et arrêter l'exécution, on utilise les boutons suivants :



On lance le programme avec le premier bouton, on le suspend avec le second, on l'arrête définitivement avec le troisième..

On peut suspendre (l'arrêter temporairement) le programme :

- avec le second bouton
- grâce à **des points d'arrêt** (pour définir un point d'arrêt en mode de conception, cliquez en face d'une ligne dans la marge grise: la ligne est surlignée en marron. Quand le code est exécuté, il s'arrête sur cette ligne marron).

```

For i= 1 To 6
  Tableau(i)=i*i

```

Next i

En plus si on clique sur le rond de gauche avec le bouton droit de la souris, on ouvre un menu permettant de modifier les propriétés de ce point d'arrêt (il y a la possibilité d'arrêter au premier ou au Xième passage sur le point d'arrêt, ou arrêter si une expression est à True ou à changé)

- en appuyant sur Ctrl-Alt-Pause
- en incluant dans le code une instruction `Stop`

Attention: Si vous utilisez des instructions `Stop` dans votre programme, vous devez les supprimer avant de générer la version finale.

Les transformer en commentaire :
' `Stop`

ou utiliser des instructions conditionnelles :
`#If DEBUG Then`
`Stop`
`#End If`

Déboguage

Quand le programme est suspendu, on peut **observer les variables, déplacer le point d'exécution**, on peut aussi faire marcher le programme **pas à pas (instruction par instruction)** et observer **l'évolution de la valeur des variables**, on peut enfin modifier la valeur d'une variable afin de tester le logiciel avec cette valeur.

F11 permet l'exécution pas à pas (y compris des procédures appelées: si il y a appel à une autre procédure, le pas à pas saute dans l'autre procédure)

F10 permet le pas à pas (sans détailler les procédures appelées: exécute la procédure appelée en une fois)

Maj+F11 exécute jusqu'à la fin de la procédure en cours.

On peut **afficher** ou **définir** l'instruction suivante, **exécuter jusqu'au curseur**, **insérer un point d'arrêt** ou un **espion** en **cliquant sur le bouton droit de la souris** et en **choisissant une ligne du menu**.



Espion express permet de saisir une expression (variable, calcul de variables) et de voir ensuite dans une fenêtre 'espion' les modifications de cet expression au cours du déroulement du programme.

On peut grâce au menu 'Débogage' puis 'Fenêtre' ouvrir les fenêtres:

Automatique, qui affiche les valeurs des variables de l'instruction en cours et des instructions voisines.

Immédiat ou il est possible de taper des instructions ou expressions pour les exécuter ou voir des valeurs.

Taper "?I" (c'est l'équivalent de "Print I" qui veut dire: afficher la valeur de la variable I) puis valider, cela affiche la valeur de la variable I.

Autre exemple, pour voir le contenu d'un tableau A(), tapez sur une seule ligne: "For i=0 to 10: ?i: Next i"

Enfin, il est possible de modifier la valeur d'une variable: Taper " I=10" puis valider, cela

modifie la valeur de la variable.

Espions permettant d'afficher le contenu de variables ou d'expressions.

Espions Express permet d'afficher la valeur de l'expression sélectionnée.

Points d'arrêts permet de modifier les propriétés des points d'arrêts. on peut mettre un point d'arrêt en cliquant dans la marge grise à gauche: l'instruction correspondante s'affiche en marron et l'exécution s'arrêtera sur cette ligne.

Me affiche les données du module en cours.

Variables locales affiche les variables locales.

Modules affiche les dll ou .exe utilisés.

Mémoire, Pile d'appels, Thread, Registres, Code Machine permettent d'étudier le fonctionnement du programme à un niveau plus spécialisé et technique.

Comment voir rapidement la valeur de propriétés ou de variables.

Il est toujours possible de voir **la valeur d'une propriété d'un objet** en la sélectionnant avec la souris:

Exemple on sélectionne `label1.Text` et on voit apparaître sa valeur.

```
'Dim MyNumber As Integer
Label1.Text = IsReference(MyArray) '
Label2.Text = Label1.Text = "1" & "e (myString) '
```

Pour les variables, il suffit que le curseur soit sur une variable pour voir la valeur de cette variable.

On peut aussi copier une expression dans la fenêtre 'immédiat', mettre un ? avant et valider pour voir la valeur de l'expression.

Attention à l'affichage:

Parfois en mode pas à pas on regarde le résultat d'une instruction dans la fenêtre du programme. Par exemple on modifie la propriété text d'un label et on regarde si le label à bien changé. Parfois la mise à jour n'est pas effectuée car **le programme met à jour certains contrôles seulement en fin de procédure**. pour palier à cela et afficher au fur et à mesure, même si la procédure n'est pas terminée, on utilise la méthode [Refresh](#) de l'objet qui 'met à jour'.

Exemple:

```
Label1.text="A"
```

```
Label1.Refresh
```

Cela ne semble pas toujours fonctionner. Avez-vous une explication?

Objet Console

On peut écrire sur la fenêtre console, quand on a parfois besoin d'afficher des informations, mais **uniquement pour le programmeur**:

```
Console.WriteLine( myKeys(i))
```

Mais dans un programme Windows, il n'y a pas de console!! la sortie est donc envoyée vers la fenêtre de sortie (voir Debug)(Menu Affichage>Autres fenêtres>Sortie pour voir la fenêtre)

Autre exemple:

```
Dim amis() As String = {"pierre", "jean", "jacques",  
"toto"}  
For Each nom As String In amis  
Console.Out.WriteLine(nom)  
Next
```

Objet Debug

L'espace de noms **Systems.Diagnostics** est nécessaire.

Pour déboguer du code, on a parfois besoin d'afficher des informations, mais **uniquement pour le programmeur**, en mode debug afin de suivre le cheminement du programme ou la valeur d'une variable ou si une condition se réalise; pour cela on utilise une fenêtre nommée 'Sortie'(Output). (Menu Affichage>Autres fenêtres>Sortie)

Pour écrire dans la fenêtre Output (**sans arrêter le programme**):

▫ Du texte:

```
Debug.Write(Message)
```

et pour ajouter un passage à la ligne:

```
Debug.WriteLine(Message)
```

▫ Le contenu d'une variable:

```
Debug.Write(Variable)
```

▫ Les propriétés d'un objet:

```
Debug.Write(Objet)
```

Exemple:

```
Debug.Write("ça marche")      'Affiche 'ça marche'
```

```
Dim A as Integer=2
```

```
Debug.Write(A)      'Affiche 2
```

```
Debug.Write(A+2)    'Affiche 4
```

On voit que s'il y a une expression, elle est évaluée.

On peut aussi afficher un message **si** une condition est remplie en utilisant **WriteLineIf** ou **WriteIf**:

```
Debug.WriteLineIf(i = 2, "i=2")
```

```
Affiche 'i=2' si i=2
```

Cela vous permet, **sans arrêter le programme** (comme le fait Assert), d'être informé quand une condition est vérifiée.

Debug.Assert par contre affiche une fenêtre Windows et

stoppe le programme si une assertion(une condition) passe à **False**.

```
Debug.Assert(Assertion)
```

```
Debug.Assert(Assertion, Message1)
```

```
Debug.Assert(Assertion, Message1, Message2)
```

L'exemple suivant vérifie si le paramètre 'type' est valide. Si le type passé est une référence null (Nothing dans Visual Basic), Assert ouvre une boîte de dialogue nommé 'Echec Assertion' avec 3 boutons 'Abandonner', 'Recommencer' 'Ignorer'.. La liste des appels est affichée dans la fenêtre (procédure en cours en tête de liste, module et numéro de ligne en première ligne)

```
Public Shared Sub UneMethode (type As Type, Typedeux As Type)
```

```
Debug.Assert( Not (type Is Nothing), "Le paramètre Type est=Nothing ", "Je ne peux pas utiliser un Nothing")
```

```
....
```

```
End Sub UneMethode
```

```
□
```

```
Debug.Fail
```

Fait pareil mais sans condition.

Trace:

Trace possède les mêmes fonctions que Debug (Write, WriteIf, Assert, Fail..) mais la différence est que Trace permet d'afficher à l'utilisateur final par défaut.

Trace est activé par défaut. Par conséquent, du code est généré pour toutes les méthodes **Trace** dans les versions release et debug. Ceci permet à un utilisateur final d'activer le traçage pour faciliter l'identification du problème sans que le programme ait à être recompilé. Par opposition, Debug est désactivé par défaut dans les versions release, donc aucun code exécutable n'est généré pour les méthodes **Debug**.



Retour



Index



Sommaire



Suivant

Site **LDF** Cours  **VB.net**

D 1.1	Comprendre le FrameWork.	Suivant 	 Sommaire
--------------	---	--	---

Comment fonctionne un programme VB.NET ?

Le Framework.NET

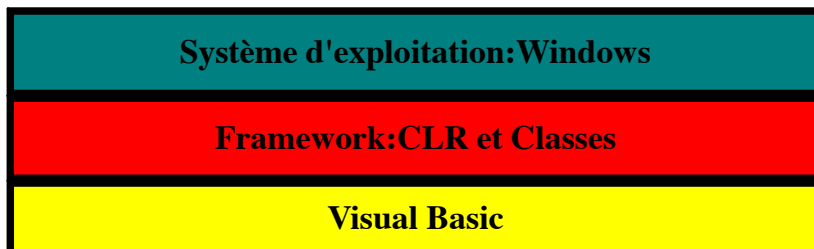
Pour qu'un programme fonctionne, il faut installer le **Framework.NET**:

C'est une **plate-forme informatique**, une couche entre Windows et l'application VB.

Cette infrastructure offre un vaste accès à :

1. **l'ensemble du système d'exploitation.**
2. **une collection d'objets utilisables pour créer des programmes.**
3. **des routines d'exécution de programme.**

de manière homogène et très fiable.



L'exécutable en Visual Basic

1. **utilise les objets** (WindowsForms, Contrôles, type de variable,..)du Framework.
2. **appelle les fonctions** (exécution, affichage, gestion de la mémoire, lecture dans une base de donnée..)du Framework.

A la limite on peut considéré le Framework comme une machine virtuelle (comme celle de Java). Il suffirait de porter le Framework sous Linux pour que les programmes VB fonctionnent. (Info? Intox?)

Inconvénients?

La couche supplémentaire ralentie le programme?

A peine, cela serait insignifiant.

Et s'il y a une nouvelle version du Framework?

Les versions successives devront être compatible ascendante et descendante!!

Intérêts?

On installe une seule fois le Framework.

Une fois le Framework installé, il suffit pour installer un nouveau programme de n'installer que l'exécutable.

On peut utiliser plusieurs langages.

Nous appelons les fonctions du Framework avec Visual Basic mais on peut aussi le faire avec C# et 30 autres langages. La vitesse d'exécution sera la même, les types de variables, les objets seront les mêmes. Plus de problèmes de transfert de paramètres.

Il est même possible de faire cohabiter plusieurs langages dans une même application.

Le code est homogène.

Plus de bidouille, de ficelle de programmation pour contourner les lacunes du langage et l'accès limité au système d'exploitation: Les milliers de Classes du Framework donne accès à une multitude de fonctions et aux services du système d'exploitation, cela nativement.

Revoyons en détails le contenu du Framework:

Il contient deux composants principaux :

- **Le Common Language Runtime.** Le runtime peut être considéré comme un agent qui manage le code au moment de l'exécution, qui l'exécute, fournit des services essentiels comme la gestion de la mémoire, la gestion des threads, et l'accès distant.
- **La bibliothèque de classes du .NET Framework.** C'est une collection complète orientée objet, de types réutilisables que vous pouvez utiliser pour développer des applications allant des traditionnelles applications à ligne de commande

ou à interface graphique utilisateur (GUI, *Graphical User Interface*) jusqu'à des applications qui exploitent les dernières innovations fournies par ASP.NET, comme les services Web XML et Web Forms.

Par exemple, les classes Windows Forms sont un ensemble complet de types réutilisables qui simplifient grandement le développement Windows. Si vous écrivez une application Web Form ASP.NET, vous pouvez utiliser les classes Web Forms.

Il existe un éventail de tâches courantes de programmation y compris des tâches comme la gestion de chaînes, la collection de données, la connectivité de bases de données, et l'accès aux fichiers.

Il y a 3300 Classes!!

Plus d'appel aux Api Windows, on fait tout directement en utilisant les classes du Framework.

Code managé:

Le code écrit pour le Framework est dit managé (ou géré): il bénéficie des avantages du Framework: gestion de la mémoire, optimisation de cette mémoire, règles communes, utilisation de plusieurs langages..

Les anciens composants COM sont utilisables mais non managés.

L'interopérabilité entre les codes managés et non managés permet aux développeurs de continuer à utiliser des composants COM et des DLL nécessaires.

Compilation:

Lors de la génération du projet, un code intermédiaire est produit (IL: Intermédiaire Langage), ce code est commun à tous les langages. Lors de la première exécution, le code est compilé en binaire. Les exécutions suivantes seront plus rapides.

Installation:

Les applications et contrôles écrits pour le .NET Framework imposent que celui-ci soit installé sur l'ordinateur où ils s'exécutent. Microsoft fournit un programme d'installation redistribuable, **Dotnetfx.exe**, qui contient les composants Common Language Runtime et .NET Framework nécessaires à l'exécution des applications .NET Framework.

Pour installer le Framework, il faut au moins:

• Microsoft® Windows® 98

- Microsoft® Windows® 98 Deuxième Édition
- Microsoft® Windows® Millennium Edition (Windows Me)
- Microsoft® Windows NT 4 (Workstation ou Server) avec le Service Pack 6a
- Microsoft® Windows® 2000 (Professionnel) avec MAJ.
- Microsoft® Windows® XP (Édition familiale ou Professionnel)
- Famille Microsoft® Windows® Server 2003

Où le trouver:

Dans le **MSDN Download Center** sur Internet:

<http://msdn.microsoft.com/downloads/> puis

<http://www.microsoft.com/downloads/details.aspx?FamilyID=262d25e3-f589-4842-8157-034d1e7cf3a3&displaylang=fr>

(ou <http://msdn.microsoft.com/downloads/list/netdevframework.asp>)

Choisir autre langue=Français puis **charger** le **FrameWork 1.1 Français (23.6 Mo)** puis **lancer l'installation avec Dotnetfx.exe.**

(Bien choisir la version française car on ne peut pas installer plusieurs versions de langue différente)

Pour développer il faut ensuite installer Visual Studio.Net.

L'utilisateur final de l'exécutable installera le Framework et l'exécutable (programme simple).

Où est le Framework?

dans:

c:\Windows\Microsoft.NET\Framework\v1.1 4322\

On voit les DLL qui composent le Framework: System.dll, System.Drawing.dll..

La version 1.1 fait 73 Mo sur le disque

On peut installer à côté la version 1.0; Pourquoi installer les 2 si il y a compatibilité ascendante? c'est une question!!

Nouvelle version 2:

VisualStudio 2005 utilise le Framework 2.0, bien sur il y a un SDK 2.0



Ce sont des versions beta. (11/2004)

 Retour

 Index

 Sommaire

Suivant 

D 1.2	Distribution d' une application.	Suivant 	 Sommaire
--------------	---	--	---

Comment distribuer un programme VB.NET ?

Programme simple

Le développeur doit **compiler** son application, la **générer**,

- Il indique les propriétés du programme à générer: Menu Projet->Propriété du projet.

- Génération en **mode Release** (et pas en mode Debug)
- On peut choisir de générer ou non des **informations de débogage**.

- Puis, il génère en utilisant le Menu Générer-> Générer la Solution

Le programme exécutable ainsi créé se trouve dans le répertoire \bin.

Chez l'utilisateur:

- Il faut installer le **Framework.NET**:
- **Copier l'exécutable** (et éventuellement les fichiers de données) dans un répertoire.

Pour utiliser le programme, l'utilisateur lance l'exécutable.

Créer un programme d'installation automatique:

Il est bien sur plus simple de créer un **programme d'installation automatique**:

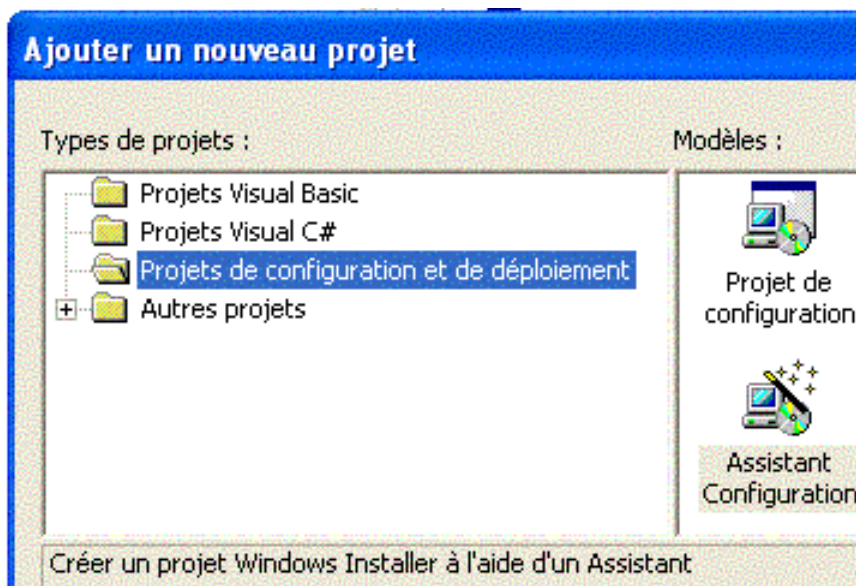
L'utilisateur lance Setup.exe qui est sur un cd d'installation et ce programme installe le logiciel.

L'assistant à la création du programme d'installation est **dans** VB:

Pour cela il faut créer un projet de configuration et déploiement, en modifier certaines propriétés puis le générer.

Menu Fichiers->Ajouter un projet->Nouveau Projet-> Cliquez dans la liste sur 'Projet de configuration et de déploiement.' puis sur l'icône 'Assistant de configuration'.

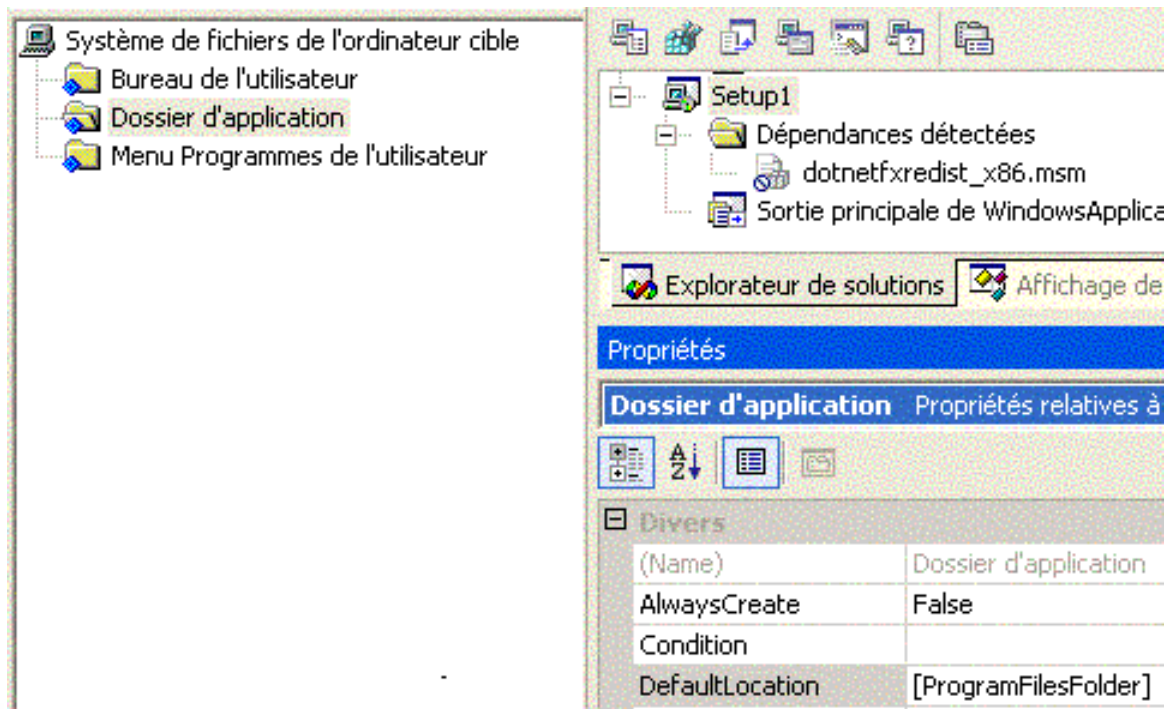
Il faut vérifier en bas de la fenêtre 'Ajouter un nouveau projet' le chemin.



Suivez les divers écrans en vous rappelant que vous utiliser une application Windows en sortie principale, n'oubliez pas de rajouter si nécessaire certains fichiers (les fichiers de données nécessaires).

Après le bouton 'Terminez', il est ajouté dans la fenêtre de l'explorateur de solution une ligne nommé par défaut 'Setup1' correspondant au projet de l'installateur. Il est crée un onglet 'Système de fichiers' dans la fenêtre principale.

Vous venez de créer votre projet de configuration et déploiement, vous pouvez maintenant le modifier.

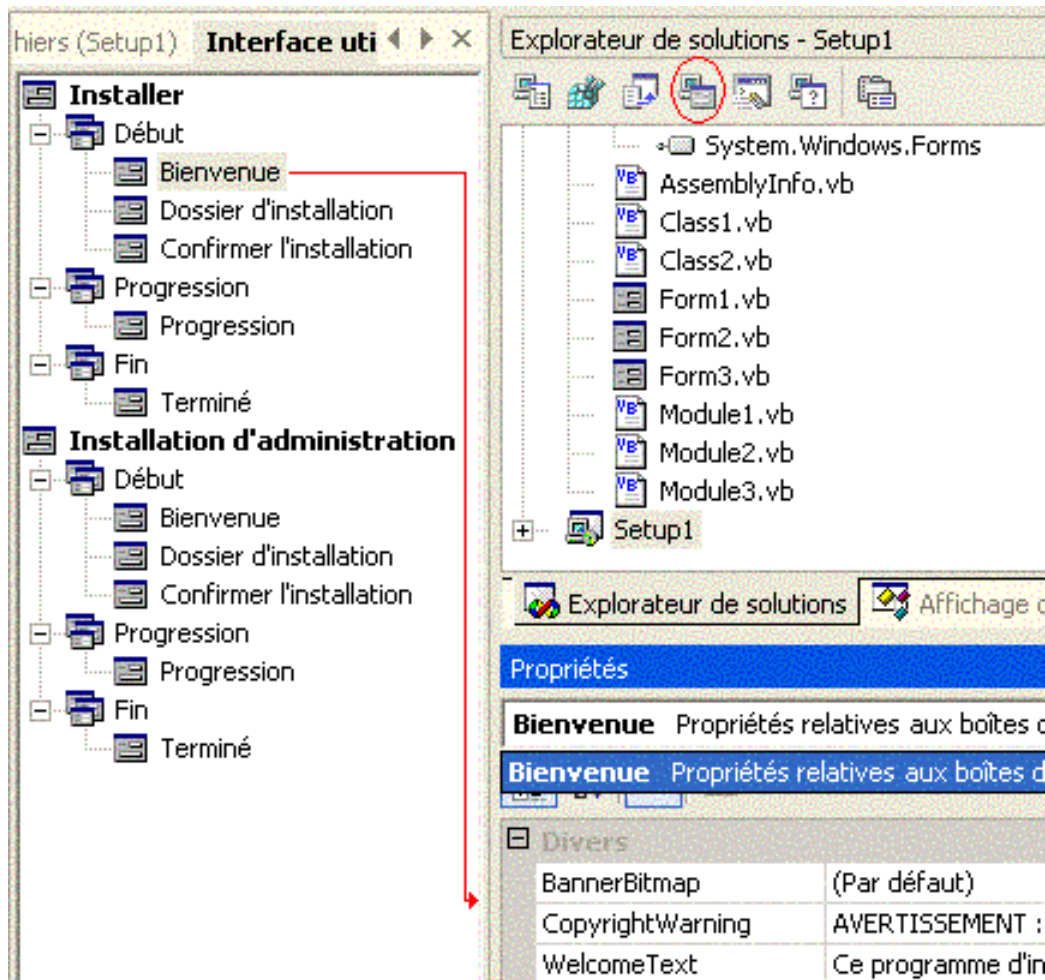


Le fait de cliquer sur le 'dossier d'application' ou sur sur ligne dans l'explorateur de solution affiche dans la fenêtre de propriétés, les propriétés de l'installation.

La propriété **DefaultLocation** donne par exemple l'emplacement, le répertoire d'installation. Il y a bien d'autres propriétés permettant de personnaliser votre installateur (Auteur, nom de l'entreprise, Version...)

Enfin quand on clique sur Setup1 dans l'explorateur de solutions, il apparaît des boutons donnant accès à des éditeurs de registre, de d'interface de l'installateur, de condition de lancement..

Si on clique sur le 3eme bouton on ouvre **l'éditeur d'interface** qui donne accès au déroulement de l'installateur. En cliquant sur la première fenêtre ('Bienvenue') on a accès aux propriétés de cette fenêtre: texte, image..



Pour créer effectivement l'installateur il faudra enregistrer puis utiliser le Menu Générer-> Générer Setup1.

Un répertoire nommé dans notre exemple 'SeptUp1' est créée; il contient

Setup.exe

Setup1.msi

Setup.ini

il suffit de mettre ces fichiers sur un cd et de le donner à l'utilisateur final qui installera votre logiciel en lançant Setup.exe.

Le logiciel d'installation vérifie si le Framework est bien installé.

Les assembly

Pour les installations de programme , mises à jour, utilisation de composants propres au programme ou partagés avec d'autres programmes; pour gérer les versions, éviter les problèmes de conflit de composants, on utilisa les **assembly** (ou **assemblage**).

Dans l'explorateur de solution, double-cliquer sur

Site
Assemblyinfo.vb, dans la fenêtre principale s'ouvre permettant d'avoir accès à certaines données.



Site **LDF** Cours **VB.net**



E 4.1	Exemples : Horloge.	Suivant 	 Sommaire
------------------------	--------------------------------------	--	---

Très Simple: Comment créer une horloge numérique?

15:21:45

Dans la fenêtre Form1.

Mettre un Timer (Timer1)

Mettre un label (Label1)



Ajouter le code

```
Private Sub Form1_Load(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Timer1.Interval = 1000    'Timer1_Tick sera déclenché  
    toutes les secondes.  
  
    Timer1.Start()           'On démarre le Timer  
  
End Sub  
  
Private Sub Timer1_Tick(ByVal sender As Object, ByVal e As  
System.EventArgs) Handles Timer1.Tick  
  
    Label1.Text = Now.ToLongTimeString    'Affiche l'heure  
    format long.
```

[End](#) [Sub](#)

Simple!! non!!



5.1	Programmation Orientée Objet =POO.	Suivant 	 Sommaire
------------	---	--	---

VB.NET permet maintenant de faire de la POO (Programmation Orientée Objet) à part entière :

Les Classes prédéfinies déjà existantes dans le Framework, obéissent à la POO, les classes que l'on crée soi même dans des modules de Classe suivront elles aussi les règles de la POO :



Pour ce chapitre, nous sommes du côté de l'application utilisatrice des objets (et non dans les objets).

Interface et implémentation:

Nous savons déjà:

On utilise une Classe (le moule) pour instancer (créer) un objet.

Une classe est une combinaison de code et de données.

- Le code et la définition des données constituent l'**implémentation**.
- L'**interface** de l'objet est l'ensemble de ses membres visibles et utilisables (les membres se sont les propriétés, les méthodes, les évènements).

Exemple:

Prenons un objet d'une classe `ListBox`:

- L'**interface** c'est `ListBox.Visible` `ListBox.AddItem...` Je la vois , je peux l'utiliser.
- L'**implémentation**, je ne la vois pas, c'est le code qui gère la `ListBox`, la définition des éléments; c'est une 'boite noire', je ne sais pas ce qui s'y passe, je n'y est pas accès, et c'est tant mieux!!!

Encapsulation:

Le fait de ne pas voir l'implémentation (le code), c'est l'**encapsulation**.

Le code, les définitions de données sont privés à l'objet et non accessibles, ils sont enfermés, encapsulés dans une boite noire.



L'encapsulation permet donc d'exposer aux applications clientes uniquement l'interface.

Les applications clientes n'ont pas à se soucier du fonctionnement interne.

Cela a une conséquence, si je modifie le code mais pas l'interface, l'application cliente n'a pas à être modifiée.

Héritage:

On a vu d'un objet issu d'une Classe hérite des membres de la classe parent, cela crée une relation mère/fille (parent/enfant); la classe fille pouvant réutiliser les membres de la classe mère.

A noter qu'une classe ne peut hériter que d'une classe en VB.NET.

La Classe fille peut utiliser **les membres de la classe mère**, mais aussi **ajouter ses propres membres** ou **redéfinir certains membres** de la classe mère.

Exemple:

On a vu que quand on dessine une Form1, cela crée une Classe 'Form1' qui hérite des Windows.Forms ([Inherits System.Windows.Forms.Form](#))

Polymorphisme:

Le nom de *polymorphisme* signifie *qui peut prendre plusieurs formes*.

Il y a **3 sortes de polymorphisme**:

- **Le polymorphisme de surcharge** ou en anglais *overloading*.

Le polymorphisme de surcharge permet d'avoir des **fonctions de même nom, avec des fonctionnalités similaires, dans des classes sans aucun rapport entre elles**. Par exemple, la classe Texte, la classe image et la classe Titre peuvent avoir une fonction affiche.

- **Le polymorphisme d'héritage** (*redéfinition, spécialisation* ou en anglais *overriding*)

C'est la forme la plus naturelle du polymorphisme: elle permet d'appeler la méthode d'un objet sans devoir connaître son type.

- **Le polymorphisme paramétrique** (également *généricité* ou en anglais *template*)

Le polymorphisme paramétrique, représente la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type).

Ainsi, on peut par exemple définir plusieurs méthodes `add()` effectuant une somme.

- La méthode `int add(int, int)` retourne la somme de deux entiers
- La méthode `char add(char, char)` retourne la somme de deux caractères
- La méthode `int add(int, int, int)` retourne la somme de trois entiers

On appelle *signature* chaque combinaison d'arguments d'une fonction (combinaison en nombre et en type). Une fonction a donc autant de signature que de manière d'appeler cette fonction C'est donc la signature d'une méthode qui détermine quel code sera appelé.

Constructeur, destructeur:

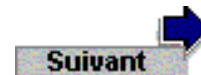
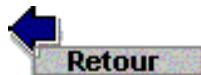
Un **constructeur** est une fonction effectuée **lors de l'instanciation** d'un objet de la Classe; il sert généralement à 'initialiser' l'objet.

Souvent il y a plusieurs signatures.

Exemple: pour créer un objet graphique `Point`, j'utilise un constructeur permettant de définir les coordonnées du point:



```
Dim P As New Point(45, 78)
```

La **destruction** d'un objet est effectuée lorsqu'on lui affecte la valeur `Nothing` ou lorsqu'on quitte la portée où il a été défini.



Site **LDF** Cours **VB.net**



5.2	Module de Classe.	 Suivant	 Sommaire
------------	----------------------------------	---	--

On a vu qu'il existait des classes prédéfinies (celle du Framework par exemple) mais on peut soi même **CREER SES PROPRES CLASSES**:

Prenons un exemple:

On va créer une classe 'Médicament' (c'est l'objet de ce chapitre).

Une fois cette classe créée, du coté utilisateur, comment l'utiliser?

Pour instancer un objet Medicament: **Dim M As
New Medicament**

Donner une valeur à une propriété: **M.Nom=
"Aspirine": M.Laboratoire="RP"**

Récupérer la valeur d'une propriété:
LeNom=M.Nom : Nb= M.NombreMedicament

Mais il faut la créer avant cette Classe 'Médicament':
C'est l'objet de ce chapitre.



Pour la suite de ce chapitre, nous sommes **DANS la Classe de l'objet (et non dans l'application utilisatrice)**.

Créer une Classe

Menu **Projet** puis **Ajouter une Classe**.

Entrée 'Medicament' dans la zone nom puis OK

Une nouvelle fenêtre de module est ajoutée à notre projet, contenant le code suivant:

```
Public Class Medicament  
...  
End Class
```

Le mot avant Class indique la portée de la classe:

Public (Classe instancable dans et hors du projet, utilisable par un autre programme)

Private (Classe instancable que dans elle même!!)

Friend (Classe instancable dans le projet)

Protected (Classe instancable uniquement dans les classes dérivées)

On peut ajouter:

MustInherit: Cela donne une classe non instancable, on ne peut pas créer d'objet avec!! Alors à quoi cela sert!! A fournir une base pour des classes qui en hériteront.

NotInheritable: Cette classe ne peut-être héritée.

Ajouter des variables dans le module de classe:

La variable peut être 'Privée' et non visible à l'extérieur:

Private mNom As String

Elle peut être 'Public' , visible à l'extérieur et donc non encapsulée.

Public Laboratoire As String

(A l'extérieur , si on instance M comme un médicament, M.laboratoire sera valide)

On peut définir un champ 'Public' en lecture seule qui a une valeur constante:

Public ReadOnly NombreMedicament=2000

Vous pouvez ajouter à votre variable : **Shared**. Cela signifie que la variable déclarée comme Shared est **partagée par toutes les instances de la classe :**

Public Shared Societe as String

La propriété Societe de toutes les instances de Medicament aura la même valeur (alors que chaque instance aura sa propre propriété Nom.)

Ajouter des membres grâce à 'Property'

Il faut ajouter des propriétés et des méthodes (les membres) à la Classe.

Tapez 'Property Nom' puis validez, la définition de la propriété est générée en faisant apparaître

- un bloc **Get** qui correspond à la lecture de la propriété par l'utilisateur.
- un bloc **Set** qui correspond à l'écriture de la propriété par l'utilisateur, on récupère la valeur dans le paramètre value.

```
Property Nom()
```

```
Get
```

```
End Get
```

```
Set (By Val Value)
End Set
End Property
```

J'ajoute **Public** pour que cette propriété soit accessible hors de la classe, j'indique que c'est une **String**. Lors de la lecture de la propriété (par l'utilisateur de l'instance) **Get** retourne (grâce à Return) la valeur mNom qui est une variable privée de la classe et qui sert à stocker la valeur de la propriété. Lors de l'écriture de la variable, **Set** récupère la valeur dans Value et la met dans mNom:

```
Public Property Nom() as String
Get
    Return mNom
End Get
Set(By Val Value)
    mNom=value
End Set
End Property
```

Encapsulation:

La propriété '**Nom**' est encapsulée: l'utilisateur qui utilise une instance de la classe, ne voit pas ce qui se passe (ici c'est très simple) quand il lit la propriété Nom, **il ne voit pas l'implémentation** (l'implémentation c'est Get...Set...), **il ne voit que l'interface** c'est à dire .Nom

Une propriété peut être en **lecture seule**:

```
Public ReadOnly Property InitialeNom() As String
Get
    Return Left(mNom,1)
End Get
End Property
```

□
Mais aussi en **écriture seule** grâce à WriteOnly.

Les propriétés comme les méthodes peuvent être Public, Private, Protected, Friend, ProtectedFriend:

Vous pouvez ajouter à votre membre : **Shared**. Cela signifie que la propriété ou la méthode déclarée comme Shared est **partagée par toutes les instances de la classe**.

Constructeur:

Un constructeur est une procédure exécutée lors de

l'instanciation.

Dans le module de classe, elle est définie par:

```
Sub New()
```

```
End sub
```

On peut ajouter des paramètres qui serviront à instancer.

Par exemple pour instancer et donner le nom en même temps:

```
Sub New(ByVal LeNom As String)
```

```
    nNom=LeNom
```

```
End sub
```

Cela permet `Dim M As New Medicament("Aspirine")`

On peut définir plusieurs constructeurs avec un nombre de paramètres différents (plusieurs signatures), dans ce cas il y a **surcharge**, le constructeur par défaut étant celui sans paramètres.

Donnons un autre exemple: je veux savoir combien il a été créé d'instance de 'Médicament':

Créons une variable commune à toutes les instances `Private Shared Nb as Integer=0`

Le constructeur va l'incrémenter à chaque instanciation:

```
Sub New()
```

```
    Nb+=1
```

```
End sub
```

Il suffit de lire sa valeur pour savoir le nombre d'objet Medicament qui existent:

```
Public ReadOnly Property NbInstance()
```

```
    Get
```

```
        NbInstance=Nb
```

```
    End Get
```

```
End Property
```

Destructeur:

Il est toujours préférable de détruire un objet qui n'est

plus utilisé en lui affectant la valeur Nothing. L'objet est aussi détruit si on sort de sa portée.

Comme on le fait avec New on peut utiliser une procédure **Finalize** qui est automatiquement appelée quand l'objet est détruit.

```
Protected Overrides Sub Finalize ()  
    Ens Sub
```

On peut y mettre le code libérant les ressources ou d'autres choses.

Noter que la procédure Finalize est ici la redéfinition (d'ou 'Overrides') de la procédure Finalize (qui est Overridable) de la Classe Objet.

Attention la méthode Finalize est exécutée quand l'objet est réellement détruit (Objet=Nothing le rend inutilisable mais il est toujours présent). C'est parfois très tardivement que l'objet est détruit: quand il y a besoin de mémoire (c'est le Garbage Collector qui entre en action) ou à la sortie du programme.

Pour forcer la destruction on peut utiliser l'interface **IDisposable:**

```
Il faut mettre dans l'entête de la classe  
Implements IDisposable
```

et mettre dans le code de la classe

```
Public Sub Dispose() Implements System.IDisposable.Dispose  
End sub
```

C'est une méthode Public, on peut l'appeler de l'application cliente:

```
M.Dispose()  
M=Nothing
```

Dans la pratique **quelle est d'utilité de gérer la destruction** autrement que par Objet=Nothing si le Garbage Collector nettoie la mémoire? C'est une question.

Réponse donnée par Microsoft:

Votre composant a besoin d'une méthode **Dispose** s'il alloue des objets système, des connexions à la base de données et d'autres ressources rares qui doivent être libérées dès qu'un utilisateur a fini de se servir d'un composant.

Vous devez également implémenter une méthode **Dispose** si votre composant contient des références à d'autres objets possédant des méthodes **Dispose**.

Délégation:

Si dans le module de classe j'ai besoin de connaître le nombre d'instance de médicament, je peux, bien sur, regarder la valeur de Nb mais je peux aussi utiliser `Me.NbInstance` ; cela revient à utiliser un membre de la classe au sein d'une classe; cela s'appelle une délégation.

Surcharge

On peut **surcharger un constructeur**.

Pour cela il suffit de rajouter autant de procédure New que l'on veut avec pour chacune un nombre de paramètre différent (signatures différentes).

Exemple surchargeons un constructeur:

```
Class Figure
Sub New()
    Bla Bla
End Sub
```

```
Sub New( ByVal X As Integer, ByVal Y As Integer)
    Blabla
End Sub.
End Class
```

On peut donc instancer la classe correspondante de 2 manières:

```
Dim A As New Figure[] 'Constructeur par défaut
```

ou

```
Dim A As New Figure(100,150)
```

On peut **surcharger une property**.

Pour cela il suffit de rajouter des procédure Property **ayant le même nom de méthode** avec pour chacune un nombre de paramètre différent (signature différente)

On peut ajouter **Overloads** mais c'est facultatif.

Exemple surchargeons un membre:

```
Class Figure
Public Overloads Property Calcul()  
    Bla Bla
End Sub
```

```
Public Overloads Property Calcul( ByVal X As Integer,  
ByVal Y As Integer)  
    Blabla
End Sub.
End Class
```

Evènement :

On peut définir un évènement pour la classe.

Dans la classe:

- Il faut **déclarer l'évènement**, avec le mot `Event`
- Il faut utiliser l'instruction `RaiseEvent` pour le déclencher lorsqu'un état ou une condition le nécessite.

Exemple: je crée une classe nommée 'Class1' qui contient un membre 'Texte' (Property Texte), si 'Texte' change, alors on déclenche l'évènement `TextChanged`:

```
Public Class Class1
    Private mTexte As String
    ' Déclare un évènement
    Public Event TextChange(ByVal UserName As String)
    Public Property Texte()
        Get
            Return mTexte
        End Get
        Set(ByVal Value)

            If Value <> mTexte Then
                RaiseEvent TextChange("hello")
            End If
            mTexte = Value
        End Set
    End Property
End Class
```

Si l'application cliente modifie la propriété `.Texte` d'un objet `Class1` alors on compare l'ancien et le nouveau texte, s'il est différent on déclenche un évènement `TextChanged`.

Dans l'application cliente:

- Il faut définir **dans la partie déclaration** l'objet `M` de classe `Class1` en indiquant qu'il gère des évènements.

```
Private WithEvents M As Class1
```

- q Dans Form_Load par exemple il faut instancer l'objet:

```
M= New Class1()
```

- q Il faut écrire la procédure évènement avec son code:

```
Private Sub M_TexteChange(ByVal v As String)  
Handles M.TextChanged  
    MsgBox("le texte a changé")  
End Sub
```

Ici on demande simplement quand le texte change, d'ouvrir une MessageBox.

Lors de l'utilisation:

`M.Text="Nouveau text"` déclenche la Sub `M.TextChange` qui dans notre exemple simple ouvre une MessageBox indiquant que le texte à changer.

On remarque que la Classe définit l'évènement, la procédure correspondant à l'évènement est dans l'application cliente. (De la même manière que quand on clique sur un objet bouton cela déclenche la procédure `Bouton-Click`.)

Héritage

Une classe peut hériter d'une autre classe, il suffit d'utiliser `: Inherits Nom de la classe mère'`

La classe fille possède tous les membres de la classe mère.

Il est possible en plus de **redéfinir un des membres** (de créer une nouvelle définition du membre dans la classe fille et uniquement pour cette classe fille) pour que cela marche il faut que le membre de la classe mère soit modifiable (**overridable**) et que le membre de même nom de la classe fille soit modifié (**Overrides**)

Créons une Classe `Salarié1` avec une méthode 'Salaire annuel sur 13 mois'

```
Class Salarié1
```

```
Public Overridable ReadOnly Property SalaireAnnuel() As
```

Integer

Get

```
SalaireAnnuel = SalaireMensuel * 13
```

```
End Get
```

```
End Property
```

```
End Class
```

Créons maintenant une classe Salarié2 qui hérite de toutes les propriétés de la classe salarié1 mais donc la méthode SalaireAnnuel est sur 12 mois:

```
Public Class Salarié2
```

```
Inherits Salarié1
```

```
Public Overrides ReadOnly Property SalaireAnnuel() As
```

```
Integer
```

```
Get
```

```
SalaireAnnuel = SalaireMensuel * 12
```

```
End Get
```

```
End Property
```

```
End Class
```



Site **LDF** Cours **VB.net**



5.3

Créer un composant.

[Suivant](#)
[Sommaire](#)

On a vu qu'on pouvait **CREER SES PROPRES CLASSES** dans un projet, mais on peut aussi:

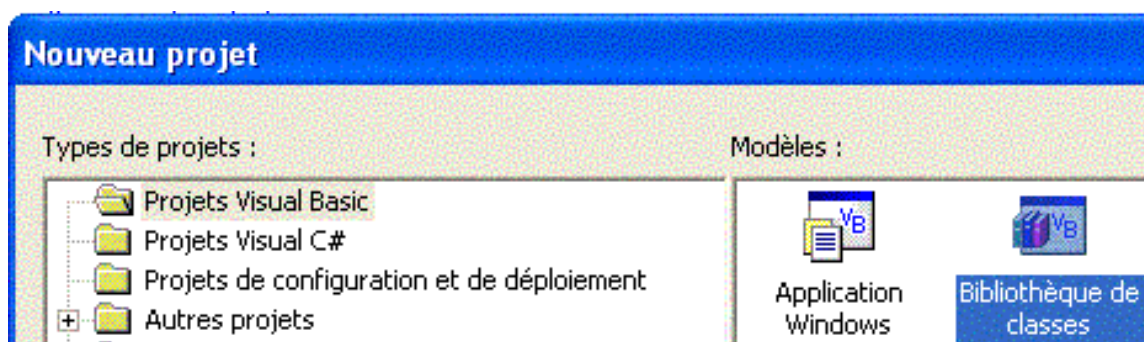
- **créer une Classe autonome** qui sera utilisée par plusieurs autres programmes, c'est une **Bibliothèque de Classe**.
- **créer un contrôle utilisateur** utilisé lui aussi par d'autres programmes.

Jusqu'a présent , on chargeait et utilisait, dans notre application cliente, **des composants** (classe ou contrôles) **tout faits**.

Maintenant nous allons créer ces classes ou contrôles, ils seront utilisés par l'application cliente.

Bibliothèque de Classe:

Pour créer une bibliothèque de Classe, il faut faire menu Fichier, Nouveau, Projet :



Cliquer sur l'icône 'Bibliothèque de Classes'.

Le nom par défaut est ClassLibrary1 , valider sur *Ok*.

Dans la fenêtre principale, il y a :

`Public Class Class1`

`End Class`

On peut écrire le code, la description d'une classe avec ses propriétés, ses méthodes, ses constructeurs... (Voir page précédente)

On peut **ajouter une autre Classe** (Menu Projet, ajouter une Classe), ou **importer une Classe** (Menu Projet, Ajouter un élément existant)



Il n'y a **pas de procédure Sub Main**. (c'est évident, un composant n'est jamais autonome; c'est l'application cliente qui a cette procédure).

Une bibliothèque de classe ne possède pas les composants que possède une application Windows, **il n'y a pas d'interface utilisateur**, pas de MessageBox, pas de gestion du curseur; c'est l'application cliente qui s'occupe de gérer l'interface.

Il faut enfin enregistrer la bibliothèque, la compiler.

Comment utiliser ce composant?

- ¶ Si la bibliothèque de Classe a été compilée, on obtient une DLL:

-Il faut **la référencer**: Ajouter la Dll au projet (Menu Projet, Ajouter une référence)

-Importer l'espace de nom par **Imports Espace de nom** au début du module.

-On peut ensuite utiliser la Classe dans l'application cliente.

- ¶ On peut travailler en même temps sur l'application cliente et le projet de la bibliothèque de Classe en les chargeant tous les 2 dans **une solution**.

Contrôle utilisateur:

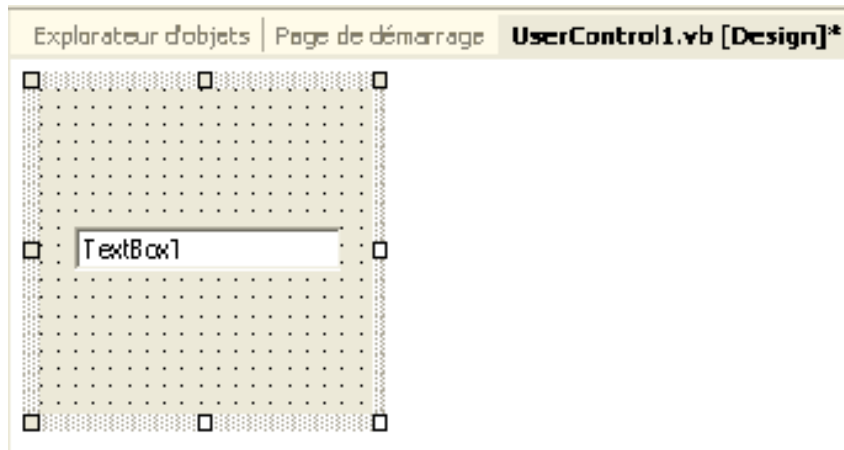
Permet de **créer des contrôles spécifiques** qui enrichiront la 'Boite à outils' :

(Pour les 'anciens' c'est comme les contrôles OCX, sauf que c'est des contrôles .NET et qu'il sont dans des fichiers .dll)

Exemple: on veut avoir une TextBox qui a un comportement différent de la TextBox habituelle.

Pour créer une bibliothèque de Contrôle, il faut faire menu Fichier, Nouveau, Projet, Icône '**Bibliothèque de contrôle Windows**' :

On obtient une fenêtre qui ressemble à un formulaire mais sans bord, on peut y ajouter un contrôle (un TextBox par exemple comme ici)



Si on regarde le code correspondant, Vb a créé une classe UserControl1 qui hérite de la classe Forms.UserControl

```
Public Class UserControl1
```

```
Inherits System.Windows.Forms.UserControl
```

```
End Class
```

Dans ce UserControl il y a les procédures **privées** des événements des composants, (ici le TextBox1 et **Private TextBox1_Changed** par exemple si on double clique sur le TextBox1); bien sûr, elles ne seront pas visibles ni accessibles par l'utilisateur du composant.

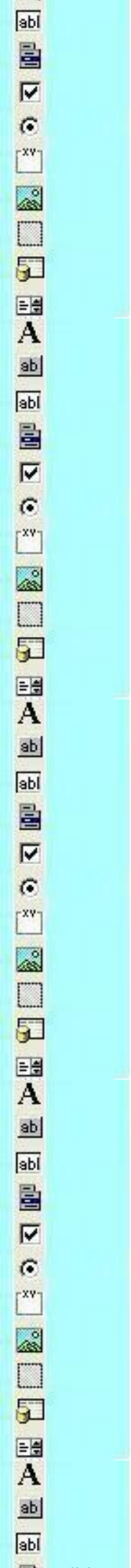
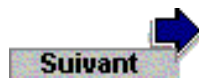
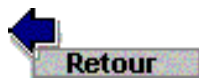
L'interface du UserControl (ce que verra l'utilisateur du composant) est créée de toute pièce comme dans un module de Class.

Si on double-clique sur le fond, on voit quand même apparaître:

```
Private Sub UserControl1_Load(..
```

```
End Sub
```

Mais on rajoutera **des membres publics** qui seront accessibles à l'utilisateur du composant. On utilise pour cela les 'Property', 'Sub' et 'variables' **publics** pour créer une interface. Le code contenu dans ces procédures de l'interface ira 'piloter' les ou les contrôles (comme le TextBox). Ce code modifiera (dans notre exemple) le comportement du TextBox initial.



Site **LDF** Cours **VB.net**



5.4

Les interfaces.

Suivant

Sommaire

Définition: Interface et implémentation:

Ce que je vois de l'objet, c'est son **interface** (le **nom** des propriétés, le **nom** des méthodes..) exemple: le nom de la méthode **Clear** fait partie de l'interface d'une ListBox. Par contre le code qui effectue la méthode (celui qui efface physiquement toutes les lignes de la listBox), ce code se nomme **implémentation**, lui n'est ni visible ni accessible (Quand on est du coté du développeur qui utilise l'objet).

Un objet a donc une interface et une implémentation.

Quand maintenant on est du coté du créateur d'objet, dans un module de classe, si on a crée un objet et ses membres, sans le savoir, on crée en même temps l'interface et l'implémentation.

Mais il est possible de dissocier les 2.

Quel intérêt d'utiliser les interfaces?

Vous pouvez développer des implémentations avancées pour vos interfaces sans endommager le code existant, ce qui limite les problèmes de compatibilité. Vous pouvez également ajouter de nouvelles fonctionnalités à tout moment en développant des interfaces et implémentations supplémentaires.

Différences entre Classe et Interface:

Tout comme les classes, les interfaces définissent un ensemble de propriétés, méthodes et événements. Cependant, contrairement aux classes, les interfaces n'assurent pas l'implémentation. Elles sont implémentées par les classes et définies en tant qu'entités distinctes des classes.

Les interfaces ne peuvent pas être modifiées après publication. En effet, toute modification apportée à une interface publiée risque d'endommager le code existant. Il faut partir du principe qu'une interface est une sorte de contrat. (La partie qui publie une interface accepte de ne jamais modifier cette dernière et l'implémenteur accepte de l'implémenter exactement comme elle a été conçue.)

Comme d'habitude:

Il y a

-les interfaces présentes dans les classes du Framework (IList, ICollection...)

-les interfaces que vous créez de toutes pièces pour créer des objets.

Visual Basic .NET vous permet de définir des interfaces à l'aide de l'instruction **Interface** et de les implémenter avec le mot clé **Implements**.

Les interfaces présentes dans les classes du Framework.

Pour 'uniformiser' le comportement des objets, les interfaces sont largement utilisées dans VB.

Prenons l'exemple **des collections**:

Plutôt que de rendre communs à toutes les collections une méthodes (Clear par exemple), Vb donne la même interface à plusieurs types de Collections, ce qui uniformise la totalité des membres.

Les collections reposent sur l'interface **ICollection**, **IList** ou **IDictionary**. Les interfaces **IList** et **IDictionary** sont toutes les deux dérivées de l'interface **ICollection**.



Le nom des interfaces commence toujours 'I'.

Dans les collections fondées sur l'interface **IList** ou directement sur l'interface **ICollection** (telles que Array, ArrayList, Queue ou Stack), chaque élément contient une valeur unique. Dans les collections reposant sur l'interface **IDictionary** (telles que Hashtable ou SortedList), chaque élément contient à la fois une clé et une valeur.

Détaillons l'interface **Ilist**:

L'interface **Ilist** permet de présenter une collection d'objets accessibles séparément par index.

Les méthodes de l'interface **IList** sont répertoriées ici.

Méthodes publiques

Add	Ajoute un élément.
Clear	Supprime tous les éléments.
Contains	Détermine si la liste contient une valeur spécifique.

IndexOf	Détermine l'index d'un élément spécifique.
Insert	Insère un élément dans la liste à la position spécifiée.
Remove	Supprime la première occurrence d'un objet spécifique.
RemoveAt	Supprime l'élément correspondant à l'index spécifié.

Propriétés publiques

IsFixedSide	Obtient une valeur indiquant si IList est de taille fixe.
IsReadOnly	Obtient une valeur indiquant si IList est en lecture seule.
Item	Obtient ou définit l'élément correspondant à l'index spécifié.

Les tableaux (Array) utilisent l'interface `Ilist`, mais aussi les collections (ArrayList), des contrôles utilisent aussi cette interface (les `ListBox`, `ComboBox`), mais aussi les `DataView`...

Les `ListBox` possèdent donc l'interface `Ilist`, on s'en doutait car on utilisait les méthodes `Clear`, `Insert`, `Item`...

Il y a plein d'autres interface.

Les interfaces créés par vous:

De même que vous savez créer des classes, il est possible de créer de toutes pièces des interfaces.

Pour créer une Interface

1. Dans un nouveau Module, définissez votre Interface en commençant par le mot clé `Interface` et le nom de l'interface et se terminant par l'instruction `End Interface`. Par exemple, le code suivant définit une Interface appelée `Cryptage` :

```
Interface Cryptage
End Interface
```

2. Ajoutez des instructions définissant les propriétés, méthodes et événements pris en charge par votre Interface. Par exemple, le code suivant définit deux méthodes, une propriété et un événement :

```
Interface Cryptage
    Function Encrypt(ByVal esttring As String) As String
    Function Decrypt(ByVal dstring As String) As String
    Property CledeCodage() As Integer
    Event FinDecoding(ByVal RetVal As Integer)
End Interface
```

L'interface est créée.

Pour implémenter une Interface

1. Si l'interface que vous implémentez ne fait pas partie de votre projet, ajoutez une référence à l'assembly qui contient l'interface.
2. **Créez une nouvelle classe** qui implémente votre Interface et ajoutez le mot clé **Implements** dans la ligne à la suite du nom de la classe. Par exemple, pour implémenter l'interface Cryptage , vous pouvez nommer la classe d'implémentation MonEncrypte, comme dans le code suivant :

```
Class MonEncrypte
    Implements Cryptage
End Class
```

3. Ajoutez des procédures **pour implémenter les propriétés, méthodes et événements** de la classe :

```
Class MonEncrypte
    Implements Cryptage
    Event FinDecoding(ByVal RetVal As Integer) _
        Implements Cryptage.FinDecoding

    Function Encrypt(ByVal estring As String) _
        As String Implements Cryptage.Encrypt
        ' Placer le code de cryptage ici.
    End Function

    Function Decrypt(ByVal dstring As String) _
        As String Implements Cryptage.Decrypt
        ' Placer le code de décryptage ici.
    End Function

    Property CledeCodage() As Integer _
        Implements Cryptage.CledeCodage
        Get
            'Placer ici le code qui retourne la valeur de la
            propriété.
        End Get
        Set
            'Placer ici le code qui donne une valeur à la
            propriété.
        End Set
    End Property
End Class
```

Noter que :

Pour chaque membre implémenté dans ce code, une instruction **Implements** indique le nom de l'interface et du membre implémenté.

Tous les membres de l'interface doivent être implémentés.

Enfin utiliser la classe MonEncrypte dans votre programme.

```
Dim C As New MonEncrypte()
```

```
C.CledeCodage=3
```

```
ChaineEncryptée= C.Encrypt( ChaineAEncrypter)
```

ou

Il faut créer une instance de la classe qui implémente MonEncrypte, crée une variable du type de l'interface, qui associe un gestionnaire d'événements à l'événement déclenché par l'instance, qui définit une propriété et exécute une méthode via l'interface.

```
Dim C As New MonEncrypte() 'Classe
```

```
Dim I As New Cryptage()'Variable d'interface
```

```
I=C
```

```
I.CledeCodage=3
```

```
ChaineEncryptée= I.Encrypt( ChaineAEncrypter)
```

Les 2 versions marchent-elles?

S'il y a un RaiseEvent dans une procédure qui déclenche un évènement de la classe il faut aussi ajouter une ligne AddHandles.

Il peut y avoir héritage de plusieurs interface:

```
Interface IComboBox
```

```
    Inherits ITextBox, IListBox
```

```
End Interface
```

```
Public Class EditBox
```

```
    Inherits Control
```

```
    Implements ITextBox
```

```
    Implements IListBox
```

```
    Public Sub Paint()Implements ITextBox.Paint
```

```
        ...
```

```
    End Sub
```

```
    Public Sub Bind(b As string) Implements IListBox.Clear
```

```
    End Sub
```

```
End Class
```



Retour



Index



Sommaire



Suivant



5.10	Approche 'fonctionnelle', approche 'Objet'.	Suivant	Sommaire
------	--	---------	----------

On a vu qu'on pouvait créer des programmes avec des Sub et des Fonctions mais aussi avec des objets. Détaillons.

Approche fonctionnelle:

Elle découpe le problème en fonctions:

Chaque fonction effectue une tâche précise. Avec l'aide de variables et de structures.

Il y a donc

Une logique de classification de données en variables , structures.

Une logique de traitement: les Fonctions et Sub contiennent le code.

Exemple:

Calcul du salaire d'un employé. (Nombre d'heure * Taux horaire)

Il faut écrire une Fonction CalculSalaire:

```
Public Function CalculSalaire(Taux As Single, Heure As Single) As Single
    Return Taux*Heure
End Function
```

Pour calculer un salaire il faut appeler la fonction avec les bons paramètres.

```
Dim TauxHoraire As Single
Dim HeuresTravaillées As Single
```

Dim Salaire As Single

TauxHoraire=30

HeureTravaillées=70

Salaire=**CalculSalaire**(TauxHoraire,HeuresTravaillées)

Pour structurer le programme, on utilise, en plus des modules de formulaire, des '**Modules Standards**' contenant les divers fonctions.

Le point d'entrée du programme est:

```
Module main1
Sub main()
....
End Sub
End Module
```

Approche Objet:

Elle nécessite de créer une Classe

Avec l'aide de la classe on peut déclarer des objets.

Chaque Objet à des propriétés, des méthodes.

Les données sont dans les propriétés des l'objets.

Le traitement est implémenté dans les méthodes de l'objet.

Exemple:

Calcul du salaire d'un employé. (Nombre d'heure * Taux horaire)

Il faut écrire dans un module de Class une Class Employé:

```
Public Class Employé
```

```
Private T As Single 'propriétés privées à la classe pour stocker les
heures et taux horaires
```

```
Private H As Single
```

```
Public Property Taux As Single 'propriété Taux
```

```
Get
```

```
Return T
```

```
End Get
```

```
Set(By Val Value)
```

```
T=value
```

```
End Set
```

```
End Property
```

```
Public Property Heure As Single 'propriété heure
```

```
Get
```

```
Return H
```

```
End Get
```



```
Set(By Val Value)
```

```
    H=value
```

```
End Set
```

```
End Property
```

```
Public Property Salaire As Single 'méthode Salaire
```

```
Get
```

```
    Return T*H
```

```
End Get
```

```
End Property
```

```
End Class
```

Pour calculer un salaire il faut créer un objet employé, donner les bonnes valeurs aux propriétés et appeler la méthode salaire.

```
Dim UnEmployé As new Employé
```

```
UnEmployé.Taux=30
```

```
UnEmployé.Heure=70
```

```
    Dim Salaire As Single =UnEmployé.Salaire
```

On voit donc qu'il faut créer des modules de Classe pour y mettre les nouveaux objets.

On évitera les modules standards qui ne sont pas dans l'esprit 'Objet'. Les routines 'public', les fonctions des modules deviennent des méthodes 'statiques' d'objet.

Le point d'entrée du programme pourrait être une Classe statique:

```
Public Class main2
```

```
Public Shared Sub main()
```

```
    ...
```

```
End Sub
```

```
End Class
```

Que choisir?

La méthode fonctionnelle est plus intuitive, on a l'impression d'être plus proche de la réalité 'physique', le code est probablement plus rapide.

L'emploi d'objet permet une abstraction plus importante, une puissance inégalée.

Conséquence pratique:

On peut par exemple créer une classe qui hérite des propriétés d'une autre classe:

Dans notre exemple en programmation Objet, on créera une Class 'Patron'

qui héritera de la classe 'Employé', mais dont la méthode Salaire sera différente, en programmation fonctionnelle il faudra écrire une nouvelle fonction SalairePatron.



Site LDF Cours VB.net



6.1	Les bases de données.	Suivant	Sommaire
-----	-----------------------	---------	----------

Comment lire et écrire des informations complexes et structurées?

Généralités:

Pour travailler avec du texte, des octets, des données très simple (sans nécessité d'index, de classement..), on utilise les fichiers séquentiels, aléatoires, binaires (chapitre 4-6).

Mais dès que les informations sont plus structurées, il faut utiliser les **bases de données (Data Base en anglais)**.

Une base de données peut être:

- locale: utilisable sur un ordinateur par un utilisateur.
- répartie, c'est-à-dire que la base est stockée sur des machines distantes et accessibles par réseau.

Plusieurs utilisateurs peuvent accéder à la base simultanément.

Exemple de type de base de données:

Site
Dbase Format très utilisé, qui date maintenant un peu, les fichiers contenant ses bases ont l'extension .dbf

Paradox

FileMaker

FoxPro

Interbase

Access Format très répandu, les fichiers contenant ses bases ont l'extension .mdb

SQLServeur les fichiers contenant ses bases ont l'extension .dbo

SyBase

MySql

Oracle..

Pour pouvoir contrôler les données, l'accès à ces données et les utilisateurs utilisant une base de données, un **système de gestion** est nécessaire. La gestion de la base de données se fait grâce à un système appelé **SGBD (système de gestion de bases de données)**, si la base de données est relationnelle (Existence de relation entre les tables) on parle de **SGBDR (système de gestion de bases de données relationnelles)**



Un SGBD est un logiciel qui joue le rôle d'interface entre les utilisateurs et la base de données.

Un SGBD permet de décrire, manipuler et interroger les données d'une 'Base de Données'.

Tables:

Dans une base de données, il y a des **tables**:

Une table sert à stocker physiquement des données sous forme d'un tableau comportant **des lignes** (rows) et **des colonnes** (columns).

Exemple:

Une base de données Access nommée Cabinet.mdb contient **les patients d'un cabinet, leurs consultations, les ordonnances, les médicaments..**

Dans cette base il y a plusieurs tables: une **table patient**, une **table consultation**..

Examinons la table patient:

Sur **chaque ligne**, il y a un patient,.

Chaque colonne représente un type de données (première colonne= civilité, seconde=nom, troisième=prénom, quatrième= numéro interne propre à chaque patient.)

L'ancienne terminologie parlait d'**enregistrements** (lignes) et de **champs** (colonnes)

Champs Civilité Champ Nom Champ Prénom Champ Numéro
Interne

M.	Durand	Luc	1
Mme	Dupont	Josette	2
M.	Thomas	Guy	3

Ici la seconde ligne (le 2eme enregistrement) contient la civilité, le nom, le prénom, le numéro du patient Dupont Josette.

Chaque colonne à un type bien définie: dans notre cas la première colonne contient du texte, ainsi que la seconde, la troisième; la quatrième colonne contient un numérique long par exemple.

Examinons la table consultation:

Sur **chaque ligne**, il y a une consultation,.

Chaque colonne représente un type de données (première colonne= numéro correspondant au patient, seconde=date, troisième=texte de la consultation, quatrième= Courrier.)

Champs Numéro Champ Date Champ Texte Champ Courrier
Interne

1	02/12/2003	Angine	
2	02/02/2004	Hta	
1	05/04/2004	Bronchite	

Il n'est pas question pour chaque consultation d'enregistrer de nouveau le nom et le prénom du patient, cela enregistrerait 2 fois la même information puisque le nom et le prénom du patient sont déjà dans la table 'patient'. On va donc, pour éviter les redondances, utiliser un numéro interne: chaque patient a un numéro unique (4ème champ de la table 'nom'); il suffit de noter dans chaque consultation le numéro du patient.

Ensuite, si je consulte le patient Durand Luc, sachant que son numéro interne est '1', il suffit de rechercher dans la table consultation les consultations dont le premier champ est 1: Durand Luc à 2 consultations.

Type de colonne:

Il existe des types de colonne (de champs) **alphanumériques**

- de longueur fixe (pour le champ 'nom' je prévois 30 caractères par exemple).
- de longueur variable (champ mémo dans la base Dbase par exemple)

Il existe aussi

- des champs **numériques**,
- des champs **dates**
- et dans certains base de données des champs **booléens, image...**

Clé primaire

Quand il est nécessaire de **différencier chaque enregistrement de manière unique**, il faut définir un champ comme **clé primaire**.

Ce champ doit être unique pour chaque enregistrement (il ne doit pas y avoir de doublons: 2 enregistrements ne peuvent pas avoir la même clé primaire), et la valeur de la clé primaire ne peut pas être égale à null.

Dans notre exemple de la table patient, on ne peut pas utiliser le champ 'nom' comme clé primaire car plusieurs patients peuvent avoir le même nom, il est judicieux de choisir le champ 'numéro interne' comme clé primaire car chaque patient (donc chaque enregistrement) à un numéro interne unique.

Quand on enregistre une nouvelle fiche patient, il faut donc donner un nouveau 'numéro interne' qui n'a jamais été utilisé, en pratique:

- On gère soi même les numéros: on prend le dernier numéro interne (on cherche la dernière fiche dont on récupère le numéro ou bien on lit ce numéro qui a été enregistré quelque part), on ajoute 1 pour avoir le nouveau numéro.
- On utilise un champ qui s'incrément automatiquement a chaque fois que l'on crée une enregistrement (NumeroAuto dans Access)

Index

Un index permet d'**optimiser les recherches** dans une table, de **les rendre beaucoup plus rapide**.

Expliquons:

Si j'ai une table contenant les noms des médecins utilisateurs et que je veux chercher un nom, comme il y a au maximum 5 à 6 médecins dans un cabinet, pour rechercher un nom, il suffit de lire successivement chaque enregistrement et de voir si c'est celui recherché. C'est suffisamment rapide.

Par contre si je recherche dans la table patient un patient, comme il y a 4000 à 8000 enregistrements, je ne peux pas les lire un à un, c'est trop long, aussi je crée un index: c'est comme l'index d'un livre, le nom me donne directement l'endroit où se trouve l'enregistrement correspondant.

On peut combiner plusieurs champs pour en faire la base d'un index.

Pour ma table 'patient', je peux créer un index nommé IndexPatient qui sera indexé sur Nom +Prenom.

Il peut y avoir plusieurs index sur une même table.



Les index accélèrent les recherches mais s'il y en a

trop, cela alourdit le fonctionnement; on ne peut pas tout indexer!!

Relations entre les tables: différents types de relations.

On a déjà vu que 2 tables peuvent être liées et avoir **un champ commun présent dans les 2 tables.**

Sur ce champ commun, il peut exister plusieurs types de relation:

Relation 1 à N

Relation 1 à 1

Relation N à M

Voyons cela en détail:

1 à N (relation un à plusieurs)

Dans notre exemple la table 'patient' et la table 'consultation' ont chacune un champ numéro interne. Ce qui permet de lier à l'enregistrement du patient de numéro interne X toutes les consultations pour ce patient (elles ont dans leurs champs 'numéro interne' la valeur X.

Comme pour UN patient il peut y avoir N consultations, on parle de relation 1 à N.

Un enregistrement unique est lié à plusieurs enregistrements de l'autre table par un champ présent dans les 2 tables.

On remarque que le champ du côté patient est une clé primaire.

Table 'patients'

**Champ Numéro
Interne**

M.	Durand	Luc	1
Mme	Dupont	Josette	2
M.	Thomas	Guy	3

Table 'consultations'

Champ Numéro
Interne

1	02/12/2003	Angine	
2	02/02/2004	Hta	
1	05/04/2004	Bronchite	

Le patient Durand Luc a 2 consultations : le 02/12/2003 et le 05/04/2004 (Le numéro interne de ce patient est 1, mais l'utilisateur final n'a pas à le savoir ni à le gérer: la relation utilisant le numéro interne est **transparente** pour l'utilisateur final)

Il existe aussi les relations:

1 à 1

Un enregistrement unique est lié à un autre enregistrement unique par un champ présent dans les 2 tables.

On peut imaginer dans notre exemple, créer une table Antécédents contenant aussi un champ numéro interne; pour chaque enregistrement de la table patient, il y a un enregistrement unique dans la table Antécédents, de même numéro interne contenant les antécédents du patient.

Table 'patient'

M.	Durand	Luc	1
Mme	Dupont	Josette	2
M.	Thomas	Guy	3

Table 'antécédents'

Champ Numéro interne.

1	02/01/2003	appendicite
2	02/02/2004	Hta
3	05/05/2004	Cancer du colon

Enfin existe les relations:

N à M

Relation plusieurs à plusieurs. Plusieurs enregistrements de la première table peuvent être liés à plusieurs de la seconde table et vice versa.

Exemple:

J'ai une table 'ordonnances' qui peut contenir plusieurs médicaments, et une table 'médicaments' dont les médicaments peuvent être utilisés dans plusieurs ordonnances différentes.

Il faut dans ce cas avoir la table 'ordonnances' avec une clé primaire sur un numéro d'ordonnance (numéro d'ordonnance unique s'incrémentant à chaque nouvelle ordonnance), une table 'médicaments' avec une clé primaire sur le numéro unique du médicament, et **créer une troisième table gérant la relation ordonnance-médicament.**

Table 'Ordonnances'

'Numéro ordonnance' 'Numéro Interne patient' 'Champ' date'

1	2	02/05/2002	
2	3	02/04/2003	
3	2	06/05/2004	

Table 'Médicaments'

'Numéro médicament' 'Libelle médicament' 'Code CIP'

1	Amoxicilline	65897	
2	Omeprazone	66589	
3	Allopurinol	78456	

Table supplémentaire 'Contenu ordonnance'

'Numéro ordonnance' 'Numéro médicament'

1	1
1	2
2	2

Ici le patient de numéro interne 2 (Dupont Josette) a une ordonnance visible dans la table 'Ordonnances' (numéro interne: 2; numéro de l'ordonnance: 1); si on cherche dans la table 'Contenu ordonnance' (Index créé sur le numéro d'ordonnance) on retrouve 2 enregistrements (ayant un numéro d'ordonnance 1), on constate que l'ordonnance contient les médicaments 1 et 2 qui correspondent (table

'médicaments') à de l'amoxicilline et de l'oméprazole.

On remarque qu'une ordonnance peut avoir autant de médicaments que l'on veut.

Relation N à M avec N fixe et petit

Dernier cas non décrit dans les livres:

J'explique: si chaque ordonnance à au maximum 3 médicaments (que la sécu serait contente si c'était vrai!!), il est possible de créer une table 'ordonnances' contenant 3 champs médicaments. Dans ce cas on se passe de la 3eme table.

Table 'Ordonnances'

'Numéro ordonnance' 'Numéro Interne patient' 'Champ date' 3 champs médicaments

1	2	02/05/2002	1	2	
2	3	02/04/2003			
3	2	06/05/2004			

Table 'Médicaments'

'Numéro médicament' 'Libelle médicament' 'Code CIP'

1	Amoxicilline	65897	
2	Omeprazole	66589	
3	Allopurinol	78456	

Contraintes

Un champ peut avoir certaines contraintes:

- On peut **interdire la valeur Null**: Cela empêche d'enregistrer un champ vide. On peut aussi donner une valeur par défaut.
- On peut **empêcher les doublons**.
- On peut exiger l'**intégrité référentielle**: La valeur d'un champ doit exister dans le champ d'une autre

table.(On ne peut pas enregistrer une consultation pour le patient de numéro interne 2000 s'il n'existe pas de fiche patient ayant le numéro 2000)

- ¶ On peut exiger des règles de validation pour un champ: interdire les valeurs négatives par exemple.

Serveur de fichier, Client serveur.

Si plusieurs utilisateurs sont connectés à une base Access à travers un réseau, chaque utilisateur a sur son poste un 'moteur' Access, qui récupère l'ensemble des données à utiliser et qui les traite en local. On parle de **serveur de fichier**.

Le moteur d'accès est présent sur chaque poste.

Si plusieurs utilisateurs sont connectés à une base SQLServer: la base est sur le serveur avec le logiciel SQLServer.

Un logiciel utilisateur(le client) envoie au serveur une requête.

Le logiciel SQLServer traite la requête sur le serveur et retourne au logiciel client uniquement le résultat de la requête.

On parle d'**architecture Client-serveur**.

Le moteur d'accès est présent uniquement sur le serveur.

Si on cherche un enregistrement parmi 60 000 enregistrements, en serveur de fichiers, les 60 000 enregistrements sont envoyées par le réseau vers le moteur Access de l'utilisateur ; le moteur les traite pour en sortir un.

En client serveur, le logiciel utilisateur envoie une requête au serveur, le logiciel serveur cherche sur le serveur dans la base l'enregistrement, il le trouve et envoie à travers le réseau vers le logiciel client uniquement un enregistrement.

Opérations sur les enregistrements

De manière générale, on peut:

Ouvrir une base de données (Open)

Ajouter un enregistrement (Add)

Effacer un enregistrement (Delete)

Modifier un enregistrement (Update)

Chercher un ou des enregistrements.

Fermer la base. (Close)

Avant:

Il y a bien longtemps, on choisissait un index, on cherchait un enregistrement (avec Seek), on le lisait, le modifiait, on avançait (MoveNext) ou on reculait (MovePrevious) dans la base, mais c'est de l'histoire ancienne!!



Avec **ADO.NET**:

Maintenant quelle que soit la base de données, on utilise un langage unique: **le 'SQL' pour faire une requête** sur une base de donnée (extraction de certains enregistrements ou de certains champs **en fonction de critères**), le résultat (un ensemble d'enregistrements ou de champs) se retrouve dans un **DataSet**.



Site **LDF** Cours **VB.net**



6.2	ADO.NET	Suivant 	 Sommaire
------------	----------------	--	---

Comment travailler sur les Base de données en VB.NET? Avec ADO.NET

Généralités:

Pour avoir accès à partir de VB.NET aux bases de données il faut utiliser ADO.NET.

ADO veut dire **ActiveX Database Objet** .

C'est la couche d'accès aux bases de données, le SGBD (Système de Gestion de Base de Données) de VB.

ADO.NET est ".NET" donc **managé** et géré par le CLR.

Il est indépendant de la base de donnée: alors que initialement chaque type de gestionnaire de base de données avait ses instructions, sa manière de fonctionner, ADO.NET à un langage unique pour ouvrir, interroger, modifier une base de données quelle que soit la base de données.

Le langage de requête est le SQL.

Les Managed Providers

Pour avoir accès aux données il faut charger les **DRIVERS**

(ou providers).

Comme d'habitude, il faut:

- Charger les références des drivers (les Dll)
- Importer les espaces de nom.

Ainsi on a accès aux objets.

Voyons cela:

- **OLE DB** Managed Provider est fourni; après avoir importé le NameSpace System.Data.OLEDB, on peut travailler sur des base Access par exemple.
- **SQL Server** Managed Provider est fourni; après avoir importé le NameSpace System.Data.SqlClient, on peut travailler sur des base SqlServeur.
- Un composant **ODBC** et un composant **ORACLE** sont disponible sur le site MSDN.
- Pour travailler sur une base **MySQL** lisez le très bon didacticiel [MySQLDotNet](#) (sur developpez.com bien sur);il utilise le Managed Provider [ByteFX](#)

Les Objets ADO.NET

Il faut disposer d'un objet **Connexion** pour avoir accès à la base de données, on met dans la propriété **ConnectionString** les paramètres de la base de données.

Avec la méthodes **Open** et **Close**, on ouvre et on ferme la base.

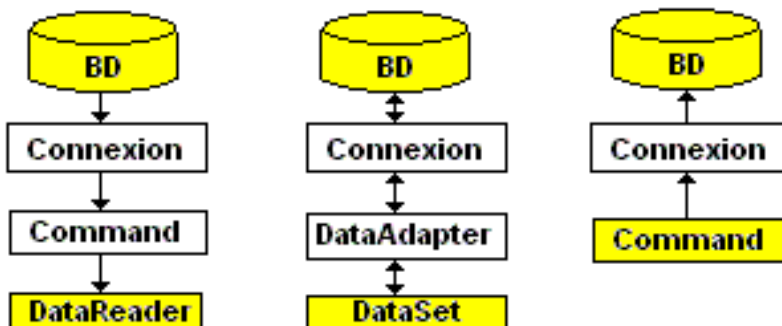
on peut ensuite travailler de 3 manières:

- Avec un objet **DataReader** on extrait les données en **lecture seule**: c'est rapide; on peut lire uniquement les données et aller à l'enregistrement suivant. Il travaille en mode connecté. Pour gérer un DataReader on a besoin d'un objet **Command**.
- Avec un objet **DataSet** on manipule les données: une requête SQL charge le DataSet avec des enregistrements ou des champs, on travaille sur les lignes et colonnes du DataSet en local, en mode déconnecté(une fois que le DataSet est chargé, la connexion à la base de données est libérée). Pour

alimenter un DataSet on a besoin d'un objet DataAdapter qui fait l'intermédiaire entre la BD et le DataSet.

- Avec un objet **Command** on peut manipuler directement la BD (Update, Insert, Delete)

Résumons les différents objets nécessaires pour travailler sur une BD:



Noter bien le sens des flèches: le DataReader est en lecture seule, le DataSet peut lire et écrire, l'objet Command peut modifier la BD.

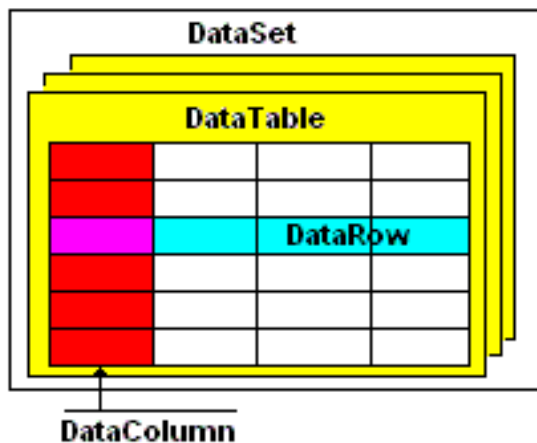
Ce schéma souligne aussi les objets intermédiaire nécessaire: un objet connexion dans tous les cas, un objet Command pour le DataReader, un objet DataAdapter pour le DataSet.

Enfin certains contrôles comme le DataGridView par exemple peuvent afficher des données à partir d'un DataSet.

On peut aussi créer une DataTable à partir d'un DataSet et alimenter une listBox ou une comboBox.

DataSet

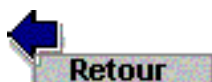
Une requête SQL charge le DataSet **en local** avec des enregistrements ou des champs, on travaille sur les lignes et colonnes du DataSet en local, en mode déconnecté(une fois que le DataSet est chargé, la connexion à la base de données est libérée).



Le DataSet a la structure d'une base de données :

Exemple :

Avec ADO.NET je lance une requête SQL demandant toutes les fiches de la table 'nom' dont le champ 'prénom' est 'Philippe', je récupère un DataSet local contenant tous les fiches (Le DataColumn "Prénom" ne contient que des 'Philippe'). Je peux modifier en local le DataSet, (modifier une date de naissance par exemple) et mettre à jour automatiquement la base de données distante.



Site LDF Cours VB.net



6.3	SQL.		
		Suivant	Sommaire

Comment adresser une requête vers une Base de données de ADO.NET? Avec SQL

Généralités:

SQL veut dire **Structured Query Language** : Langage d'interrogation structurée

SQL grâce au couplage avec un SGBD relationnelle permet un **traitement interactif des requêtes.**

SQL est le langage unique qui permet de décrire, manipuler, contrôler l'accès et interroger les bases de données relationnelles.

C'est un langage déclaratif, qui est régi par une norme (ANSI/ISO) qui assure la portabilité du langage sur différentes plates-formes aussi bien matérielles que logicielles. Une commande SQL écrite dans un environnement Windows sous ACCESS peut, souvent sans modification, être utilisée directement dans un environnement ORACLE sous Unix...

SQL est utilisé dans ADO.NET pour manipuler **toutes** les bases de données.

Les commandes SQL

Catégorie	Commandes SQL
-----------	---------------

<i>Manipulation des tables</i>	CREATE	Création de tables
	ALTER	Modification de tables
	DROP	Suppression de tables
<i>Manipulation des données</i>	INSERT	Insertion de lignes dans une table
	UPDATE	Mise à jour de lignes dans une table
	DELETE	Suppression de lignes dans une table
<i>Contrôle des données</i>	GRANT	Attribution de droits d'accès
	REVOKE	Suppression de droits d'accès
	COMMIT	Prise en compte des mises à jour
	ROLLBACK	Suppression des mises à jour
<i>Interrogation des données</i>	SELECT	Interrogations diverses

SELECT

Permet d'extraire ,de sélectionner des données.

Ces données extraites à partir d'une base de données grâce à l'instruction SQL vont se retrouver dans un DataSet.

Syntaxe simplifiée:

SELECT champ FROM table WHERE condition

Dans la table 'table' sélectionner les enregistrements vérifiant la condition 'condition' et en afficher les champs 'champs'

Exemple

SELECT	Précise les colonnes qui vont apparaître dans la réponse
---------------	--

`SELECT Nom FROM Patient WHERE Prenom='Luc'`

WHERE ajoute un critère de sélection.

Cela signifie: dans la table Patient extraire le champ Nom de tous les enregistrements dont le prénom est "Luc" .

Dans l'exemple on obtient :

Durand

`SELECT Nom, Prenom FROM Patient WHERE Sexe='F'`

Cela signifie: dans la table Patient extraire le champ Nom et prénom de tous les enregistrements dont le champ sexe est "F"(F comme féminin) .

Dans l'exemple on obtient :

Dupont	Josette
--------	---------

`SELECT * FROM Patient WHERE Datenais>=#01/01/1950#`

Cela signifie: dans la table Patient extraire tous les champs de tous les enregistrements dont le champ date de naissance est supérieur ou égal à 01/01/1950 .

Dans l'exemple on obtient

M.	Durand	Luc	1	12/02/1952	M
M.	Thomas	Guy	3	08/02/1980	M

On remarque que

*** signifie : extraire tous les champs.**

Pour utiliser les dates , il faut les entourer de "#".

Les dates sont au format mm/jj/aaaa

En ADO.NET:

Notez bien que le résultat de la requête, les 2 enregistrements ci-dessus, se retrouvent dans un Dataset qui comporte des lignes et des colonnes.

`SELECT * FROM Patient WHERE Datenais>= #01/01/1950#
AND Datenais<= #01/01/1980#`

Cela signifie: dans la table Patient extraire tous les

champs de tous les enregistrements dont le champ date de naissance est supérieur ou égal à 01/01/1950 et inférieur ou égal à 01/01/1980 .

On remarque que on peut utiliser avec Where, les opérandes

AND OR NOT.

Il est bien sur possible de combiner des conditions sur des champs différents:

Sexe='M' AND Prenom='Luc'

```
SELECT * FROM Patient WHERE BETWEEN #01/01/1950# AND #01/01/1980#
```

Même signification que le précédent mais en utilisant BETWEEN AND qui est plus performant.

```
SELECT Nom FROM Patient WHERE Prenom IN ('Luc' , 'Pierre', 'Paul')
```

Cela signifie qu'il faut extraire les enregistrements dont le prénom est Luc, Pierre ou Paul .

```
SELECT Nom FROM Patient WHERE Prenom LIKE 'D%'
```

Cela signifie qu'il faut extraire les enregistrements dont le prénom commence par un 'D'.

LIKE recherche des chaînes de caractères avec l'aide de caractères génériques:

% représente une chaîne de caractères même vide.

_ représente un caractère.

On peut spécifier une série de caractères en les mettant entre ""

Exemple :

LIKE 'D%' commence par D

LIKE '%D%' contient D

LIKE '[DF]%' commence par D ou F

LIKE '___' contient 3 caractères

`SELECT Nom FROM Patient WHERE SEXE IS NULL`

Cela signifie qu'il faut extraire les enregistrements dont le sexe n'a pas été enregistré.

`SELECT DISTINCT Nom FROM Patient WHERE SEXE IS NULL`

DISTINCT permet d'éviter les doublons

Si dans les Noms extraits il y a 2 fois le même (2 membres d'une même famille) , il n'en est gardé qu'un.

TRI des enregistrements:

ORDER BY sert à trier les enregistrements.

Il est placé à la fin.

DESC sert à trier par ordre décroissant.

`SELECT Nom, Prenom , Sexe, DateNais FROM Patient WHERE Sexe='F' ORDER BY DateNais`

Trie les enregistrements de sexe 'F' par date de naissance

`SELECT Nom, Prenom, DatNais, NumInt FROM Patient WHERE Sexe='F' ORDER BY DateNais DESC, NumInt`

Trie les enregistrements de sexe 'F' par date de naissance mais décroissante et pour une même date de naissance par numéro interne croissant.

Statistiques:

`SELECT COUNT(*) AS NombrePatient FROM Patient`

Compte le nombre total d'enregistrement dans la table Patient et met le résultat dans le champ NombrePatient

On peut aussi utiliser:

MIN retourner la plus petite valeur.

MAX retourner la plus grande valeur.

SUM retourner la somme.

AVG retourner la moyenne.

VAR retourner la variance

STDEV retourner l'écart type.

```
SELECT Prenom ,COUNT(*) AS NombrePrenom FROM Patient  
GROUP BY Prenom
```

Extrait la liste des prénom avec le nombre de fois que le prénom est utilisé.

GROUP BY regroupe les enregistrements par valeur.

```
SELECT Prenom ,COUNT(*) AS NombrePrenom FROM Patient  
GROUP BY Prenom HAVING CONT(*)>3
```

Extrait la liste des prénoms avec le nombre de fois que le prénom est utilisé S'il est utilisé plus de 3 fois..

HAVING rajoute un critère au regroupement.

Extraction de données sur plusieurs tables:

Parfois on a besoin d'extraire des champs de plusieurs tables différentes, mais ayant une relation (un champ commun); pour cela on utilise **une jointure**.

Pour chaque enregistrement de la première table, on affiche en regard les enregistrements de la 2eme table qui ont la même valeur de jointure.

Exemple:

Soit la table patient:

Champs Champ Champ Numéro Datenais Sexe
'Civilité' 'Nom' 'Prénom' 'Ville'

M.	Durand	Luc	1	12/02/1952	M
Mme	Dupont	Josette	2	06/04/1936	F
M.	Thomas	Guy	3	08/02/1980	M

Soit la table Ville:

Nom ville Numéro ville

Paris	1
Lyon	2
Marseille	3

Comment récupérer nom et ville (en clair, pas son numéro)

```
SELECT Patient.Nom, Ville.NomVille From Patient INNER JOIN Ville  
ON Patient.NuméroVille= Ville.NuméroVille
```

On obtient:

Durand	Paris
Dupont	Lyon
Thomas	Paris



Site LDF Cours VB.net



6.4	<h2>Travailler avec un DataReader.</h2>	
-----	---	--

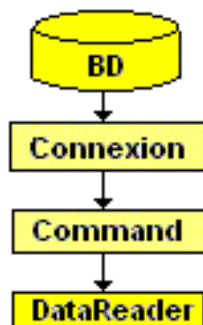
Comment lire rapidement des enregistrements? Avec un DataReader.

Généralités:

Un objet **DataReader** fournit des données en lecture seule en un temps record. La seule possibilité est de se déplacer en avant.

En contrepartie de sa rapidité il monopolise la connexion.

Il faut créer un objet **Connexion** puis un objet **Command**, ensuite on exécute la propriété **ExecuteReader** pour créer l'objet **DataReader**; enfin on parcourt les enregistrements avec la méthode **Read**.



Exemple avec une base access.

Il existe une base Access nommée 'consultation.mdb', elle contient une table 'QUESTIONS', dans cette table existe un champ 'NOM', je veux récupérer tous les noms et les afficher rapidement dans une ListBox.

Il faut **importer les NameSpaces** nécessaires.

Imports System

Imports System.Data

Imports System.Data.OleDb

Imports Microsoft.VisualBasic

Il faut **créer un objet connexion:**

```
Dim MyConnexion As OleDbConnection = New  
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data  
source=" & _  
"C:\consultation.mdb")
```

Il faut donner les paramètres Provider= et Data source=
Dans le cas d'une base Access le provider (le moteur à
utiliser est le moteur OLEDB Jet 4.

Il faut **créer un objet Command:**

```
Dim Mycommand As OleDbCommand =  
MyConnexion.CreateCommand()
```

Il faut **donner dans la propriété CommandText la requête
SQL** permettant d'extraire ce que l'on désire.

```
Mycommand.CommandText = "SELECT NOM FROM QUESTIONS"  
Ici dans la table QUESTIONS, on extrait le champ NOM
```

Il faut **ouvrir la connexion:**

```
MyConnexion.Open()
```

Il faut **créer un objet DataReader:**

```
Dim myReader As OleDbDataReader =  
Mycommand.ExecuteReader()
```

On crée une boucle permettant de **lire les enregistrements
les uns après les autres**, on récupère le champ (0) qui est
une String, on la met dans la ListBox

```
Do While myReader.Read()  
ListBox1.Items.Add(myReader.GetString(0))  
Loop
```

Remarquons que le champ récupéré est récupéré typé (ici
une string grâce à GetString); il y a d'autres types:
**GetDateTime, GetDouble, GetGuid, GetInt32, GetBoolean,
GetChar, GetFloat, GetByte, GetDecimal** etc..; on aurait pu
récupéré sans typage: on aurait écrit myReader(0).

Read avance la lecture des données à l'enregistrement
suivant , il retourne True s'il y a encore des
enregistrements et False quand il est en fin de fichier;
cela est utilisé pour sortir de la boucle Do Loop.

On ferme pour ne pas monopoliser.

```
myReader.Close()  
MyConnexion.Close()
```

Simple, non!! (Je rigole!!)

Cela donne:

```
Imports System  
Imports System.Data  
Imports System.Data.OleDb  
Imports Microsoft.VisualBasic  
Public Class Form1  
Inherits System.Windows.Forms.Form  
Private Sub Form1_Load(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles MyBase.Load  
Dim MyConnexion As OleDbConnection = New  
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data  
source=" & _  
"C:\consultation.mdb")  
Dim Mycommand As OleDbCommand =  
MyConnexion.CreateCommand()  
Mycommand.CommandText = "SELECT NOM FROM QUESTIONS"  
MyConnexion.Open()  
Dim myReader As OleDbDataReader =  
Mycommand.ExecuteReader()  
Do While myReader.Read()  
ListBox1.Items.Add(myReader.GetString(0))  
Loop  
myReader.Close()  
MyConnexion.Close()  
End Sub  
End Class
```

Dans le cas d'une **base SQLSERVEUR**, on aurait les changements suivant:

```
Imports System.Data.SqlClient  
  
Dim MyConnexion As SqlConnection = New SqlConnection("Data  
Source=localhost;" & _  
"Integrated Security=SSPI;Initial Catalog=northwind")  
Dim Mycommand As SqlCommand = MyConnexion.CreateCommand()  
Dim myReader As SqlDataReader = Mycommand.ExecuteReader()
```

Voyons en détail les objets utilisés:

L'objet Connection:

Il permet de définir une connexion.

Il faut donner les paramètres 'Provider=' indiquant le moteur de BD et 'Data source=' indiquant le chemin et le nom de la BD. Il peut être nécessaire de donner aussi 'Password=' le mot de passe, 'User ID=' Admin pour l'administrateur par exemple.

Il faut mettre ces paramètres avec le constructeur, ou dans la propriété **ConnectionString**.

Dans le cas d'une base Access le provider (le moteur à utiliser) est le moteur OLEDB Jet 4.

Il y a d'autres propriétés:

State état de la connexion (Open, Close, Connecting, Executing, Fetching, Broken)

Open, Close

ConnectionTimeout,

BeginTransaction,

..

L'objet Command:

Pour chaque provider il y a un objet Command spécifique:

SqlCommand, OleDbCommand, mais tous les objets 'Command' ont la même interface, les mêmes membres.

Command permet de définir la commande à exécuter: SELECT, UPDATE, INSERT, DELETE. en utilisant le membre **CommandText** (Seul SELECT retourne des données)

CommandType indique comment doit être traité la commande CommandText:

- Text ; par défaut (exécution direct des instructions SQL contenues dans CommandText)
- StoredProcedure: exécution de procédure stockée dont le nom est dans CommandText.
- Tabledirect.

CommandTimeout indique la durée en secondes avant qu'une requête qui plante soit abandonnée.

ExecuteReader exécute le code SQL de CommandText et retourne un DataReader.

ExecuteScalar fait de même mais retourne les champs de la

premier colonne pour une fonction Count ou Sum.

ExecuteNonQuery exécute le code SQL de CommandText (généralement une mise à jour par UPDATE, INSERT, DELETE) sans retourner de données.

L'objet DataReader

Pour chaque provider il y a un objet 'DataReader' spécifique:

SqlDataReader, OleDbDataReader

On a vu le membre **Read** qui avance la lecture des données à l'enregistrement suivant , il retourne True s'il y a encore des enregistrements et False quand il est en fin de fichier.

On a vu **GetDateTime, GetDouble, GetGuid, GetInt32, GetBoolean, GetChar, GetFloat, GetByte, GetDecimal** permettant de récupérer les valeurs typées des champs.

Il a bien d'autres propriétés:

GetName retourne le nom du champ (numéro du champ en paramètre)

GetOrdinal fait l'inverse: retourne le numéro du champ (nom en paramètre)

FieldCount retourne le nombre de colonne.

GetDataTypeName retourne le nom du type de donnés.

IsDBNull retourne True si le champ est vide

.....

Exceptions:

Chaque SGDB a des erreurs spécifiques. Pour les détecter il faut utiliser les types d'erreur spécifique:

SQLException et **OleDbException** par exemple:

Exemple d'interception d'erreur:

```
Try
    MyConnection.Open()
Catch ex As OleDbException
    MsgBox(ex.Message)
End Try
```



Retour



Index



Sommaire

Suivant 



6.5	Travailler avec un DataSet.	S ▶	S ◀
		S uivant	S ommaire

Comment travailler (lire, écrire, modifier, trier..) sur des enregistrements?

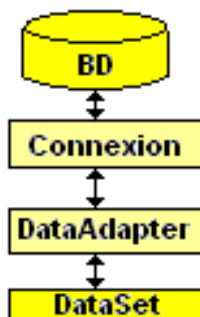
Avec un DataSet

Généralités:

Pour obtenir un ensemble de données modifiables, qui permet le tri et la recherche de données, il faut utiliser un **DataSet**.

Le DataSet est une représentation en mémoire des données; Une fois chargé on peut travailler en mode déconnecté.

Pour le remplir il faut un **DataAdapter**.

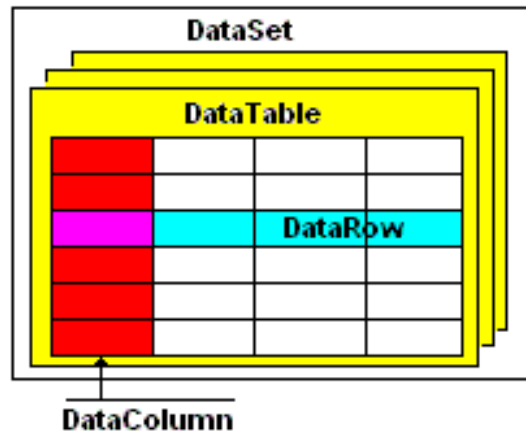


Il faut créer un objet **Connexion** puis un objet **Adapter** qui par sa propriété **Fill** charge le **DataSet**.

On pourra charger dans le DataSet des données extraites à l'aide de requête SQL.

Le DataSet est organisé comme une base de données, il possède:

- Une propriété **Tables** qui contient des **DataTable** (Comme les tables d'un BD)
- Chaque **DataTable** contient une propriété Columns qui contient les **DataColumn** (les colonnes ou champs des BD) et une propriété Rows composé de **DataRow** (Les lignes ou enregistrements des BD)



DataColumn contient des informations sur le type du champ.

DataRow permet d'accéder aux données.

Un DataSet possède aussi la propriété Constraints qui contient les **Constraint** (Clé primaire ou clé étrangère), et la propriété Relations qui contient les **DataRelations** (Relation entre les tables)

En pratique:

Soit une **base de données Access** nommée 'Nom.mdb', elle contient une table 'FichePatient' qui contient les champs (ou colonnes) 'Nom', 'Prenom'.

Je vais me connecter à cette base, extraire les enregistrements de la table 'FichePatient' et les mettre dans un DataSet; Chaque ligne du DataSet contiendra un patient. Je travaillerais sur ces lignes.

En tête du module, il faut importer l'espace de nom correspondant à OleDb.

```
Imports System.Data.OleDb
```

Dans la zone Général, Déclarations du module, il faut déclarer les objets ADO:

```
' Déclaration Objet Connexion  
Private ObjetConnexion As OleDbConnection
```

```

' Déclaration Objet Commande
Private ObjetCommand As OleDbCommand
' Déclaration Objet DataAdapter
Private ObjetDataAdapter As OleDbDataAdapter
' Déclaration Objet DataSet
Private ObjetDataSet As New DataSet()
'String contenant la 'Requête SQL'
Private strSql As String
' Déclaration Objet DataTable
Private ObjetDataTable As DataTable
' Déclaration Objet DataRow (ligne)
Private ObjetDataRow As DataRow
'Numéro de la ligne en cours
Private RowNumber As Integer
'Paramètres de connexion à la DB
Private strConn As String
'Pour recompiler les données modifiées avant de les remettre
dans le
"DataAdapter"
Private ObjetCommandBuilder As OleDbCommandBuilder

```

'Ouverture

```

'Initialisation de la chaîne de paramètres pour la connexion
strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
"Data Source= c:\nom.mdb;"
'Initialisation de la chaîne contenant l'instruction SQL
strSql = "SELECT FICHEPATIENT.* FROM FICHEPATIENT"
'Instanciation d'un Objet Connexion
ObjetConnection = New OleDbConnection()
'Donner à la propriété ConnectionString les paramètres de
connexion
ObjetConnection.ConnectionString = strConn
'Ouvrir la connexion
ObjetConnection.Open()
'Instancer un objet Commande
ObjetCommand = New OleDbCommand(strSql)
'Instancer un objet Adapter
ObjetDataAdapter = New OleDbDataAdapter(ObjetCommand)
'initialiser l'objet Command
ObjetCommand.Connection() = ObjetConnection
'Avec l'aide de la propriété Fill du DataAdapter charger le
DataSet
ObjetDataAdapter.Fill(ObjetDataSet, "FICHEPATIENT")
'Mettre dans un Objet DataTable une table du DataSet
ObjetDataTable = ObjetDataSet.Tables("FICHEPATIENT")

```

Voir un enregistrement:

Routine affichant un enregistrement:

Une TextBox nommée 'Nom' et une TextBox 'Prenom' afficheront les noms et prénom des patients.

On rappelle que RowNumber est une variable contenant le numéro de la ligne en cours (allant de 0 à Rows.Count-1)

```
If RowNumber < 0 Then Exit Sub
```

'Lors de l'ouverture de la BD, s'il n'y a aucun enregistrement
`If` RowNumber > ObjetDataTable.Rows.Count - 1 `Then Exit` `Sub`
' ObjetTable.Rows(Numéro de lignes).Item(Nom de colonne)
donne le contenu d'un champ dans une ligne donnée
`Me`.Nom.Text = ObjetDataTable.Rows(RowNumber).Item("Nom")
`Me`.Prenom.Text =
ObjetDataTable.Rows(RowNumber).Item("Prenom")
'Item peut avoir en paramètre le nom de la colonne ou sont
index

Pour se déplacer:

Voir la ligne suivante par exemple:

RowNumber += 1 incrémente le numéro de la ligne en cours
puis on affiche.

RowNumber -= 1 pour la précédente.

RowNumber = 0 pour la première.

RowNumber = ObjetDataTable.Rows.Count - 1 pour la
dernière.

Modifier un enregistrement:

' Extraire l'enregistrement courant
ObjetDataRow =
ObjetDataSet.Tables("FICHEPATIENT").Rows(intRowNumber)

'Modifier les valeurs des champs en récupérant le contenu
des TextBox
ObjetDataRow("Nom") = `Me`.Nom.Text
ObjetDataRow("Prenom") = `Me`.Prenom.Text

'Pour modifier les valeurs changées dans le DataAdapter
ObjetCommandBuilder = `New`
OleDbCommandBuilder(ObjetDataAdapter)

'Mise à jour
ObjetDataAdapter.Update(ObjetDataSet, "FICHEPATIENT")

'On vide le DataSet et on le 'recharge' de nouveau.
ObjetDataSet.Clear()
ObjetDataAdapter.Fill(ObjetDataSet, "FICHEPATIENT")
ObjetDataTable = ObjetDataSet.Tables("FICHEPATIENT")



Attention: quand la commande Update est effectuée, si l'enregistrement ne répond pas au spécification de la base (doublon alors que c'est interdit, pas de valeur pour une clé

primaire, Champ ayant la valeur Null alors que c'est interdit..), une exception est levée; si vous ne l'avez pas prévue cela plante!!

Ajouter un enregistrement:

```
ObjetDataRow = ObjetDataSet.Tables("FICHEPATIENT").NewRow()
ObjetDataRow("Nom") = Me.Nom.Text
ObjetDataRow("Prenom") = Me.Prenom.Text
ObjetDataSet.Tables("FICHEPATIENT").Rows.Add(ObjetDataRow)
'Pour modifier les valeurs changées dans le DataAdapter
ObjetCommandBuilder = New
OleDbCommandBuilder(ObjetDataAdapter)
'Mise à jour
ObjetDataAdapter.Update(ObjetDataSet, "FICHEPATIENT")
'On vide le DataSet et on le 'recharge' de nouveau.
ObjetDataSet.Clear()
ObjetDataAdapter.Fill(ObjetDataSet, "FICHEPATIENT")
ObjetDataTable = ObjetDataSet.Tables("FICHEPATIENT")
```

Effacer l'enregistrement en cours:

```
ObjetDataSet.Tables("FICHEPATIENT").Rows(LineNumber).Delete()

ObjetCommandBuilder = New
OleDbCommandBuilder(objetDataAdapter)

ObjetDataAdapter.Update(objetDataSet, "FICHEPATIENT")
```

Fermer

```
'Objet connectée
ObjetConnection = Nothing
ObjetCommand = Nothing
ObjetDataAdapter = Nothing
'Objet déconnectée
ObjetDataSet = Nothing
ObjetDataTable = Nothing
ObjetDataRow = Nothing
```

Remplir un DataGridView avec un DataSet:

Une fois le dataSet existant, en UNE lignede code, on peut l'afficher dans un DataGridView.(Grille avec des lignes et des colonnes comme un tableur)

```
DataGrid1.SetDataBinding(ObjetDataSet, "FICHEPATIENT")
```

On peut modifier un DataSet:

Un DataSet est un groupe de données. On a vu qu'on pouvait le remplir avec une base de données mais on peut imaginer le créer de toute pièce et le remplir en créant des tables, des colonnes, des données.

Dans un DataSet on peut donc ajouter des tables, les columns, des enregistrements créer de toute pièce.

L'exemple suivant crée un objet DataTable, qui sera ajouter au DataSet.

```
private myDataSet As DataSet

' Créer une nouvelle DataTable.
Dim myDataTable As DataTable = new DataTable("ParentTable")
' Declaration de variables DataColumn et DataRow objects.
Dim myDataColumn As DataColumn
Dim myDataRow As DataRow

' Créer un nouveau DataColumn, lui donner un DataType, un
nom, divers valeur pour ses propriétés et l'ajouter à la
DatTable.
myDataColumn = New DataColumn()
myDataColumn.DataType = System.Type.GetType("System.Int32")
myDataColumn.ColumnName = "id"
myDataColumn.ReadOnly = True
myDataColumn.Unique = True
myDataTable.Columns.Add(myDataColumn)

' Créer une seconde column.
myDataColumn = New DataColumn()
myDataColumn.DataType = System.Type.GetType("System.String")
myDataColumn.ColumnName = "ParentItem"
myDataColumn.AutoIncrement = False
myDataColumn.Caption = "ParentItem"
myDataColumn.ReadOnly = False
myDataColumn.Unique = False
myDataTable.Columns.Add(myDataColumn)

'La colonne id doit être une clé primaire.
Dim PrimaryKeyColumns(0) As DataColumn
PrimaryKeyColumns(0)= myDataTable.Columns("id")
myDataTable.PrimaryKey = PrimaryKeyColumns

' Créer un objet DataSet
myDataSet = New DataSet()
' Ajouter la Table au DataSet.
myDataSet.Tables.Add(myDataTable)

' Créer 3 DataRow objects (3 lignes) et les ajouter à la
DatTable
```

```
Dim i As Integer
For i = 0 to 2
myDataRow = myDataTable.NewRow()
myDataRow("id") = i
myDataRow("ParentItem") = "ParentItem " + i.ToString()
myDataTable.Rows.Add(myDataRow)
Next i
End Sub
```

Travailler avec la Base MySQL

Pour travailler sur une base MySQL lisez le très bon didacticiel [MySQLDotNet](#) (sur developpez.com bien sur)



Retour



Index



Sommaire



Suivant

Site **LDF** Cours **VB.net**



	<h2>Charger des 6.6 contrôles avec des SGBD.</h2>	Suivant	Sommaire
--	---	-------------------------	--------------------------

Comment remplir des listBox ComboBox, DataGrid avec des tables venant de base de données.

Remplir une ListBox avec une colonne d'une table d'une BD:

Exemple:

Dans une base de données Accès nommé 'BaseNom', j'ai une table 'NomPatient' avec plusieurs colonnes, je veux afficher la liste des noms:

Champs Civilité	Champ Nom	Champ Prénom	Champ Numéro Interne
-----------------	-----------	--------------	----------------------

M.	Durand	Luc	1
Mme	Dupont	Josette	2
M.	Thomas	Guy	3

On peut utiliser un 'DataReader'

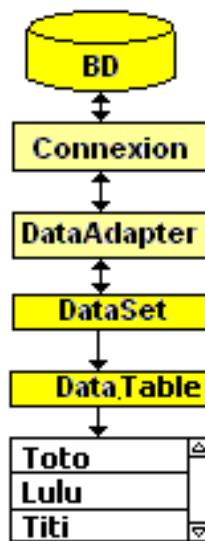
Voir la page 6-4 [Lire rapidement en lecture seule: le DataReader](#)

C'est de la 'lecture seule' très rapide.

On peut utiliser un 'DataSet', créer un 'DataTable' et la lier au contrôle.

Le DataSet est une représentation en mémoire des données; Une fois chargé on peut travailler en mode déconnecté.

Pour le remplir il faut un **DataAdapter**.



Bien sur il faut importer des espaces de nom:

```
Imports System
Imports System.Data
Imports System.Data.OleDb
```

Dans la zone déclaration de la fenêtre:

```
'Déclarer la connexion
Private ObjetConnection As OleDbConnection
' Déclaration l'Objet Commande
Private ObjetCommand As OleDbCommand
' Déclaration Objet DataAdapter
Private ObjetDataAdapter As OleDbDataAdapter
' Déclaration Objet DataSet
Private ObjetDataSet As New DataSet
'String contenant la 'Requête SQL'
Private strSql As String
' Déclaration Objet DataTable
Private ObjetDataTable As DataTable
```


'Paramètres de connexion à la DB
Private strConn As String

Dans une routine Button1_Click créer les divers objets et remplir la listbox.

```
Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
'Initialisation de la chaîne de paramètres pour la
connexion
strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" & "Data
Source= c:\Basenom.mdb;"
'Initialisation de la chaîne contenant l'instruction SQL
strSql = "SELECT FICHEPATIENT.* FROM FICHEPATIENT"
'Instanciatiion d'un Objet Connexion
ObjetConnection = New OleDbConnection
'Donner à la propriété ConnectionString les paramètres de
connexion
ObjetConnection.ConnectionString = strConn
'Ouvrir la connexion
ObjetConnection.Open()
'Instancer un objet Commande
ObjetCommand = New OleDbCommand(strSql)
'Instancer un objet Adapter
ObjetDataAdapter = New OleDbDataAdapter(ObjetCommand)
'initialiser l'objet Command
ObjetCommand.Connection() = ObjetConnection
'Avec l'aide de la propriété Fill du DataAdapter charger
le DataSet
ObjetDataAdapter.Fill(ObjetDataSet, "FICHEPATIENT")
'Mettre dans un Objet DataTable une table du DataSet
ObjetDataTable = ObjetDataSet.Tables("FICHEPATIENT")

'Indiquer au ListBox d'afficher la table "fichepatient"
(indiquer la source)
ListBox1.DataSource = ObjetDataSet.Tables("FICHEPATIENT")
'Indiquer quelle colonne afficher
ListBox1.DisplayMember = "NOM"
End Sub
```

Voilà, cela affiche la liste des noms.

Bien respecter les minuscules et majuscules dans les noms de tables et de champs+++

Récupérer la valeur d'un autre champ

On a vu que dans la table, chaque enregistrement avait un champ 'Nom' mais aussi un champ 'NumInt' (numéro interne) Dans les programmes, on a souvent besoin de récupérer le

numéro interne (un ID) quand l'utilisateur clique sur un des noms; le numéro interne servira à travailler sur ce patient.

Exemple: comment récupérer le numéro interne 3 quand l'utilisateur clique sur 'Thomas'?

Il faut indiquer à la ListBox que la Value de chaque ligne est 'NumInt' en utilisant la propriété

`ListBox1.ValueMember`.

Quand l'utilisateur clique sur une ligne de la ListBox, cela déclenche l'évènement `ListBox1_SelectedIndexChanged`, là on récupère la valeur de NumInt correspondant; elle se trouve dans `ListBox1.SelectedValue`. (C'est un Int32)

Attention:

Les exemples de Microsoft ne fonctionnent pas!! car, on n'explique nulle part l'importance de l'ordre des lignes remplissant la ListBox.

C'est DataSource qui semble déclencher le chargement de la ListBox avec la prise en compte de DisplayMember et ValueMember.

Si on fait comme Microsoft (renseigner `ListBox1.DataSource` en premier):

```
ListBox1.DataSource = ObjetDataSet.Tables("FICHEPATIENT")
ListBox1.DisplayMember = "NOM"
ListBox1.ValueMember = "NUMINT"
```

`ValueMember` ne fonctionne pas bien, (liaison tardive?) et `ListBox1.SelectedValue` retourne un objet de type `DataRowView` impossible à utiliser.

Il faut donc renseigner le DataSource en dernier.

```
ListBox1.DisplayMember = "NOM"
ListBox1.ValueMember = "NUMINT"
ListBox1.DataSource = ObjetDataSet.Tables("FICHEPATIENT")
```

Dans ce cas `ListBox1.SelectedValue` contient bien un Int32 correspondant au 'NumInt' sélectionné.

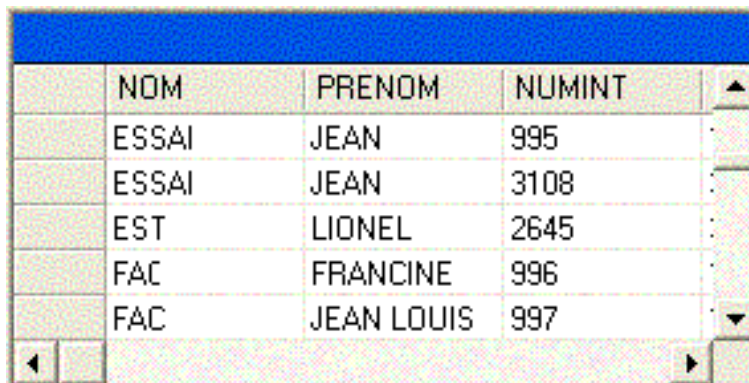
'Ensuite on peut récupérer sans problème NumInt (et l'afficher par exemple dans une TextBox)

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As Object, ByVal e As System.EventArgs) Handles
ListBox1_SelectedIndexChanged
Dim o As New Object
If ListBox1.SelectedIndex <> -1 Then
TextBox1.Text = CType(ListBox1.SelectedValue, String)
End If
End Sub
```

Remplir un DataGrid avec un DataSet:

Il faut créer une [Connection](#), un [DataAdapter](#) et un [DataSet](#), puis en **UNE ligne de code**, on peut l'afficher dans un [DataGrid](#). (Grille avec des lignes et des colonnes comme un tableur)

```
DataGrid1.SetDataBinding(ObjetDataSet, "FICHEPATIENT")
```



The screenshot shows a DataGrid control with a blue header bar. The grid contains five columns: 'NOM', 'PRENOM', 'NUMINT', and an empty column with a scroll arrow. There are six rows of data. The first row is a header row with the column names. The following five rows contain patient records.

	NOM	PRENOM	NUMINT	
	ESSAI	JEAN	995	
	ESSAI	JEAN	3108	
	EST	LIONEL	2645	
	FAC	FRANCINE	996	
	FAC	JEAN LOUIS	997	

Les lignes, colonnes, nom des champs, ascenseurs... sont créés automatiquement!! **Génial.**

On peut aussi remplir le DataSet avec:

```
DataGrid1.DataSource = ObjetDataSet  
DataGrid1.DataMember = "FICHEPATIENT"
```

On peut modifier l'aspect du DataGrid dans la fenêtre de propriété ou en utilisant la mise en forme automatique (lien en bas de la fenêtre de propriétés.)

Mise en forme automatique

Formats :

Professionnel 1
Professionnel 2
Professionnel 3
Professionnel 4
Classique
Simple
Couleur 1
Couleur 2
Couleur 3
Couleur 4

Aperçu :

Grille de données d'exemple

	Prénom	Prénom
▶	Robert	Brown
	Nate	Sun
	Carole	Poland
*		

OK

Annuler



Site **LDF** Cours **VB.net**



7.1	<h2>Différence entre VB6 et VB.NET</h2>		Suivant	Sommaire
-----	---	--	-------------------------	--------------------------

Cela concerne surtout ceux qui passent de VB6 à VB.NET, pour les autres c'est une révision.

Les petits nouveaux qui ne connaissaient pas VB6 (précédente version de VB) ne doivent pas lire ce qui est en vert.

Les Objets

En VB.NET **tout est objet**: les fenêtres, les contrôles, les variables....

Set et **Let** ne sont plus pris en charge.

Les objets peuvent être assignés par une simple opération d'assignation :

```
Object1 = Object2
```

Pour définir une propriété par défaut d'un objet, vous devez désormais référencer explicitement la propriété. Exemple :

```
Object1.Text = Object2.Text
```

Vous pouvez définir vous même un nouveau type d'objet, une **Classe**.

Puis instancer des objets à partir de cette Classe.

VB.NET permet une **vraie programmation objet**: héritage, polymorphisme, surcharge, Interface ...

Les formulaires ou fenêtres:

On se rend compte que quand on dessine une fenêtre Form2 par exemple, VB crée une nouvelle classe 'Class Form2'

```
Public Class Form2
```

```
Inherits System.Windows.Forms.Form
```

```
End Class
```

Elle hérite de **System.Windows.Forms.Form**: on le voit bien dans le code.

Pour utiliser cette fenêtre, il faut créer une instance de cette fenêtre à l'aide de la Classe :

On tape `Dim f As New Form2()`, on crée une instance de la Class Form2.

Enfin la fenêtre sera ouverte grâce à la méthode ShowDialog ou Show.

Comme pour un Objet en général, la fenêtre créée sera visible dans sa portée. Si une fenêtre est instancée dans une procédure, l'objet fenêtre f ne sera visible que dans cette procédure.

Comment passer de VB6 à VB.net?

- Avant **en VB6**, on avait:

```
Form2.Load
```

```
Form2.Show
```

- Avec le programme de conversion VB6=>VB.Net on a:

```
Form2.DefInstance.Show()
```

car il ajoute une routine qui crée automatiquement une instance de form2:

```
#Region "Prise en charge de la mise à niveau"  
Private Shared m_vb6FormDefInstance As form2  
Private Shared m_InitializingDefInstance As Boolean  
Public Shared Property DefInstance() As form2  
Get  
If m_vb6FormDefInstance Is Nothing OrElse  
m_vb6FormDefInstance.IsDisposed Then  
m_InitializingDefInstance = True  
m_vb6FormDefInstance = New form2()  
m_InitializingDefInstance = False  
End If  
DefInstance = m_vb6FormDefInstance  
End Get  
Set  
m_vb6FormDefInstance = Value  
End Set
```

```
End Property
#End Region
```

¶ En fait il faut mieux avec **VB.net** écrire:

```
Dim F As New Form2

F.Show
```

On remarque que Load n'existe plus, par contre, le Dim crée le formulaire sans l'afficher, c'est à peu près équivalent..

Les Forms ont **2 contrôles menu**:

Les MainMenu

Les ContextMenu.

Visual Basic .NET ne prend pas en charge la méthode **Form.PrintForm**.

Les Contrôles

La plupart des objets ne possèdent **plus de propriétés par défaut**.

En VB6:

```
Dim str As String = TextBox1
```

Maintenant il faut écrire:

```
Dim str As String = TextBox1.Text
```

Visual Basic .NET ne **prend pas en charge le contrôle conteneur OLE** .

Il n'existe **pas de contrôle carré rectangulaire ligne** . On peut les remplacer sous la forme d'étiquettes (Label), tandis que **les ovales et les cercles qui n'existent pas** non plus, ne peuvent pas être mis à niveau.

Visual Basic .NET est doté d'un nouvel ensemble de commandes graphiques faisant partie de **GDI+**. **Circle, CLS, PSet, Line et Point** n'existent plus. Étant donné que le nouveau modèle objet est assez différent de celui de Visual Basic 6.0, il faut tout réécrire.

Visual Basic .NET **ne prend pas en charge l'échange dynamique de données (DDE, Dynamic Data Exchange)**.

Bien que Visual Basic .NET prenne en charge la fonctionnalité de glisser-déplacer, le modèle objet est différent de celui de Visual Basic 6.0. Il faut tout réécrire.

Le .NET Framework est doté d'un objet **Clipboard** amélioré (**System.Windows.Forms.Clipboard**) qui offre plus de fonctionnalités et prend en charge un plus grand nombre de formats de Presse-papiers que l'objet **Clipboard** de Visual Basic 6.0. Il faut tout réécrire.

Les groupe de contrôle n'existent plus. Il y a des moyens de les remplacer.

Les Variables.

Option Explicit étant activé par défaut, toutes les variables doivent être déclarées avant d'être utilisées.

Le type de données **Variant** n'existe plus. Celui-ci a été remplacé par le type **Object**.

Le type de données **Integer** est désormais de 32 bits ; le type de données **Long** est de 64 bits.

On peut utiliser les **type Booléens** qui ne peuvent prendre que 2 valeurs : **True** et **False** (pas 0 et -1)

En VB.NET **Option Strict** étant activé, il faut **convertir explicitement une donnée d'un type vers un autre si c'est nécessaire** .

```
Response.Write("Le total est " & CStr(total))
```

on attend des String, la variable total qui est numérique est donc transformée en String par CStr.

Les variables créées dans la même instruction **Dim** seront du même type. Par exemple, dans VB.NET, l'instruction Dim i, j, k As Integer crée chacun des trois objets (i, j, et k) comme **Integer**. Les versions précédentes de VB créaient i et j comme **Variants** et k comme **Integer** (c'était nul!!).

Il existe une **type Char** qui contient un seul caractère.

Le type **Currency** est remplacé par le type **Decimal**.

Les String de longueur fixe n'existent plus. Il y a quelques ficelles pour contourner cela mais bonjour la simplicité!!

Les String et Char sont en **Unicode** (chaque caractère est codé sur 2 octets).

Une variable peut avoir une portée locale, publique ou, et c'est nouveau, **une portée au niveau d'un bloc**:

```
For i=0 to 100
Dim Str As String 'Str est visible uniquement entre For et Next
...
next i
```


Les Tableaux

Le premier élément d'un tableau a l'indice 0 obligatoirement; plus d'Option Base.

On peut initialiser un tableau en même temps qu'on le déclare:

```
Dim Tableau(3) As Integer ={2,3,5}
```

A noter que ce tableau contient 4 élément d'index 0, 1, 2, 3.

Dim S(4 to 15) n'est plus accepté.

Dim est utilisé pour la déclaration initiale, **Redim** pour le redimensionnement uniquement.

Les tableaux font partie de la **Classe Array**, ce qui autorise l'utilisation de méthodes bien pratiques: **Sort** par exemple trie automatiquement le tableau.

Les Collections:

Elles sont omniprésentes. C'est fondamental de bien comprendre leur fonctionnement: Ce sont des listes ayant un nombre d'élément non défini, on peut y ajouter des éléments, en retirer, il y a des méthodes pour trier, rechercher..

Cela peut être :

Des listes d'objet: ArrayList

Des listes de booléens: BitArray

Des listes Clé-Valeur :HashTable

Des Queue

Des Piles :Stack

La notion de collections est généralisée et utilisée dans beaucoup d'objet: Une ListBox possède une collection d'Item (les éléments de la listBox)

Les Structures:

Elles remplacent les "Types définies par l'utilisateur".

```
Structure MaStructure  
Public i As Integer
```

```
Public c As String
End Structure
```

Les Fonctions et Sub

Les parenthèses sont désormais requises autour de la liste de paramètres dans tous les appels de méthodes y compris pour les méthodes qui ne prennent pas de paramètres. Exemple :

```
If Flag = False Then
    AfficheMessage()
End If
```

Par défaut, les arguments sont passés **par valeur**, et non pas référence. Si vous voulez passer des arguments par référence, vous devez utiliser le mot clé **ByRef** devant l'argument comme dans l'exemple suivant :

```
Call MaFunction(argbyvalue, ByRef argbyref)
```

Il peut y avoir des **paramètres optionnels**.

Return est un nouveau mot clé qui permet à une fonction de retourner une valeur.

```
Private Function MaFonction (Byval Chaine As String)
    Return Chaine.ToLower()
End Function
```

□

Dans le Code:

Simplification d'écriture:

`A= +=2` est équivalent à `A=A+2`

Nouveau type de boucle

```
While condition
```

```
End While
```

Boucle tant que condition est vraie.

Wend n'existe plus.

Le **Multithreading** est possible.

Gestion des erreurs:

La gestion des erreurs est structurées:

Elle utilise

```
Try
    Code a tester
Catch
    interception de l'erreur
Finally
    suite
End Try
```

On error goto reste utilisable.

Les graphiques:

En GDI (VB6) on utilisait les handles(HDC) pour spécifier un image.

En GDI+ (VB.Net) on travaille sur les **Graphics** et leurs méthodes.
Graphics.DrawLine..

L'unité de mesure est le pixels. (**Plus de Twips**)

Les bases de données:

Visual Basic .NET contient une version améliorée des objets de données actifs (ADO, *Active Data Objects*) appelée **ADO.NET**.

DAO, RDO et ADO peuvent toujours être utilisés dans du code Visual Basic .NET, avec toutefois quelques petites modifications . Toutefois, Visual Basic .NET ne prend pas en charge la liaison de données DAO et RDO aux contrôles ou contrôles de données ni la connexion utilisateur RDO.

Les Classes :

La syntaxe des propriétés de classe a changé et ne contient plus **Property Let**, **Property Get**, et **Property Set**. La nouvelle syntaxe des propriétés est analogue à celle de C#.

```
Public Property ThisProperty As String
    Get
        ThisProperty = InternalValue
    End Get
    Set
        InternalValue = value
    End Set
End Property
```

Les classes sont totalement objet et acceptent le polymorphisme, la surcharge, l'héritage..

GOSUB et ON GOSUB n'existent plus

Il faut remplacer une routine qui était appelée par gosub par une sub ou une **fonction**.

Remplacer

```
Sub Mon Programme
..
Gosub Routine
..
End SuB
Routine:
    Code de la routine
Return
```

Par

```
Sub Mon Programme
..
Call Routine()
..
End Sub
Sub Routine()
    code de la routine
End Sub
```

Il faudra simplement faire attention aux variables, les variables locales à Mon Programme ne seront pas accessibles dans la Routine.

Pour **On Gosub**, il faut remplacer par un **SelectCase**.

Les Timers

S'agissant du contrôle Timer, le fait d'affecter à la propriété **Interval** la valeur 0 ne désactive pas la minuterie ; l'intervalle est réinitialisé à 1. Dans vos projets Visual Basic 6.0, vous devez affecter à la propriété **Enabled** la valeur **False** plutôt que d'affecter à **Interval** la valeur 0.

Conversion VB6 vers VB.NET

Il existe **un outil de conversion** (Menu Fichier, Ouvrir, Convertir, Assistant de mise à niveau de VB.NET) pour convertir un source VB6 en VB.NET.

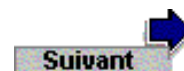
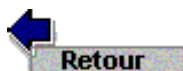
Le problème est qu'il donne un code inutilisable avec:

- Conversion des instructions VB6=>VB.NET quand il le peut.
- Conversion en utilisant une classe de compatibilité contenant des instructions spécifiques à VB6 (qui ne sont PAS du VB.NET) produisant un code hybride. Cette classe de compatibilité disparaîtra probablement dans les prochaines versions.
- Des instructions impossible à convertir automatiquement et qui seront à réécrire à la main.

Il faut convertir les programmes VB6 avec l'outil de conversion **pour voir ce que cela donne**: c'est instructif de voir le nouveau code.

Mais il faut réécrire totalement une bonne partie du code: l'appel des fenêtres en particulier..

Il faut rapidement ne pas utiliser du tout la classe de compatibilité VB 6 , éviter les instructions héritées de VB6, privilégier l'usage des classes du Framework.



site **LDF** Cours **VB.net**



7.2	Règles de bonne programmation.		<div style="display: flex; justify-content: space-around;"> Suivant Sommaire </div>
------------	---	--	---

Pour faire un code solide, éviter les bugs, avoir une maintenance facile, il faut suivre quelques règles.

Au niveau du projet

Découper un traitement complexe en plusieurs petites routines effectuant chacune une fonction précise.

Découper les différentes fonctions du logiciel en Module et procédures, ou en Objet (Créer des Classes dont les méthodes seront les divers routines). voir [Programmation 'fonctionnelle' ou 'objet'](#)

Séparer l'interface utilisateur et l'applicatif.

Exemple: un formulaire affiche les enregistrements d'une base de données:

Créer:

- Les fenêtres dont le code gère uniquement l'affichage. C'est l'interface utilisateur ou IHM (Interface Homme Machine)
- une Classe gérant uniquement l'accès au base de données.

Cela facilite la maintenance: si on désire modifier l'interface, on touche au fenêtre et pas du tout à la Classe base de données.

Architecture à 3 niveaux.

Elle peut être nécessaire dans certains programmes, les 3 niveaux sont:

- Application ,interface.
- Logique.
- Données.

Exemple: Un formulaire affiche certains enregistrements d'une base de données.

- **L'interface** affiche les enregistrements.
- Les classes ou modules '**logiques**' déterminent les bons enregistrements.
- Les classes ou modules **données** vont chercher les données dans la base de données.

Si au lieu de travailler sur une base Access, je travaille sur une base SQLServer, il suffit de réécrire la troisième couche.

Dans un module:

Respecter l'ordre suivant :

1. Instructions **Option**
2. Instructions **Imports**
3. Procédure **Main**
4. Instructions **Class, Module** et **Namespace**, le cas échéant

Dans une Class

1. Instructions **Declare**
2. Déclaration des variables (Public Private)
3. **Sub** ou **Function**

Sous peine d'erreurs à la compilation.

Rendre le code lisible:

- **Ajoutez des commentaires.**

Pour vous, pour les autres.

Au début de chaque routine, Sub, Function, Classe , noter en commentaire ce qu'elle fait et quelles sont les caractéristiques des paramètres:

- Le résumé descriptif de la routine, la Sub ou Function.
- Une description de chaque paramètre.
- La valeur retournée s'il y en a une.
- Une description de toutes les exceptions..
- Un exemple d'utilisation

- ¶ Une explication sur le fonctionnement de la routine.

Ne pas ajouter de commentaire en fin de ligne (une partie ne sera pas visible) mais plutôt avant la ligne de code. Seule exception ou on utilise la fin de ligne: les commentaires après les déclarations de variable.

```
Dim i As Integer      'Variable de boucle
```

```
'Parcours du tableau à la recherche de..
```

```
For i=0 To 100
```

```
...
```

Paradoxalement, trop de commentaires tue le code autant que le manque de commentaires. Pour éviter de tomber dans le tout ou rien, fixons nous quelques règles:

Commentez le début de chaque
Sub, Fonction, Classe

Commentez toutes les
déclarations de variables

Commentez toutes les branches
conditionnelles

Commentez toutes les boucles

- ¶ **Choisissez les noms de procédures et de variables avec soins:** leur nom doit être explicite. Microsoft propose quelque règles:

Routines

Utilisez la casse Pascal (**CalculTotal**) pour les noms de routine (la première lettre de chaque mot est une majuscule).

Évitez d'employer des noms difficiles pouvant être interprétés de manière subjective, notamment Analyse() pour une routine ou YYB8 pour une variable.

Dans les objets, il ne faut pas inclure des noms de classe dans les noms de propriétés Patient.PatientNom est inutile, utiliser plutôt Patient.Nom.

Utilisez les verbe/nom pour une routine :
CalculTotal().

Variables

Pour les noms de variables, utilisez la casse selon laquelle la première lettre des mots est une majuscule, sauf pour le premier mot (**i**Nombre**P**atient); noter ici que la première lettre indique le type de la variable (Integer), elle peut aussi indiquer la portée(**g**Ttotal pour une variable globale).

Ajoutez des méthodes de calcul (Min, Max, Total) à la fin d'un nom de variable, si nécessaire.

Les noms de variable booléenne doivent contenir Is qui implique les valeurs True/False, par exemple fileIsFound.

Évitez d'utiliser des termes tels que Flag lorsque vous nommez des variables d'état, qui diffèrent des variables booléennes car elles acceptent plus de deux valeurs. Plutôt que documentFlag, utilisez un nom plus descriptif tel que documentFormatType.

Même pour une variable à courte durée de vie utilisez un nom significatif. Utilisez des noms de variable d'une seule lettre, par exemple i ou j, pour les index de petite boucle uniquement.

N'utilisez pas des nombres ou des chaînes littérales telles que For i = 1 To 7. Utilisez plutôt des constantes par exemple For i = 1 To DAYSINWEEK, pour simplifier la maintenance et la compréhension.

Tables

Pour les tables, utilisez le singulier. Par exemple, utilisez table 'Patient' plutôt que 'Patients'.

N'incorporez pas le type de données dans le nom d'une colonne.

Divers

Minimisez l'utilisation d'abréviations,

Lorsque vous nommez des fonctions, insérez une description de la valeur retournée, notamment `GetCurrentWindowDirectory()`.

Évitez de réutiliser des noms identiques pour divers éléments.

Évitez l'utilisation d'homonymes et des mots qui entraînent souvent des fautes d'orthographe.

Évitez d'utiliser des signes typographiques pour identifier des types de données, notamment `$` pour les chaînes ou `%` pour les entiers.

Un nom doit indiquer la signification plutôt que la méthode.

□ **Eclaircir, aérer le code:**

Eviter plusieurs instructions par ligne.

Ajouter quelques lignes blanches.

Décaler à droite le code contenu dans une boucle ou une section `If.. End If`:

Une mise en retrait simplifie la lecture du code, par exemple :

```
If ... Then
    If ... Then
        ...
    Else
        ...
    End If
Else
    ...
End If
```

Forcer la déclaration des variables et les conversions explicite:

`Option Explicit` étant par défaut à `On`, **toute variable utilisée doit être déclarée**. Conserver cette option. Cela évite les erreurs liées aux variables mal orthographiées.

Si `Option Strict` est sur `On`, seules les conversions de type effectuées explicitement sur les variables

seront autorisées. Le mettre sur On.

Voir [Option Strict et Option Explicite](#)

Utiliser des constantes ou des énumérations:

L'usage de constantes facilite les modifications.

Exemple : un programme gère des utilisateurs;:

Faire:

Créer une constante contenant le nombre maximum d'utilisateurs.

```
Const NombreUtilisateur= 20
Dim VariableUtilisateur (NombreUtilisateur) 'on utilise
NombreUtilisateur et non 20
For i= 0 To NombreUtilisateur-1
Next i
```

Plutôt que:

```
Dim VariableUtilisateur (20)
For i= 0 To 19
Next i
```

Si ultérieurement on veut augmenter le nombre d'utilisateurs possibles à 50, il suffit de changer une seule ligne:

```
Const NombreUtilisateur= 50
```

Utiliser les constantes VB, c'est plus lisible:

Form1.BorderStyle=2 est à éviter

```
Form1.BorderStyle= vbSizable c'est mieux
```

Vérifier la validité des paramètres que reçoit une Sub ou Function

Vous pouvez être optimiste et ne pas tester les paramètres reçus par votre Sub. Les paramètres envoyés seront toujours probablement bons!! Bof un jour vous ferez une erreur, ou un autre n'aura pas compris le type de paramètre à envoyer et cela plantera!!

Donc, il faut vérifier la validité des paramètres.

On peut le faire au fur et à mesure de leur utilisation dans le code, il est préférable de faire toutes les vérifications en début de Sub.

Se méfier du passage de paramètres 'par valeur' ou par 'référence':

Par défaut les paramètres sont envoyés 'par valeur' vers une procédure. Aussi, si la variable contenant le paramètre est modifiée, cela ne modifie pas la valeur de la variable de la procédure appelante.

Si on a peur de se tromper utilisons 'ByVal' et 'ByRef' dans l'entête de la Sub ou de la Fonction.

Les Booléens sont des Booléens:

Utiliser une variable Integer pour stocker un Flag dont la valeur ne peut être que 'vrai' ou 'faux' et donner la valeur 0 ou -1 est à proscrire.

Faire:

```
Dim Flag As Boolean
```

```
Flag=True
```

(Utiliser uniquement True et False)

Eviter aussi d'abréger à la mode Booléens ce qui n'en est pas.

```
Dim x,y As Integer
```

```
If x And y then
```

 (pour tester si x et y sont = 0) est à éviter.

Faire plutôt:

```
If x<>0 And y <>0
```

Utiliser les variables Date pour stocker les dates:

Ne pas utiliser de type Double.

```
Dim LaDate As Date
```

```
LaDate=Now
```

Ne faire aucune confiance à l'utilisateur du logiciel:

Si vous demandez à l'utilisateur de saisir un entier entre 1 et 7.

Vérifier:

qu'il a tapé quelque chose!!

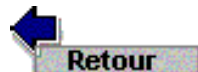
Qu'il a tapé une valeur numérique.

Que c'est un entier.

Que c'est supérieur à 0 et inférieur à 8.

Accorder les moindres privilèges:

Ne permettre de saisir que ce qui est valide.



Site **LDF** Cours **VB.net**



7.3	Optimiser en vitesse.		
		Suisvant	Sommaire

VB.NET est t-il rapide?

Comment VB.NET est situé en comparaison avec les autres langages de programmation?

Le site OsNews.com publie les résultats d'un petit benchmark comparant les performances d'exécution sous Windows de plusieurs langage de programmation. Les langages .NET - et donc le code managé en général - n'ont pas à rougir devant Java, pas plus que face au langage C compilé grâce à GCC. Voici un aperçu des résultats chiffrés (valeurs les plus faibles = les meilleures performances) :

	int	long	double	trig	I/O	TOTAL
Visual C++	9.6	18.8	6.4	3.5	10.5	48.8
Visual C#	9.7	23.9	17.7	4.1	9.9	65.3
gcc C	9.8	28.8	9.5	14.9	10.0	73.0
Visual Basic	9.8	23.7	17.7	4.1	30.7	85.9
Visual J#	9.6	23.9	17.5	4.2	35.1	90.4
Java 1.3.1	14.5	29.6	19.0	22.1	12.3	97.6
Java 1.4.2	9.3	20.2	6.5	57.1	10.1	103.1
Python/Psycy	29.7	615.4	100.4	13.1	10.5	769.1
Python	322.4	891.9	405.7	47.1	11.9	1679.0

- [Nine Language Performance Round-up: Benchmarking Math & File I/O \[OsNews.com\]](#)

Article publier sur www.DotNet-fr.org

VB.NET est-il plus rapide que VB6?

Exemple No1

Sur une même machine P4 2.4 G faisons tourner un même programme: 2 boucles imbriquées contenant une multiplication, l'addition à un sinus et l'affichage dans un label:

En Vb.Net

```
Imports System.Math
Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
Dim i As Integer
Dim j As Integer
Dim k As Integer
For i = 0 To 100
For j = 0 To 1000
Label5.Text = (k * 2 + Sin(4)).ToString : Label5.Refresh()
k = k + 1
Next
Next
End Sub
```

35 secondes dans l'IDE, **25 secondes avec un exécutable** après compilation.

En utilisant des 'Integer' ou des 'Long', il y a peu de différence.

En VB6

```
Private Sub Command1_Click()
Dim i As Long
Dim j As Long
Dim k As Long
For i = 0 To 100
For j = 0 To 1000
Label1.Caption = Str(k * 2 + Sin(4)): Label1.Refresh
k = k + 1
Next
Next
```

End Sub

9 secondes dans l'IDE , 7 secondes avec un exécutable après compilation.

Dur, dur 25 s pour VB.NET, 7 s pour VB6.

Exemple No2

Sur une même machine P4 2.4 G faisons tourner un même programme: On crée un tableau de 10000 String dans lequel on met des chiffres Puis on trie le tableau.

En Vb.Net

```
Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
Dim i As Integer
Dim A(10000) As String
For i = 9999 To 0 Step -1
    A(i) = (9999 - i).ToString
Next i
Array.Sort(A)
Label1.Text = "ok"
End Sub
```

< 1 seconde

En VB6

```
Private Sub Command1_Click()

Dim i As Integer
Dim A(10000) As String
Dim j As Integer
Dim N As Integer
Dim Temp As String
N = 9999
'remplir le tableau
For i = 9999 To 0 Step -1
    A(i) = Str(9999-i)
Next i

'trier
For i = 0 To N - 1
    For j = 0 To N - i - 1

        If A(j) > A(j + 1) Then
            Temp = A(j): A(j) = A(j + 1): A(j + 1) = Temp
        End If

    Next j

Next i
```


Next i
End Sub

35 secondes

Moins d'une seconde avec VB.NET, 35 secondes en VB6.
La méthode 'Sort' est hyper plus rapide que la routine de tri.

(Pour être honnête, il faut dire que mon test n'est pas rigoureux car le tri VB.NET est probablement un tri 'rapide' alors qu'en VB6 la routine basic n'est pas optimisée, je ne compare donc pas les mêmes routines.)

En conclusion:

La couche du Framework semble **ralentir considérablement la vitesse du code.**

Mais, en VB.net, **il faut raisonner différemment** et utiliser judicieusement les classes et les méthodes au lieu de taper de longues routines.

Cela fait que en VB.Net:

Le code est plus court et compact (moins de temps de développement)

Le code est plus rapide.

Comment accélérer une application VB.net?

Utilisation des nouvelles fonctionnalités:

Il faut **raisonner différemment** et utiliser judicieusement les classes et les méthodes au lieu de taper de longues routines.

Exemple:

On l'a vu plus haut **La méthode 'Sort' d'un tableau est hyper plus rapide que la routine de tri écrite en code.**

Choix des variables:

Sur les ordinateurs actuels:

Pour les entiers les **Integer** sont les plus rapides car le processeur calcul en Integer .Viennent ensuite les **Long**, **Short**, et **Byte**.

Dans les nombres en virgule flottante, les **Double** sont les plus rapides car le processeur à virgule flottante calcul en Double, ensuite se sont les **Single** puis les **Decimal**.

Si c'est possible utiliser les entiers plutôt que les nombres en virgules flottantes.

Exemple pour stocker les dimensions d'une image, on utilisera les pixels: l'image aura un nombre entier de pixels et on peut ainsi utiliser une variable Integer, alors que si on utilise les centimètres on devra travailler sur des fractionnaires donc utiliser par exemple des Singles.

L'usage de constantes est plus rapide que l'usage de variable, car la valeur d'une constante est directement compilée dans le code.

Pour stocker une valeur, une variable est plus rapide qu'une propriété d'objet.

Tableau:

Le CLR est optimisé pour les tableaux unidimensionnel.

L'usage des tableaux de tableau 'A(9),(9)' est plus rapide que les tableaux multidimensionnels 'A(9,9)'.

Pour rechercher un élément dans un ensemble l'élément à partir de son index , utilisez un tableau (l'accès à un élément d'index i est plus rapide dans un tableau que dans une collection)

Collections:

Si on ne connaît pas le nombre d'éléments maximum et que l'on doit ajouter, enlever des éléments, il vaut mieux utiliser une collection (ListArray) plutôt qu'un tableau avec des Dim Redim Preserve. Mais attention une collection est composée d'objet, ce que est lent.

Pour rechercher un élément dans un ensemble l'élément à partir d'une clé (KeyIndex) , utilisez une collection (l'accès à un élément ayant la clé X est plus rapide dans une collection que dans un tableau; dans un tableau il

faut en plus écrire la routine)

Eviter la déclaration de variables 'Objet' et les liaisons tardives:

Eviter de créer des variables Objet:

Pour créer une variable et y mettre une String:

```
Dim A crée un 'Objet' A
```

Il est préférable d'utiliser:

```
Dim A As String
```

La gestion des objets est plus lente que la gestion d'une variable typée.

Il faut aussi éviter **les liaisons tardives**: Une liaison tardive consiste à utiliser une variable Objet et à l'exécution, donc tardivement, lui assigner une String ou un Objet ListBox par exemple. Dans ce cas, à l'exécution, VB doit analyser de quel type d'objet il s'agit et le traiter, alors que si la variable a été déclarée d'emblée comme une String ou une ListBox, VB a déjà prévu le code nécessaire en fonction du type de variable. **Utilisez donc des variables typées.**

Utiliser les bonnes 'Option':

`Option Strict On` permet de convertir les variables de manière explicite et accélère le code.

`Option Compare Binary` accélère les comparaisons et les tris (la comparaison binaire consiste à comparer les codes unicode des chaînes de caractère).

Pour les fichiers utiliser System.IO:

L'utilisation des **System.IO** classes accélère les

opérations sur fichiers (en effet, les autres manières de lire ou d'écrire dans des fichiers comme les FileOpen font appel à System.IO: autant l'appeler directement!!) :

- q **Path, Directory, et File**
- q **FileStream** pour lire ou écrire
- q **BinaryReader** and **BinaryWriter** pour les fichiers binaires.
- q **StreamReader** and **StreamWriter** pour les fichiers texte.

Utiliser des buffers entre 8 et 64K

Opérations:

Si possible:

Utiliser :"

Pour faire une vraie division on utilise l'opérateur '/'

Si on a seulement besoin du quotient d'une division (et pas du reste ou du résultat fractionnaire) on utilise '\', c'est beaucoup plus rapide.

Utiliser :"+="

A+= 2 est plus rapide que A= A+2

Utiliser :AndAlso et ElseOr

AndAlso et ElseOr sont plus rapide que And et Or.

(puisque la seconde expression n'est évaluée que si nécessaire)

Utiliser :With End With

With.. End With accélère le code:

```
With Form1.TextBox1
    .BackColor= Red
    .Text="BoBo"
    .Visible= True
End With
```

est plus rapide que

```
Form1.TextBox1.BackColor= Red
Form1.TextBox1.Text="BoBo"
Form1.TextBox1.Visible= True
```

car Form1.TextBox1 est 'évalué' 3 fois au lieu de 1 fois.

En mettre le moins possible dans les boucles:

Soit un tableau J(100,100) d'entiers:
Soit un calcul répété 100 000 fois sur un élément du tableau, par exemple:

```
For i=1 to 100000
    R=i*J(1,2)
next i
```

On va 100000 fois chercher un élément d'un tableau, c'est toujours le même!

Pour accélérer la routine (c'est plus rapide de récupérer la valeur d'une variable simple plutôt d'un élément de tableau), on utilise une variable intermédiaire P:

```
Dim P as integer
P=J(1,2)
For i=1 to 100000
    R=i*P
next i
c'est plus rapide.
```

De la même manière si on utilise une propriété (toujours la même) dans une boucle, on peut stocker sa valeur dans une variable car l'accès à une variable simple est plus rapide que l'accès à une propriété.

Eviter aussi les Try Catch dans des grandes boucles.

Comment accélérer quand on utilise des 'String':

Exemple d'une opération coûteuse en temps:

```
Dim s As String = "bonjour";  
  
s += "mon" + "ami";
```

En réalité le Framework va créer 3 chaînes en mémoire avec toutes les pertes en mémoire et en temps que cela implique.

Pour effectuer des opérations répétées sur les string, le framework dispose donc d'une classe spécialement conçue et **optimisée** pour ça : `System.Text.StringBuilder`.

Pour l'utiliser, rien de plus simple

```
System.Text.StringBuilder sb = new  
System.Text.StringBuilder();  
  
sb.Append("bonjour");  
  
sb.Append("mon ami");  
  
string s = sb.ToString();
```

La méthode `ToString` de la classe `StringBuilder` renvoi la chaîne qu'utilise en interne l'instance de `StringBuilder`.

Comment accélérer l'affichage?

Formater le plus vite possible:

Pour mettre en forme des nombres et les afficher `Format` est puissant, mais si on peut utiliser `ToString` c'est plus rapide (`ToString` est aussi plus rapide que `Cstr`).

`ChrW` utilisé pour afficher un caractère(et `AscW`) sont plus rapide que `Chr` et `Asc` car il travaille directement sur les Unicode.

Précharger les fenêtres et les données.

Quand une fenêtre en ouvre une autre, le temps de chargement est long, l'utilisateur attend!

Solution:

En début de programme précharger les fenêtres en les rendant invisible. Lors de l'utilisation de ces fenêtres il suffira de les rendre visible, ce qui est plus rapide que de les charger.

Certaines données (liste..)doivent être chargées une fois pour toute, le faire en début de programme, lors de l'affichage de la fenêtre 'Splash' par exemple(Celle qui contient une belle image et qui s'ouvre en premier)

Afficher les modifications en une fois:

A chaque fois que l'on fait une modification de propriété (couleur, taille..) ou de contenu (texte dans un TextBox) Vb affiche chaque modification . Si on modifie tout et que l'on re-affiche tout cela va plus vite.

Rendre l'objet inactif, faire toutes les modifications puis réactiver.

Pour le cas du TextBox ne pas faire.

```
TextBox1.Text = TextBox1.Text + "Bonjour"  
TextBox1.Text = TextBox1.Text + "Monsieur"
```

faire:

```
Dim T as string  
T = "Bonjour"  
T &= "Monsieur"  
TextBox1.Text = T
```

Le texte est affiché en une fois.

Afficher en 2 fois:

A l'inverse pour ne pas faire attendre un affichage très long, afficher le début (l'utilisateur voit apparaître quelque chose à lire) il est occupé un temps, ce qui permet d'afficher le reste.

Exemple : remplir une listBox avec un grand nombre d'éléments long à préparer: en afficher 5 rapidement puis calculer et afficher les autres. L'utilisateur à l'impression que la ListBox se remplit immédiatement.

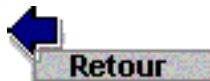
Site

Pour faire patienter l'utilisateur lors d'une routine qui dure longtemps ? (et lui montrer que l'application n'est pas bloquée):

- Transformer le curseur en sablier en début de routine, remettre un curseur normal en fin de routine.
- Utiliser une ProgressBar (pour les chargements long par exemple)

Ce qui n'influence pas la rapidité du code:

Les boucles For , Do , While ont toutes une vitesse identique.





7.4	Chronométrer le code.	Suivant 	 Sommaire
------------	----------------------------------	--	---

Je veux comparer 2 routines et savoir laquelle est la plus rapide.

Pour chronométrer une évènement long.

Entendons par évènement long, plusieurs secondes ou minutes.

Pas de problème, 2 solutions:

- On utilise un **Timer**, (dans l'évènement Ticks qui survient toutes les secondes, une variable s'incrémente comptant les secondes). ([4-5](#)).

- On peut utiliser l'**heure Système**.

```
Dim Debut, Fin As DateTime
Dim Durée As TimeSpan
Debut=Now
...Routine...
Fin=Now
Durée=Fin-Debut
```

Créer un compteur pour les temps très court.

C'est le cas pour chronométrer des routines durée bien inférieure à une seconde.

Cela semblait à première vue facile!!!

J'ai en premier lieu utilisé un **Timer**, (dans l'évènement

Ticks un compteur de temps s'incrémente) mais les intervalles de déclenchement semblent long et aléatoire

J'ai ensuite utilisé l'heure système:

Mais 'Durée' est toujours égal au 0 pour les routines rapides car il semble que Now ne retourne pas les millisecondes ou les Ticks.

J'ai trouvé la solution chez Microsoft:

Utilisation d'une routine de Kernel32 qui retourne la valeur d'un compteur (QueryPerformanceCounter).

QueryPerformanceFrequency retourne le nombre de fois que le compteur tourne par seconde.

Exemple: comparer 2 boucles, l'une contenant une affectation de variable tableau (b=a(2)) l'autre une affectation de variable simple (b=c), on gagne 33%.

```
Declare Function QueryPerformanceCounter Lib "Kernel32"  
(ByRef X As Long) As Short  
Declare Function QueryPerformanceFrequency Lib "Kernel32"  
(ByRef X As Long) As Short
```

```
Private Sub ButtonGo_Click(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles ButtonGo.Click  
Dim debut As Long  
Dim fin As Long  
Dim i As Long  
Dim a(5) As String  
Dim b As String  
Dim c As String  
Dim d1 As Long  
Dim d2 As Long  
'*****première routine  
QueryPerformanceCounter(debut)  
For i = 0 To 10000  
b = a(2)  
Next  
QueryPerformanceCounter(fin)  
d1 = fin - debut  
Label1.Text = d1.ToString  
'*****seconde routine  
QueryPerformanceCounter(debut)  
c = a(2)  
For i = 0 To 10000  
b = c  
Next  
QueryPerformanceCounter(fin)  
d2 = fin - debut  
Label2.Text = d2.ToString  
  
Label5.Text = "Gain 2eme routine:" & 100 - Int(d2 / d1 *  
100).ToString  
End Sub
```

C'est cette routine qui est utilisée pour étudier l'[optimisation du code](#).

Elle n'est pas parfaite, car sujette à variation: les valeurs sont différentes d'un essai à l'autre en fonction des processus en cours !

Y a-t-il mieux?



Site LDF Cours VB.net



8.1	Allons plus loin avec les procédures.	Suivant	Sommaire
-----	---------------------------------------	---------	----------

On savait que les procédures pouvaient être Public ou Privée.

En fait une procédure peut être:

Public

Les procédures déclarées avec le mot clé Public ont un accès public. Il n'existe aucune restriction quant à l'accessibilité des procédures publiques.

Protected

Dans un module de classe:

Les procédures déclarées avec le mot clé Protected ont un accès protégé. Elles sont accessibles seulement à partir de leur propre classe ou d'une classe dérivée.

Friend

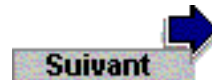
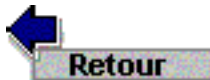
Les procédures déclarées avec le mot clé Friend ont un accès ami. Elles sont accessibles à partir du programme contenant leur déclaration et à partir de n'importe quel autre endroit du même assembly.

Protected Friend

Les procédures déclarées avec les mots clés Protected Friend ont l'union des accès ami et protégé. Elles peuvent être utilisées par du code dans le même assembly, de même que dans les classes dérivées. L'accès Protected Friend peut être spécifié uniquement pour les membres des classes.

Private

Les procédures déclarées avec le mot clé Private ont un accès privé. Elles ne sont accessibles qu'à partir de leur contexte de déclaration, y compris à partir des membres de types imbriqués, tels que des procédures.



Site LDF Cours VB.net



8.2	Comprendre le code créé par VB.	Suisvant	Sommaire
-----	--	----------	----------

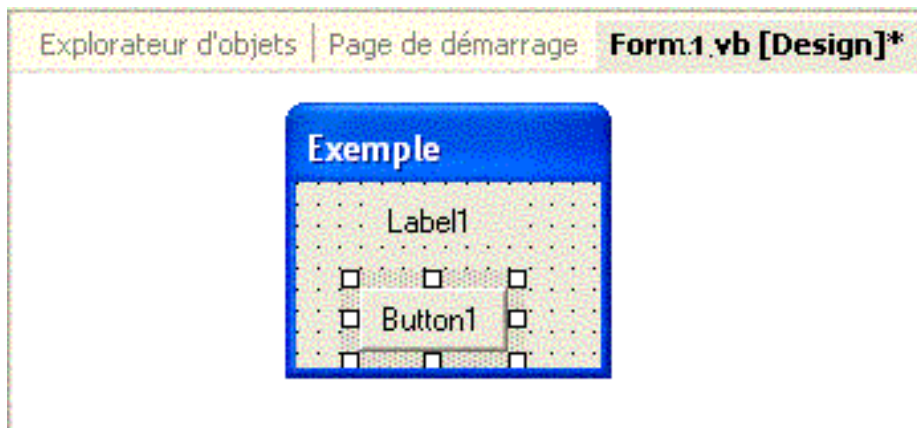
Comprendre le code généré automatiquement par Vb quand on crée une formulaire ou un contrôle.

Code généré automatiquement lors de la création d'un formulaire ou d'un contrôle.

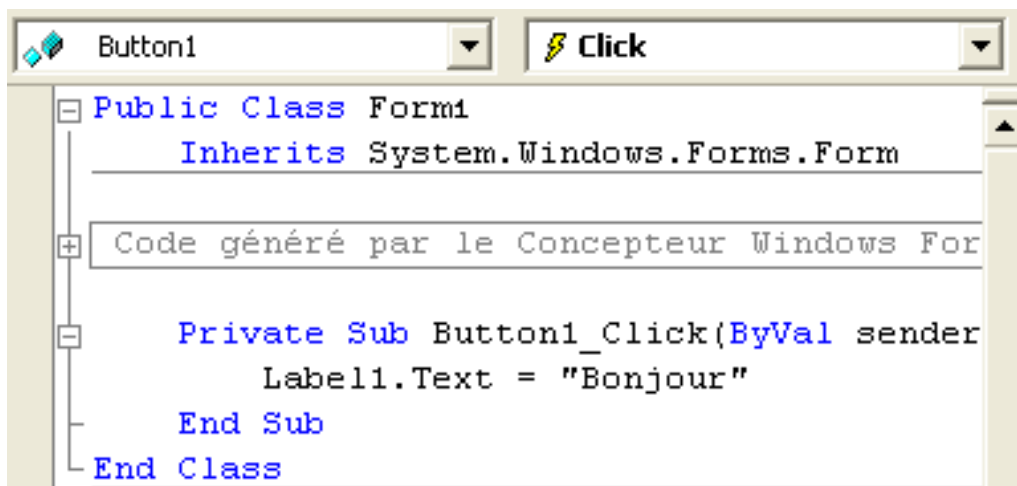
Une application 'Windows Forms' est principalement constitué de **formulaires** (ou fenêtre), de **contrôles** et de leurs **événements**.

Effectivement, pendant la création de l'interface utilisateur de votre application, vous créez généralement une fenêtre contenant des contrôles et des procédures évènements.

Quand vous créer un nouveau projet 'Windows Forms' cela dessine un formulaire, une fenêtre vide et le code correspondant, Ajoutons y un bouton cela donne l'interface utilisateur suivante:



Comme on l'a vu, VB crée le code correspondant et dans ce code une Classe correspondant à la fenêtre; cette classe dérive de la Classe **Form**.



(On rappelle que la véritable fenêtre, l'objet sera instancié à partir de cette classe)

Décortiquons le code:

Vb crée une Class nommé Form1, elle est public (accessible partout)

`Public Class Form1`

Cette Classe hérite des propriétés de la Classe Form (celle ci est fournis par le Frameworks)

`Inherits System.Windows.Forms.Form`

Ensuite il y a une région (partie du code que l'on peut

'contracter' et ne pas voir ou 'dérouler'; Cette région contient : " **Le Code généré (automatiquement) par le Concepteur Windows Form** ", si on le déroule en cliquant sur le '+', on voit:

-Le **constructeur de la fenêtre**: la routine `Sub New`

```
MyBase fait référence à la classe de
base de l'instance en cours d'une
classe,
MyBase.New 'construit' la Classe
```

-Le **destructeur de la fenêtre**: la routine `Sub Dispose`

-Le **créateur des contrôles** de la fenêtre par la procédure `Sub InitializeComponent`

Elle est nécessaire pour créer les contrôles et définir les propriétés de ces contrôles:

Exemple : création d'un label

```
Me.Label1=
NewSystem.Windows.forms.Label
```

Modification d'une propriété: `Me.Label.Text="Hello"`

Elle définit aussi les propriétés du formulaire:

```
Me.Name = "Form1"
```

Exemple d'un formulaire vide nommé Form1

```
Public Class Form1
Inherits System.Windows.Forms.Form
#Region " Code généré par le Concepteur Windows Form "
Public Sub New()
MyBase.New()
'Cet appel est requis par le Concepteur Windows Form.
InitializeComponent()
'Ajoutez une initialisation quelconque après l'appel
InitializeComponent()
End Sub
'La méthode substituée Dispose du formulaire pour nettoyer
la liste des composants.
Protected Overloads Overrides Sub Dispose(ByVal disposing
As Boolean)
If disposing Then
If Not (components Is Nothing) Then
```



```

components.Dispose()
End If
End If
MyBase.Dispose(disposing)
End Sub
'Requis par le Concepteur Windows Form
Private components As System.ComponentModel.IContainer
'REMARQUE : la procédure suivante est requise par le
Concepteur Windows Form
'Elle peut être modifiée en utilisant le Concepteur
Windows Form.
'Ne la modifiez pas en utilisant l'éditeur de code.
<System.Diagnostics.DebuggerStepThrough()> Private Sub
InitializeComponent()
'
'Form1
'
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
Me.ClientSize = New System.Drawing.Size(292, 266)
Me.Name = "Form1"
Me.Text = "Form1"
End Sub
#End Region

End Class

```

Si dans la fenêtre Design **on ajoute un bouton Button1** cela ajoute le code:

Cette ligne contenant `WithEvents` indique qu'il y a une gestion d'évènement sur les boutons.
Friend WithEvents Button1 As System.Windows.Forms.Button

Cette ligne crée le bouton.
Me.Button1 = New System.Windows.Forms.Button

Cette ligne le positionne:
Me.Button1.Location = New System.Drawing.Point(56, 144)
 Cette ligne lui donne un nom.
Me.Button1.Name = "Button1"
 Cette ligne détermine sa taille
Me.Button1.Size = New System.Drawing.Size(104, 24)
 Cette ligne indique ce qui est affiché sur le bouton
Me.Button1.Text = "Button1"

Cela donne:
Private components As System.ComponentModel.IContainer
Friend WithEvents Button1 As System.Windows.Forms.Button
 <System.Diagnostics.DebuggerStepThrough()> **Private Sub**
InitializeComponent()
Me.Button1 = New System.Windows.Forms.Button

```
Me.SuspendLayout()  
,  
'Button1  
,  
  
Me.Button1.Location = New System.Drawing.Point(56, 144)  
Me.Button1.Name = "Button1"  
Me.Button1.Size = New System.Drawing.Size(104, 24)  
Me.Button1.TabIndex = 0  
Me.Button1.Text = "Button1"
```

Les procédures évènements correspondant au bouton sont automatiquement créer:

```
Private Sub Form1_Load(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles MyBase.Load  
  
End Sub
```

On constate qu'il y a une liaison entre la fenêtre Design et le code généré; on pourrait modifier dans le code l'interface utilisateur. **C'est déconseillé d'aller trafiquer dans cette zone de "Code généré par le Concepteur Windows Form"**, il faut mieux faire des modifications dans la partie design et dans la fenêtre de propriété.

Substitution de procédures évènement:

Il est possible de substituer une méthode (utiliser sa propre méthode à la place de la méthode normale qui existe normalement dans un contrôle)

Exemple créer un contrôle simple affichant toujours 'Bonjour'

Il faut créer une classe héritant des 'Control', détourner sont évènement OnPaint (avec **Overrides**)qui survient quand le contrôle se dessine pour simplement afficher 'Bonjour'

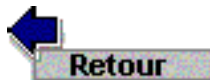
```
Public Class ControleAffichantBonjour  
Inherits Control  
Overrides Protected Sub OnPaint ( e As PaintEventArgs )  
    e.Graphics.DrawString ("Bonjour", Font, nex  
SolidBrush(ForeColor)  
End Sub  
End Class
```

Cet exemple ne sert strictement à rien!! Pour une fois!!

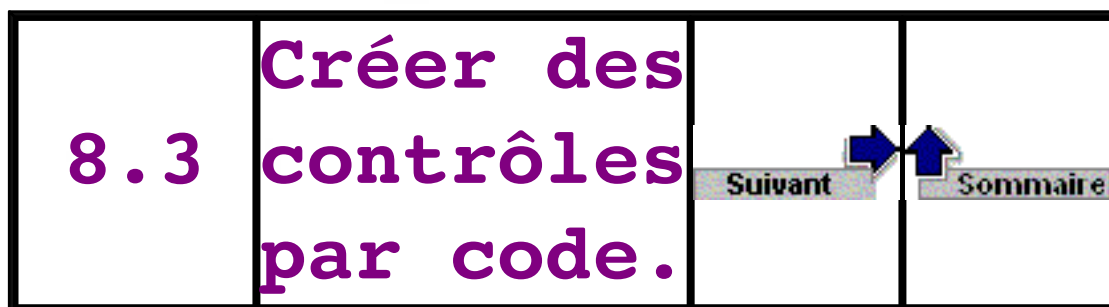
Il est aussi possible de détourner des évènements.

Dans le chapitre '[Impression](#)' il y a un bel exemple de création de "lien" entre un objet printdocument et la routine évènement PrintPage (imprimer hello avec un printdocument)

Dans le chapitre suivant on va utiliser ces connaissances pour, dans le code, créer soi-même des contrôles et leurs évènements.



Site **LDF** Cours **VB.net**



Dans le code, on peut créer soi-même de toutes pièces, des contrôles et leurs évènements.

Créer par code des contrôles.

Dans le code d'une procédure, il est possible de créer de toute pièce un contrôle, mais attention, il faut **tout faire!!**

Créons le bouton.

```
Dim Button1 = New Button
```

Modifions ses propriétés

```
Me.Button1.Location = New System.Drawing.Point(56, 144)
```

```
Me.Button1.Name = "Button1"
```

```
Me.Button1.Size = New System.Drawing.Size(104, 24)
```

```
Me.Button1.TabIndex = 0
```

```
Me.Button1.Text = "Button1"
```

Le bouton existe mais il faut l'ajouter à la collection **Controls** de la fenêtre (Cette collection contient tous les contrôles contenus dans la fenêtre):

```
Me.Controls.Add(Button1)
```

Le bouton existe mais pour le moment, **il ne gère pas les**

évènements.

Il faut inscrire le bouton dans une méthode de gestion d'évènements. En d'autres termes, Vb doit savoir quelle procédure événement doit être déclenchées quand un évènement survient. Pour cela, il y a 2 méthodes:

- Déclarer la variable avec le mot clé **WithEvents** ce qui permet ensuite d'utiliser le **Handles** du contrôle dans la déclaration d'une Sub:

Déclaration **dans la partie déclaration du module** (en haut)
(WithEvents n'est pas accepté dans une procédure):

```
Private WithEvents Button1 As Button1
```

Remarque Button1 est accessible dans la totalité du module .

Puis écrire la sub événement.

```
Sub OnClique ( sender As Object,
EvArg As EventArgs) Handles
Button1.Click
```

```
End Sub
```

Ainsi VB sait que pour l'évènement Button1.Click , il faut déclencher la Sub OnClique.

Remarque: il pourrait y avoir plusieurs Handles sur une même sub, donc des évènements différents sur des objets différents déclenchant la même procédure.

- Utiliser **AddHandler**

Déclaration (possible dans une procédure):

```
Dim Button1 As Button
```

Puis écrire la gestion de l'évènement.(L'évènement

```
Button1.click doit déclencher la  
procédure dont l'adresse est  
BouttonClique)
```

```
AddHandler Button1.Click AddressOf  
BouttonClique
```

Enfin on écrit la sub qui 'récupère
' l'évènement:

```
Private Sub BouttonClique (sender As  
Objet, evArgs As EventArgs)
```

```
End Sub
```

Ainsi VB sait que pour un évènement du Button1 , il faut déclencher la Sub BouttonClique

Exemple avec AddHandler:

Créons un TextBox nommé TB et une procédure déclenchée par KeyUp de ce TextBox:

Dans une procédure (Button1_Click par exemple): Je crée un TextBox nommé TB, je le positionne, je met dedans le texte 'ici une textbox'. Je l'ajoute aux Contrôles du formulaire.

Grâce à 'AddHandler', je lie l'évènement Keyup de cet objet **TB** à la sub que j'ai crée :TextboxKeyup.

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

```
Dim TB As New System.Windows.Forms.TextBox  
TB.Location = New System.Drawing.Point(2, 2)  
TB.Text = "ici une textBox"  
Me.Controls.Add(TB)  
AddHandler TB.Keyup, AddressOf TextboxKeyup.
```

```
End sub
```

```
Sub TextboxKeyup.(ByVal sender As Object, ByVal e As  
KeyEventArgs)
```

```
...  
End Sub
```

Si je crée un autre bouton TB2, j'ajoute de la même manière `AddHandler TB2.Click, AddressOf TextBoxKeyup2`, ainsi chaque événement de chaque contrôle à ses propres routines événement et en cliquant sur le bouton TB2 on déclenche bien `TextBoxKeyup2`

Attention, la procédure `TextBoxKeyup` doit recevoir impérativement les bons paramètres: un objet et un `KeyEventArgs` car ce sont les paramètres retournés par un `KeyUp`.

Autre exemple avec AddHandler mais avec 2 boutons:

Il est possible de créer **plusieurs contrôles ayant la même procédure événement:**

Exemple: Créons 2 boutons (BT1 et BT2) déclenchant une seule et même procédures (`BoutonClique`).

Dans ce cas, **comment savoir sur quel bouton l'utilisateur à cliqué ?**

En tête du module déclarons les boutons (Ils sont **public**):

```
Public BT1 As New System.Windows.Forms.Button
Public BT2 As New System.Windows.Forms.Button
```

Indiquons dans `form_load` par exemple la routine événement commune (`BoutonClique`) grâce a `AddHandler`.

`Form_Load`

```
BT1.Location = New System.Drawing.Point(2, 2)
BT1.Text = "Bouton 1"
Me.Controls.Add(BT1)
BT2.Location = New System.Drawing.Point(100,
100)
BT2.Text = "Bouton 2"
Me.Controls.Add(BT2)
AddHandler BT1.Click, AddressOf BoutonClique
AddHandler BT2.Click, AddressOf BoutonClique
```

`End Sub`

Si c'est le bouton 1 qui a été cliqué, afficher "button1" dans une `TextBox`:

```
Sub BoutonClique(ByVal sender As Object, ByVal e As
```

EventArgs)

```
If sender Is BT1 Then
    TextBox1.Text = "button 1"
ElseIf sender Is BT2 Then
    TextBox1.Text = "button 2"
End If
```

End Sub

La ruse est que déterminer quel objet (quel bouton) à déclenché l'évènement, pour cela on utilise le premier paramètre, le sender;

`If sender Is BT1 Then` : Si le sender est le bouton1..

Les délégués :

Pour la petite histoire, nous créons un **délégué** à chaque fois que nous créons une procédure gestionnaire d'évènement avec le mot `Handles` ou avec `AddHandler`.

En C on utilise des **pointeurs de fonction**, adresse en mémoire indiquant où le logiciel doit sauter quand on appelle une fonction ou un évènement. En VB on parle de **délégué**.

