

Secteur Tertiaire Informatique Filière étude - développement

Développer des composants d'interface

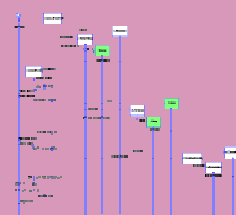
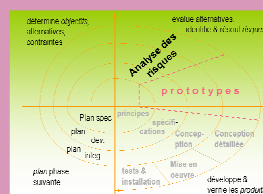
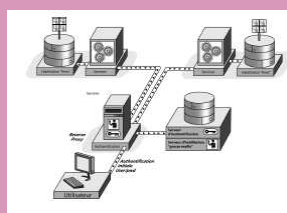
Créer des formulaires

Accueil

Apprentissage

Période en
entreprise

Evaluation



Code barre

SOMMAIRE

I	INTRODUCTION.....	5
I.1	Définitions.....	5
I.2	Les objets graphiques	6
I.2.1	Les propriétés.....	6
I.2.2	Les méthodes.....	8
I.2.3	Les évènements	8
II	CREATION D'UNE APPLICATION WINDOWS.....	11
II.1	Mode d'emploi	11
II.2	Caractéristiques du projet	16
II.3	La classe de démarrage	17
II.4	Le code modifiable d'un formulaire	18
II.4.1	Les espaces de nom.....	18
II.4.2	La classe Form.....	19
II.5	Le code généré par le concepteur	19
II.6	Mise en oeuvre : ajout d'un bouton	21
II.6.1	Ajout d'un bouton.....	21
II.6.2	Les mécanismes événementiels	22
II.6.3	Traiter le click du bouton.....	22
II.7	Exemple	24
III	L'IDE VISUAL STUDIO	26
III.1	Editer du code.....	26
III.2	Formater	27
III.3	Refactoriser	28
III.4	L'environnement graphique.....	28
IV	LES FENETRES	30
IV.1	Définition.....	30
IV.2	Les principales propriétés d'une fenêtre	30
IV.3	Les événements d'une fenêtre.....	32
IV.4	Les méthodes : Formulaires modaux et non modaux.....	33
IV.5	La classe MessageBox.....	33
V	GENERALITES SUR LES CONTROLES	35
V.1	Création de contrôles dynamiquement.....	35
V.2	Gestion des évènements.....	36
V.3	La validation des controles.....	37
V.3.1	Les évènements à utiliser	38
V.3.2	Les modes de contrôle	39
V.3.3	Affichage à l'utilisateur	43
V.3.4	Gérer le focus	45
VI	LES DIFFERENTS CONTROLES GRAPHIQUES	46

VI.1	Les zones d'affichage et d'édition.....	46
VI.1.1	Les zones de texte (TextBox).....	46
VI.1.2	Les MaskedTextBox.....	48
VI.1.3	Les zones d'affichage (Labels).....	50
VI.1.4	Les LinkLabels.....	51
VI.1.5	Les UpDown.....	51
VI.2	Boutons et cases.....	53
VI.2.1	Les boutons de commande (Button).....	53
VI.2.2	Les cases à cocher (CheckBox).....	53
VI.2.3	Les Boutons Radio (RadioButton).....	54
VI.3	Les boîtes de liste.....	56
VI.3.1	Les boîtes de liste standard (ListBox).....	56
VI.3.2	Les boîtes de liste avec cases à cocher.....	58
VI.3.3	Les boîtes de liste déroulantes (comboBox).....	59
VI.4	Les composants de défilement.....	61
VI.4.1	Les barres de défilement (xxScrollBar).....	61
VI.4.2	Les barres graduées (TrackBar).....	63
VI.4.3	Les barres de progression (ProgressBar).....	64
VI.5	Les Conteneurs.....	64
VI.5.1	Contrôles GroupBox et Panel.....	65
VI.5.2	Contrôles FlowLayoutPanel et TableLayoutPanel.....	65
VI.5.3	Contrôle SplitContainer.....	66
VI.5.4	Contrôle TabControl.....	66
VII	LES MENUS, BARRES D'OUTILS ET D'ETAT	67
VII.1	Le menu de l'application (MenuStrip).....	67
VII.2	Les menus contextuels (ContextMenuStrip).....	70
VII.3	La barre d'outils (ToolStrip).....	71
VII.4	La barre d'outils (StatusStrip).....	72
VII.5	Le ToolStripContainer.....	73
VIII	PLUSIEURS FEUILLES DANS L'APPLICATION.....	74
VIII.1	Définitions.....	74
VIII.2	Gestion.....	74
VIII.3	Les feuilles MDI.....	75
VIII.4	Communiquer entre feuilles.....	77
	

I INTRODUCTION

I.1 DEFINITIONS

On peut distinguer deux types de traitements :

- Le traitement par lot ou traitement différé ou batch (l'utilisateur lance le traitement et peut s'en aller).
- Le traitement conversationnel (sur grands systèmes on dit transactionnel) qui consiste en une suite d'échanges entre le programme et l'opérateur, à travers différents écrans (sous WINDOWS on dit fenêtre ou feuille).

WINDOWS est plutôt adapté à ce deuxième type de traitement et C# approprié à développer de telles applications.

Mieux, on peut dire que la conception en C# colle aux caractéristiques et modèles propres à WINDOWS.

Toute application WINDOWS est réductible à un ensemble de feuilles, au départ blanches, vides et sans limites.

Chaque feuille va ensuite être spécialisée et devenir une feuille de saisie, un document, une boîte de dialogue etc. *comportant un certain nombre d'objets* : zones de données, propositions de choix d'options ou d'actions.

Ces objets s'appellent *contrôles* (zones de texte, boutons, cases à cocher, options, listes déroulantes, barres d'outils etc.).

Les *actions de l'opérateur* (déplacement du curseur, de la souris, saisie ou modification ...) sur chacun de ces contrôles font partie du dialogue, correspondent à un *événement* et nécessitent le plus souvent un traitement instantané.

On parle de programmation *événementielle*.

A chaque événement correspond une séquence isolée de programme, un sous-programme, effectuant un traitement en fonction du contexte.

Un traitement réalisé sous C# est donc morcelé en une multitude de sous-programmes.

Il y en a autant que de couples événement/contrôle, plus peut-être quelques uns pour des séquences d'intérêt général.

I.2 LES OBJETS GRAPHIQUES

On distingue :

Les feuilles (Form) ou fenêtres :

- Feuille principale
- Feuilles filles (une par document)
- Feuilles ou boîtes de dialogue.

Les contrôles :

- Etiquettes ou labels, pour dénommer des données
- Zones de texte, utiles pour contenir des données
- Boutons de commande
- Boutons d'option dits boutons radio
- Cases à cocher
- Conteneurs pour contenir d'autres contrôles
- Listes simples, déroulantes ou combinées (combobox)
- Barres de défilement

Tous les contrôles de l'espace de noms **System.Windows.Forms** héritent de la classe **System.Windows.Forms.Control** ; c'est la raison pour laquelle ils possèdent un grand nombre de caractéristiques communes.

I.2.1 Les propriétés

Chaque objet possède des **propriétés** qui définissent son identification, sa position, ses couleurs, son état, sa valeur, les possibilités de le modifier, ses liens avec d'autres objets, etc.

Quelques propriétés sont semblables pour la majorité des contrôles (par exemple : propriétés de forme, positionnements, couleurs, polices de caractères). Celles-ci sont le plus souvent fixées dès la conception et fréquemment de manière implicite.

On n'intervient sur elles qu'exceptionnellement.

Nous n'en parlerons pas, sauf pour en lister quelques unes :

Couleurs	Police	Position	Tabulation
BackColor	Font.Bold	Height	TabIndex
ForeColor	Font.FontFamily	Width	TabStop
	Font.Height	Location.X	
	Font.Italic	Location.Y	
	Font.Name		
	Font.Size		
	Font.Underline		

D'autres propriétés sont le plus souvent utilisées. Nous allons en faire quelques commentaires.

Name : C'est le nom de la variable ou structure associée à l'objet. C'est avec ce nom que vous désignerez un contrôle pendant le traitement.

Text: C'est le nom du contrôle que l'on voit sur la feuille :

- Libellé de la feuille dans le bandeau
- Libellé d'un label (ou étiquette)
- Libellé du Bouton radio ou de la case à cocher
- Libellé de l'objet data qui représente une base de donnée

Souvent on le met en constante une fois pour toutes. Il arrive cependant que l'on fasse des contrôles dynamiques et que l'on soit amené, pendant le déroulement de l'application, à changer le titre d'un contrôle. Pensez-y, c'est parfois commode et intelligent.

Visible : Certains contrôles peuvent très bien ne pas apparaître sur la feuille. Par exemple, un contrôle Timer qui n'a aucun intérêt visuel, ou un contrôle qui n'a momentanément pas de sens dans le contexte.

Cette propriété vaut True ou False.

Enabled : Un contrôle, tout en étant visible peut ne pas être utilisable. Par exemple un bouton "Ajout" ou "Suppression" alors qu'il n'y a aucun élément à ajouter ou supprimer. Dans ce cas, le contrôle est inaccessible par l'opérateur et apparaît en grisé.

Cette propriété vaut True ou False.

Parent : Ce contrôle n'est accessible qu'en exécution et en lecture seulement. Il indique le nom du conteneur du contrôle courant

TabIndex : Définit l'ordre de passage d'un contrôle à un autre (à condition que sa propriété TabStop soit à true) avec les touches de tabulation.

TabStop : Indique si le bouton peut recevoir le focus.

I.2.2 Les méthodes

Un objet possède également des **méthodes**, séquences de traitement pré codées permettant de réaliser des actions

Certaines méthodes sont communes à tous les contrôles, comme la méthode **Focus** qui permettra de positionner le focus sur un contrôle particulier ou particulières à une classe de contrôle, comme la méthode **Clear** de la TextBox qui efface tout le texte du contrôle zone de texte.

I.2.3 Les événements

Un objet réagit à certains **événements** en exécutant une procédure (si on l'a programmée) dont le nom par défaut comporte le nom de l'objet et celui de l'événement.

Comme avec les propriétés, on retrouve souvent les mêmes possibilités d'événements pour les contrôles. :

C'est la base de la programmation événementielle.

Commentons les événements les plus usités pour une zone de texte :

GotFocus :

Se produit lorsque le contrôle prend le *focus*.

Enter :

Se produit lorsque le contrôle devient contrôle actif du formulaire.

Pour effectuer un traitement préalable à une saisie ou modification : dès que l'on met un pied dans le contrôle (Avec la souris en cliquant, par tabulation, avec les flèches, ...)

TextChanged :

Se produit lorsque la valeur de la propriété Text a été modifiée.

Un seul caractère suffit. Permet de déclencher des contrôles très réactifs.

Beaucoup de traitements gagneront plutôt à être faits aux événements Leave (On quitte le champ).

KeyDown, KeyPress, KeyUp

Ces événements sont utiles à traiter particulièrement lorsque l'on veut gérer finement la saisie au clavier caractère par caractère.

KeyDown et KeyUp, utilisés pour toute touche, servent plus particulièrement à traiter les touches spéciales, touches de fonction ou combinaison avec *Ctrl*, *Maj* (ou Shift) et *Alt*.

Ces événements sont exploitables à tout moment dans une feuille pour filtrer les raccourcis claviers (ou shortcuts), actions de l'opérateur alternatives à l'appui sur un bouton.

MouseDown, MouseUp, MouseMove, MouseEnter, MouseHover , MouseLeave:

On peut contrôler finement ce que fait la souris :

MouseDown : Un bouton de la souris est enfoncé ;

MouseUp : Un bouton de la souris est relâché ;

MouseMove : La souris se déplace au dessus du contrôle

MouseEnter : La souris entre dans la zone du contrôle

MouseLeave : La souris quitte la zone du contrôle

MouseHover : La souris marque un temps d'arrêt dans le contrôle

Leave:

Se produit lorsque le contrôle cesse d'être le contrôle actif du formulaire.

Déclenche les traitements lorsque l'on quitte un contrôle : après une saisie ou modification.

Validating, Validated

Validating se produit lorsque l'utilisateur veut quitter le contrôle, sous réserve que le contrôle ou il souhaite se diriger ait la valeur de sa propriété CauseValidation à true.

Validated se produit lorsque Validating s'est terminée avec succès.

(Utilisés pour contrôler la validité d'un champ)

Click:

Utilisé pour les Boutons de commande et les boutons d'option et CheckBox

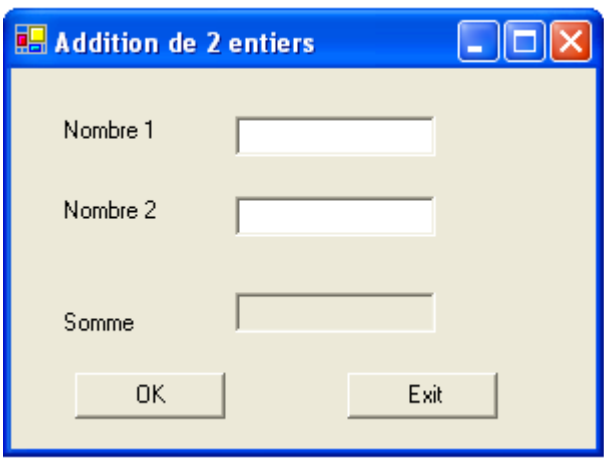
DoubleClick :

Utile pour les sélections dans des listes qui, lorsqu'elles sont "double-cliquées", déclenchent un traitement ou un affichage.

DragDrop, DragEnter, DragLeave, DragOver

Tous ces événements sont relatifs à la technique de cliquer-glisser (Drag and Drop)

Un événement peut en cacher un autre (vieux dicton d'un ex-service public!)

	<p>Le curseur étant dans la zone de saisie "Nombre1", nous pouvons aller dans Nombre2 de plusieurs manières :</p> <p>Click dans Nombre2 Tabulation ou ↓</p> <p>Nous devons donc envisager de traiter tous les cas possibles de passage de Nombre1 à Nombre2, en étant sûr de <i>traiter une fois et une seule fois</i> l'action de l'opérateur.</p>
---	---

Du côté de Nombre1 nous devons envisager l'occurrence possible des événements :

Leave quand on quitte le champ,
ou KeyPress pour filtrer l'appui sur TAB

Pour Nombre2 :

Enter On arrive dans le champ (mais d'où vient-on?)
MouseEnter On y arrive avec la souris
Click et bien sûr on clique.

Suivant le traitement souhaité, celui-ci ne devant s'effectuer qu'une fois, il faudra choisir un événement approprié, unique et non ambigu.

II CREATION D'UNE APPLICATION WINDOWS

Visual Studio a été entièrement remanié de manière à standardiser les méthodes de développement à destination des deux environnements qui cohabitent de plus en plus étroitement aujourd'hui :

- Les applications Windows.
- Les applications Internet.

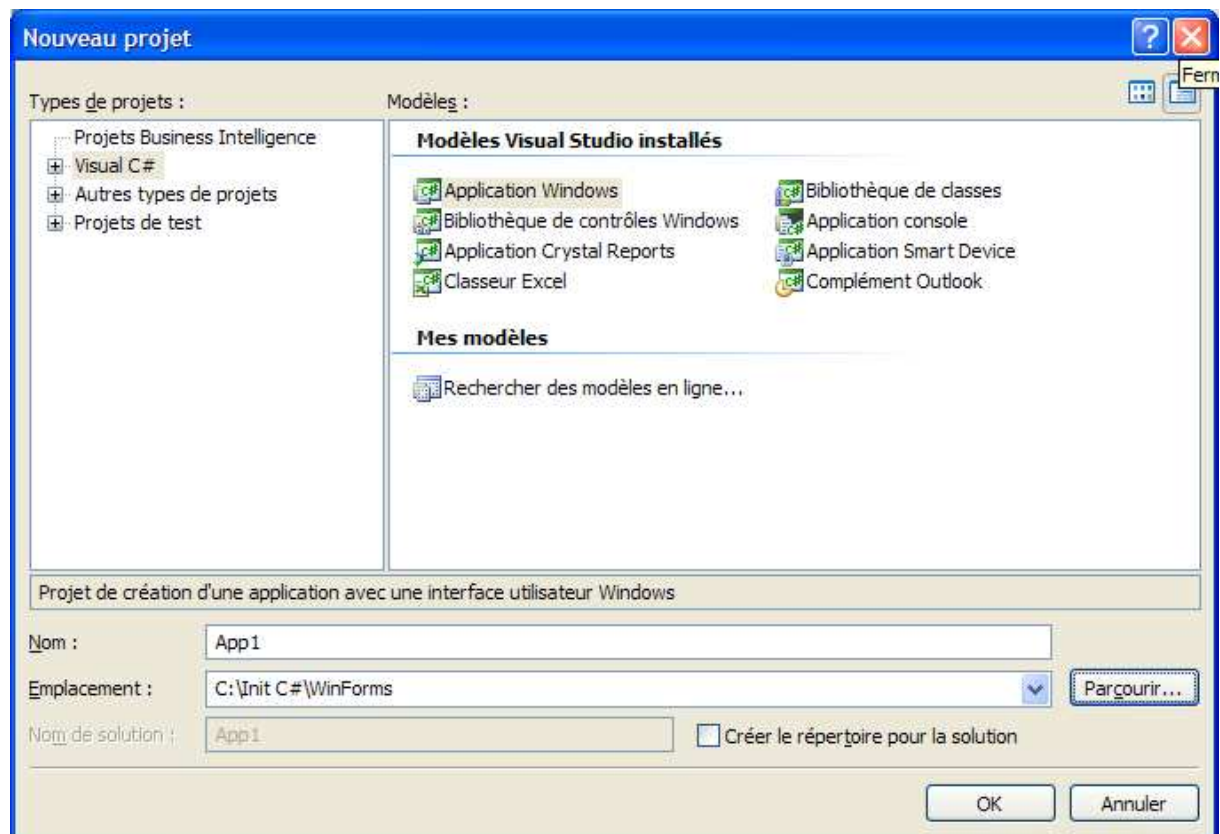
Dans les deux cas, une application se compose de fenêtres nommées **Forms**.

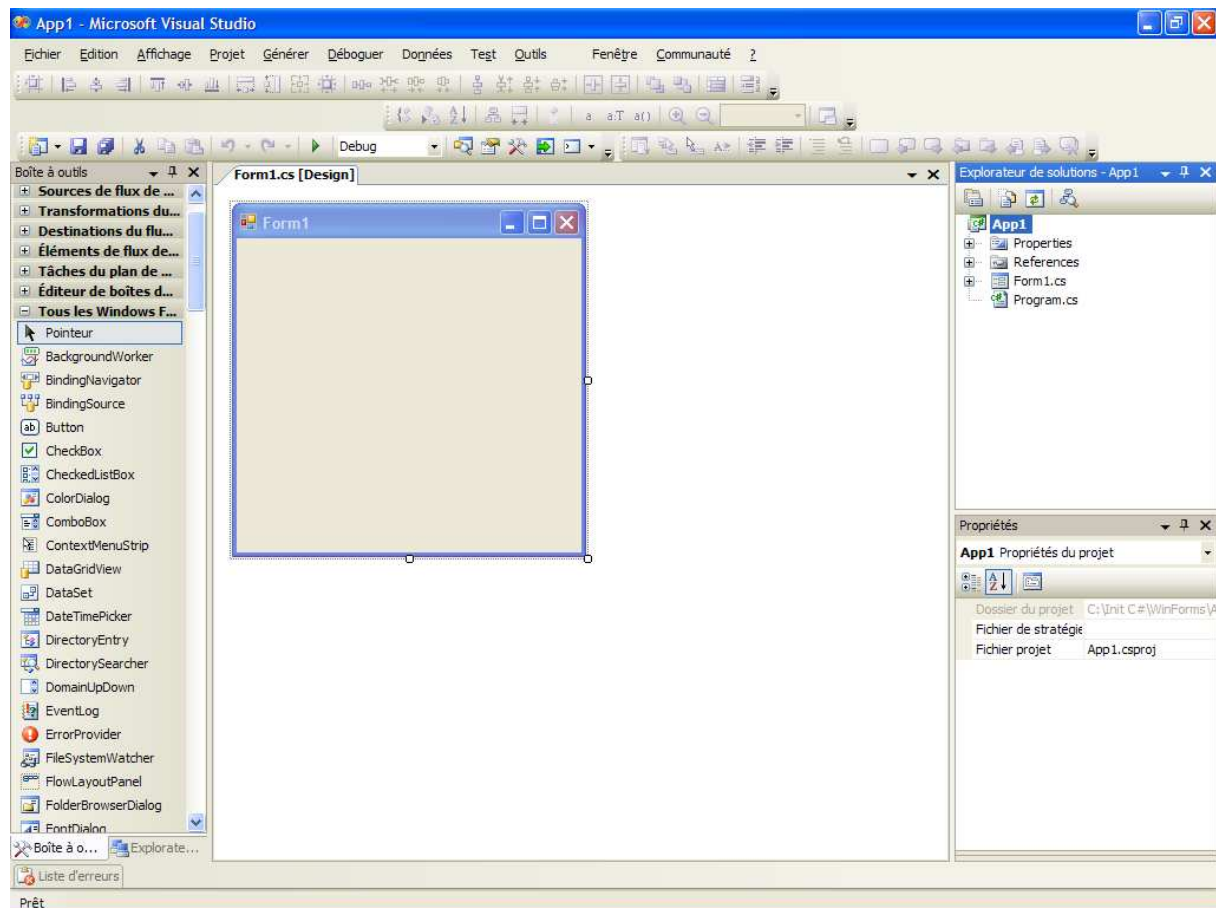
Pour les applications Windows, il s'agit de **WinForms** –que l'on nommera **feuilles** ou **formes** - tandis qu'il s'agit de **WebForms** -que l'on nommera **pages** - pour les applications dédiées au Web. Mais le développement est similaire dans les deux cas.

II.1 MODE D'EMPLOI

Un projet est créé sous l'environnement Visual Studio .NET qui se charge de créer les répertoires.

- a. créer un nouveau projet par le menu fichier, en choisissant le type "**Visual C# Windows**" et le modèle "**Applications Windows**", et le nommer (WindowsApplication1 par défaut), ici App1
Choisir comme emplacement un répertoire de travail.





- b. renommer le fichier Form1.cs en un nom mnémonique caractérisant l'application, dans cet exemple frmTest
Cette action sera à effectuer pour chaque feuille codée dans une application.

Propriétés

Actualiser

icône "Afficher tous les fichiers"

Afficher le code

Afficher le diagramme de classe

Avec Visual Studio 2003, un formulaire Windows était complètement défini sur un seul fichier source C# : le code généré par l'EDI* était séparé du code du développeur au moyen de régions.

Dans Visual Studio 2005, à un formulaire frmXX correspond 2 fichiers :

- frmXX.cs qui contient le code du développeur.
- frmXX.Designer.cs qui contient le code généré par l'EDI*.

* Environnement de Développement Intégré

c. modifier le fichier frmTest.cs

La propriété **Name** de la feuille, qui sert à adresser la feuille dans le code aura été modifiée à frmTest ; donner un titre à la feuille en modifiant la propriété **Text** dans la fenêtre des propriétés

Cette action sera à répéter systématiquement pour chaque feuille traitée dans une application.

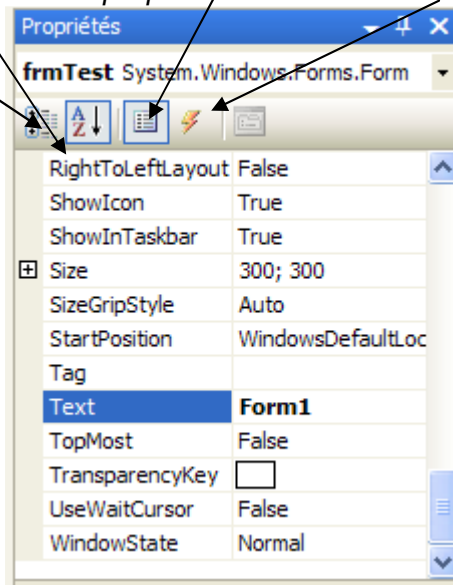
Cette fenêtre comporte la liste des propriétés de l'objet sélectionné (ou de la feuille si aucune sélection n'est faite sur un contrôle).

S'il n'est pas intéressant d'agir à ce stade sur les propriétés de taille ou position du contrôle, on y définit en revanche son nom, sa valeur et son état initial et, peut-être, quelques autres valeurs ...

Classement par thèmes ou

Classement alphabétique

.... des propriétés ou des événements



La fenêtre des propriétés n'est renseignée que si la feuille est en mode design.

d. Dans le répertoire C:\InitC#\WinForms, on constate qu'un répertoire physique *App1* et différents fichiers sont créés:

- les fichiers qui décrivent le projet App1 (.sln, .csproj). , une solution pouvant contenir plusieurs projets
- le fichier qui contiendra le code C# (frmTest.cs) modifiable par le développeur
- et celui contenant le code C# généré par le concepteur Visual Studio (frmTest.Designer.cs)
- le fichier XML (.resx) où sont stockées les ressources -données, images,...- locales au formulaire frmTest.

Ces ressources peuvent être modifiées selon des critères de "culture" qui particularisent par exemple la langue, sous Visual Studio sans entraîner de recompilation de la page

Toute nouvelle feuille créée par la suite donnera toujours lieu à ces deux mêmes types de fichiers (ouvrir par l'icône "Afficher tous les fichiers" de l'explorateur de

projet)

La fenêtre de code modifiable par le développeur peut être atteinte :





- en cliquant sur l'icône Afficher le code dans l'explorateur de solution
- en cliquant droit puis Afficher le code sur le fichier frmTest.cs dans l'explorateur de solution




La fenêtre de code générée par Visual Studio peut être atteinte par le menu contextuel du fichier Form1.Designer.cs puis **Afficher le code** dans l'explorateur de solution


On pourra choisir un élément de code particulier dans la liste déroulante de droite.

Les icônes affichées représentent chacun un élément différent, quelquefois précédé d'un icône de signalisation, indiquant leur accessibilité :

Quelques exemples ...

<i>Icônes</i>	<i>Description</i>
	Espace de Nom
	Classe
	Méthode ou fonction
	Propriétés

<i>Icônes</i>	<i>Description</i>
	Champ ou variable
	Protégé
	Privé
	Publique

- e. Sauvegarder et **surtout penser à générer le projet par le menu "générer"** - faute de quoi les fichiers **.cs** de codes sources des classes du projet ne sont pas compilés. Vérifier dans la fenêtre de sortie en bas qu'il n'y a pas d'erreurs de compilation.
- f. Tester l'application créée, soit en la lançant depuis l'EDI¹ (bouton  de la barre d'outils) soit depuis l'explorateur Windows par le fichier exécutable App1.exe qui a été généré sous le répertoire bin

Remarques:

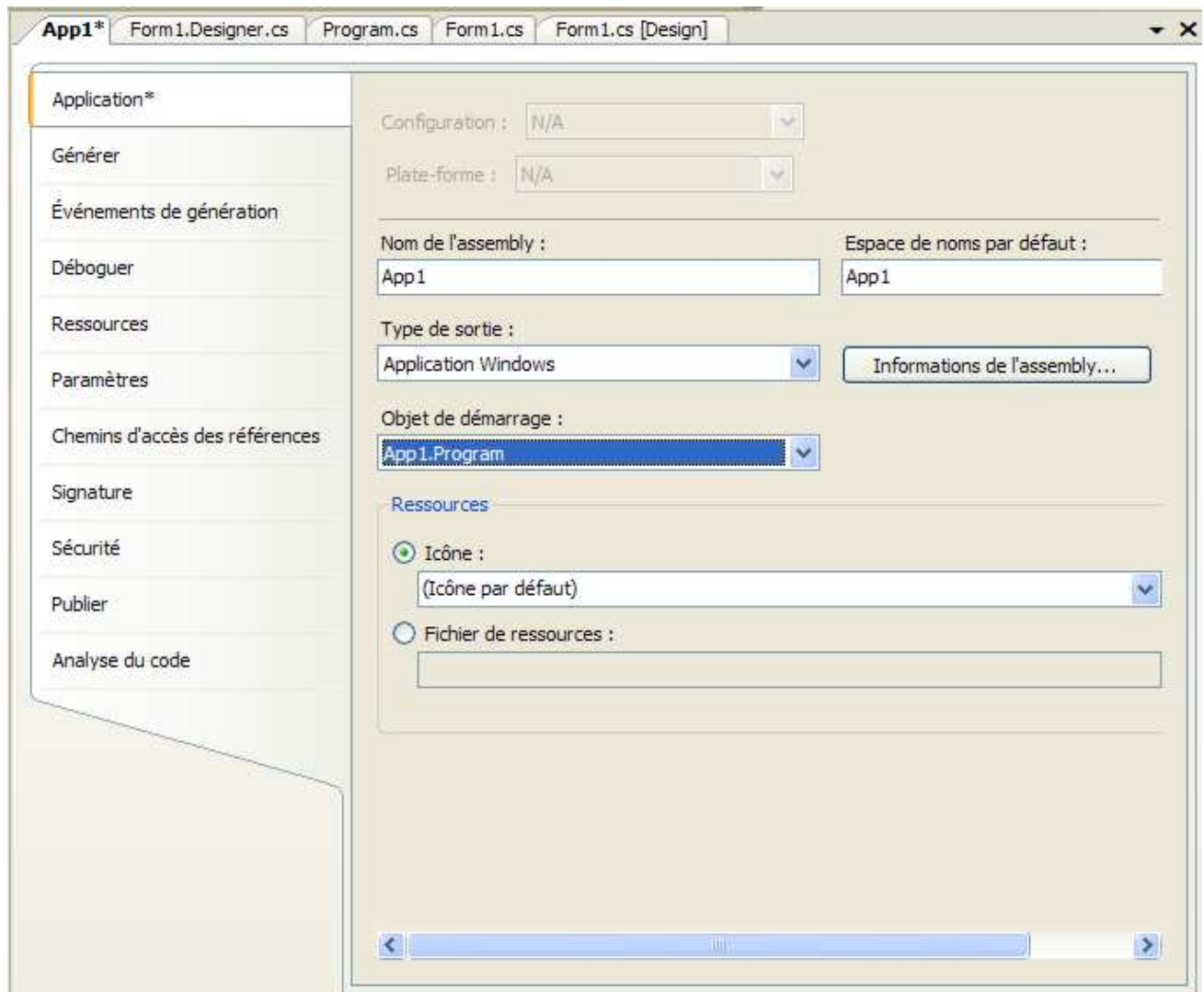
- la génération du projet (= la compilation de tous les sources) entraîne la sauvegarde des sources par défaut (paramétrable dans Outils / Options / Projets et Solutions / Générer et Exécuter)
- le lancement de l'application depuis l'EDI entraîne une génération préalable si cela le nécessite (fichiers sources modifiés récemment)
- la génération crée un fichier .exe rangé dans le répertoire bin/release s'il n'y a pas l'option de DEBUG ou dans le répertoire bin/debug dans le cas contraire (défaut)
- ces options DEBUG ou RELEASE du projet et le chemin de sortie de la génération se modifient depuis la fenêtre des propriétés du projet
- La version Debug génère des informations de débogage sous la forme de fichiers .pdb elle est utilisée pendant la phase de développement ; la version Release est destinée à être déployée : elle est entièrement optimisée et ne contient aucune information de débogage

¹ * Environnement de Développement Intégré

II.2 CARACTERISTIQUES DU PROJET

Les ressources globales au projet sont stockées dans un fichier Resources.resx présent dans le répertoire Properties du projet, modifiable dans les propriétés du projet.

La fenêtre des propriétés du projet App1 permet d'ouvrir sa *page des propriétés* depuis sa dernière icône à droite.



On remarque dans les *propriétés de l'application*

1. on peut choisir l'objet de démarrage de l'application (ici la classe App1.Program). Nous verrons plus tard que si l'on peut choisir entre différentes feuilles au démarrage, celles ci devront alors posséder une méthode Main() de point d'entrée dans leur classe.
2. on peut donner un autre nom au fichier exécutable résultant de la génération en modifiant la nom de l'assembly. Nous verrons ce qu'est une assembly.

II.3 LA CLASSE DE DEMARRAGE

La classe Program sert de point d'entrée à l'application (différence avec Visual Studio 2003)

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace App1
{
    static class Program
    {
        /// <summary>
        /// Point d'entrée principal de l'application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new frmTest());
        }
    }
}
```

La méthode Main appelle la méthode **Run** de la classe Application permet de lancer le traitement des messages Windows envoyés à l'application puis de créer le formulaire frmTest : le constructeur frmTest est alors exécuté avec la méthode InitializeComponent(), générée par l'éditeur (voir §3).

L'attribut **[STAThread]** (STA pour Single Threaded Apartment) signifie que seul le thread de la fenêtre pourra accéder aux composants de la fenêtre. Cet attribut est géré par la classe STAThreadAttribute qui dérive de la classe System.Attribute.

La méthode **EnableVisualStyles()** active les styles visuels pour l'application.

La méthode **SetCompatibleTextRenderingDefault()** active le rendu de texte par défaut pour les nouveaux contrôles.

II.4 LE CODE MODIFIABLE D'UN FORMULAIRE

Une application Windows IHM est composée d'un ensemble de fenêtres qui se présentent à l'écran, soit directement au démarrage, soit à la demande de l'utilisateur.

Chaque fenêtre -ou feuille ou forme- est fractionnée sur 2 fichiers sources (en C# - extension .cs pour CSharp- dans notre cas)..

Chaque fichier source contient une section de la définition de classe, et toutes les parties sont combinées lorsque l'application est compilée.

La définition de classe est fractionnée grâce au modificateur de mot clé **partial**.

Code modifiable par le développeur :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace App1
{
    public partial class frmTest : Form
    {
        public frmTest()
        {
            InitializeComponent();
        }
    }
}
```

II.4.1 Les espaces de nom

Notre application avec toutes ses feuilles (donc classes) forme un espace de noms appelé App1 (du nom du projet par défaut).

Un **espace de noms** ("name space") est un conteneur de types que l'on peut utiliser dans le programme: des classes, des interfaces, des structures, des énumérations. Ces espaces sont organisés hiérarchiquement et fonctionnellement.

On constate les directives dans le source de la feuille frmTest.cs qui font référence aux espaces de noms:

System - Contient des classes fondamentales et des classes de base qui définissent les types de données référence et valeur (string, String, int ...), les événements et gestionnaires d'événements, les interfaces, les attributs et le traitement des exceptions courantes. Il contient aussi de nombreux espaces de noms de deuxième niveau.

System.Collections.Generic - Contient des interfaces et des classes qui définissent différentes collections d'objets, telles que des listes, des files d'attente, des tableaux de bits, des tables de hachage et des dictionnaires.

System.ComponentModel - Fournit des classes qui sont utilisées pour implémenter le comportement au moment de l'exécution et au moment du design des composants et des contrôles.

System.Data - permet d'accéder aux classes qui représentent l'architecture ADO.NET

System.Drawing - Permet d'accéder aux fonctionnalités graphiques de base de GDI+.

System.Text - Contient des classes représentant les codages de caractères ASCII, Unicode, UTF-7 et UTF-8.

System.Windows.Forms - Contient des classes permettant de créer des applications Windows qui profitent pleinement des fonctionnalités élaborées de l'interface utilisateur disponibles dans le système d'exploitation Microsoft Windows.

Les **directives using** du source dispensent de qualifier l'utilisation d'un type dans le code source si ce type fait partie d'un de ces espaces de noms.

Par exemple la directive **using System.ComponentModel;** permettra d'utiliser sa classe *Container* en la nommant simplement **Container** et non pas "

System.ComponentModel.Container ()".

Cependant la notation entièrement qualifiée reste utile dans le cas où deux espaces de noms présenteraient des classes de même nom.

II.4.2 La classe Form

La classe **Form** est utilisée pour créer des fenêtres standard (SDI, Single Document Interface), boîtes de dialogues ou multidocuments (MDI, Multiple Document Interface).

Ici la classe est dérivée en une classe **frmTest** qui permettra de personnaliser notre fenêtre.

La méthode **frmTest()** est le **constructeur de classe** ; elle appelle la méthode **InitializeComponent()** contenant le code d'initialisation du formulaire, généré par Visual Studio et généré dans le fichier frmTest.Designer.cs.

Cette méthode privée- sera enrichie au fur et à mesure que l'on rajoutera des composants à la feuille.

Au départ elle ne comporte que les initialisations de la feuille elle même:

C'est à partir de cette classe frmTest que sera créé en mémoire un objet feuille lors du démarrage de l'application (ou par la suite pour d'autres feuilles de l'application, à la demande de l'utilisateur).

On utilisera pour ce faire la classe **Application** dont le rôle est de fournir les méthodes statiques permettant de démarrer et arrêter une application de type Windows, et des propriétés statiques permettant d'obtenir des informations telles que path, versions

Ainsi l'instruction

```
Application.Run(new frmTest());
```

permet ici de lancer le traitement des messages Windows envoyés à l'application puis d'afficher une feuille à partir d'une instance en mémoire de la classe frmTest.

II.5 LE CODE GENERE PAR LE CONCEPTEUR

Le code généré par le concepteur se trouve dans le fichier frmTest.Designer.cs.

```

namespace App1
{
    partial class frmTest
    {
        /// <summary>
        /// Variable nécessaire au concepteur.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Nettoyage des ressources utilisées.
        /// </summary>
        /// <param name="disposing">true si les ressources managées
        /// doivent être supprimées ; sinon, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Code généré par le Concepteur Windows Form

        /// <summary>
        /// Méthode requise pour la prise en charge du concepteur - ne
        /// modifiez pas
        /// le contenu de cette méthode avec l'éditeur de code.
        /// </summary>
        private void InitializeComponent()
        {
            this.SuspendLayout();
            //
            // frmTest
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(292, 266);
            this.Name = "frmTest";
            this.Text = "Premier test";
            this.ResumeLayout(false);

        }

        #endregion
    }
}

```

Un champ de type **Container** est destiné à contenir d'autres composants (contrôles graphiques, ...).

La méthode **Dispose** sera exécutée à la fermeture du formulaire et permet de libérer les ressources.

Dans la méthode **InitializeComponent** sont codées toutes les descriptions des objets graphiques générées par le concepteur Visual studio.

On retrouve le titre de la fenêtre donné précédemment dans la fenêtre des propriétés (propriété **Text**), et le nom de la feuille modifié par sa propriété **Name**

Les méthodes **SuspendLayout** et **ResumeLayout** sont utilisées en tandem pour supprimer les événements *Layout* multiples lorsque vous ajustez plusieurs attributs du contrôle (leur place, leur taille, leur couleur, leur fonte, leur contenu pour les contrôles conteneur, etc ...).

II.6 MISE EN OEUVRE : AJOUT D'UN BOUTON

II.6.1 Ajout d'un bouton

Le choix du composant graphique se fait dans la boîte à outil de Visual Studio, en glissant le bouton de l'onglet **Windows Forms** sur la feuille.

Premier réflexe : Modifier les propriétés Name (btnQuitter) et Text (Quitter) du contrôle sur la feuille .

On s'aperçoit qu'une donnée membre btnQuitter de la classe des Button est déclarée dans la classe frmTest.

```
private System.Windows.Forms.Button btnQuitter;
```

et qu'il est pris en compte par la méthode **InitializeComponent()** vue précédemment

```
this.btnQuitter = new System.Windows.Forms.Button(); ❶
// ...
//
// btnQuitter
//
this.btnQuitter.Location = new System.Drawing.Point(104, 111);
this.btnQuitter.Name = "btnQuitter";
this.btnQuitter.Size = new System.Drawing.Size(75, 23);
this.btnQuitter.TabIndex = 0;
this.btnQuitter.Text = "Quitter";
this.btnQuitter.UseVisualStyleBackColor = true;
```

❷

❶ Instanciation du bouton

❷ Définition de ses propriétés

II.6.2 Les mécanismes événementiels

Les événements peuvent être déclenchés par l'action d'un utilisateur (via son clavier ou sa souris), par le programme lui-même (instructions C#) ou par le système (timers par exemple).

Le programmeur peut décider de gérer ou non un événement en lui associant du code spécifique, appelé gestionnaire de l'événement.

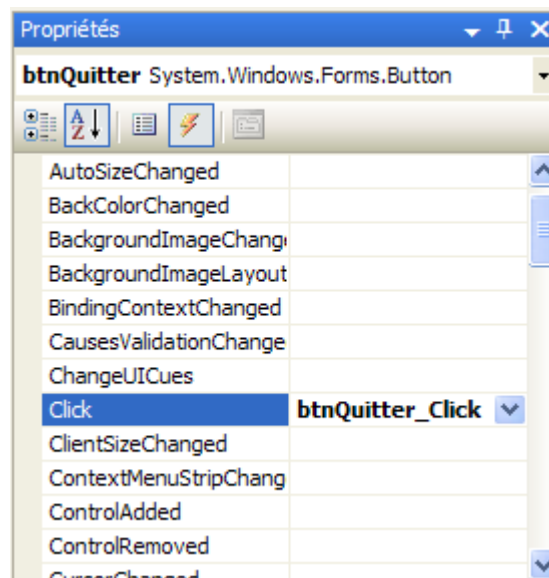
Il doit donc dans un premier temps écrire la méthode C# qui va traiter cet événement, puis dans un deuxième temps associer cette méthode à l'événement; l'événement peut d'ailleurs être associé à plusieurs méthodes qu'il activera toutes lorsqu'il claquera.

Il faut donc mettre en place un mécanisme qui va permettre de relier un objet "**abonné**" (notre page frmTest par exemple) à un objet "**annonceur**" (notre bouton btnQuitter par exemple) qui va déclencher un événement (clic par exemple). Lorsque l'événement va surgir chez l'annonceur, tous les abonnés seront prévenus et exécuteront le traitement qu'ils ont prévu pour cet événement.

Le .NET Framework utilise la technique du **délégué d'événement** qui connecte un événement avec son gestionnaire. On crée ce délégué de type **System.EventHandler** en lui donnant la référence à la méthode de traitement de l'abonné; ensuite le délégué est rajouté à l'événement de l'annonceur (se rappeler que les classes en C# présentent des membres de type événements).

II.6.3 Traiter le click du bouton

Le clic étant l'événement le plus courant du bouton, il suffit de double cliquer sur le bouton depuis la fenêtre de conception pour que le code soit généré **automatiquement** par Visual Studio, ou alors par le biais de la fenêtre de propriétés, double cliquer sur l'événement choisi.



La feuille (l'abonné) prépare sa méthode privée de traitement. Microsoft conseille de normaliser les noms de gestionnaires de la sorte: *annonceur_événement* et oblige à récupérer les deux paramètres *sender* (qui référence à l'objet annonceur qui génère l'événement) et *e* (qui est une structure d'infos de l'événement, et dont la classe est EventArgs ou une de ses classes dérivées pour certains événements spécifiques...)

- Génération de la méthode d'événement dans frmTest.cs

```
public partial class frmTest : Form
{
    // ....
    private void btnQuitter_Click(object sender, EventArgs e)
    {
    }
}
```

- Génération du code qui crée un **délégué** d'événement (*new*) et rajoute (+=) au membre événement **Click** du bouton (l'annonceur) (codé dans la méthode **InitializeComponent** dans **frmTest.Designer.cs**)

```
this.btnQuitter.Click += new System.EventHandler(this.btnQuitter_Click);
```

Il ne reste au développeur qu'à coder le traitement à exécuter lorsque l'utilisateur clique sur ce bouton Quitter.

Première possibilité: la forme se ferme elle même par **this.Close()**. Comme c'est la seule feuille de l'application, l'application est donc arrêtée.

Deuxième possibilité: par la méthode statique **Application.Exit()**; de la classe Application, qui arrête le traitement des messages windows et ferme toutes les fenêtres de l'application.

On remarquera qu'un arrêt par la fermeture (Close) de la feuille rend la main au Main() de démarrage, lui donnant la possibilité d'enchaîner une autre instruction en sortie de la fonction Run (et pourquoi pas un autre Run de l'Application?)

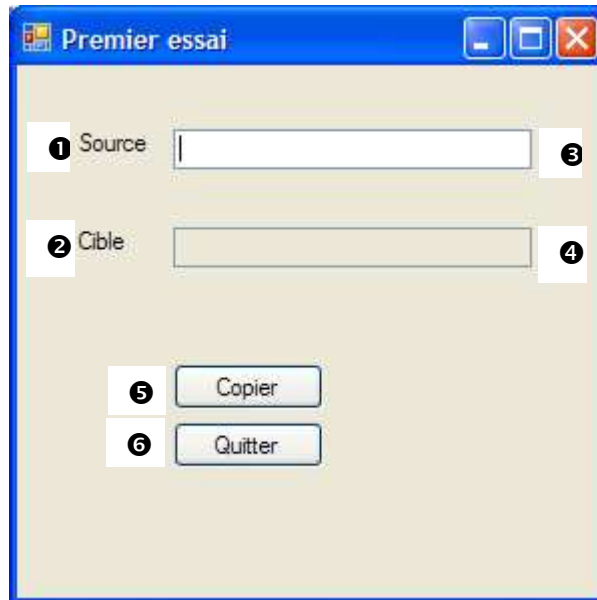
II.7 EXEMPLE

Après avoir saisi du texte dans le contrôle source, en cliquant sur le bouton Copier, le texte saisi est ajouté au contrôle cible inaccessible en saisie.

Le contrôle source est alors automatiquement effacé.

Le focus se positionne sur le contrôle source.

Le bouton Quitter permet de terminer l'application.



Les contrôles graphiques et leurs propriétés modifiées :

- ❶ un contrôle Label
Modifier sa propriété **Text** en positionnant la valeur **Source**
- ❷ un contrôle Label
Modifier sa propriété **Text** en positionnant la valeur **Cible**
- ❸ un contrôle TextBox
Modifier sa propriété **Name** en positionnant la valeur **txtSource**
- ❹ un contrôle TextBox
Modifier sa propriété **Name** en positionnant la valeur **txtCible**
Positionner sa propriété **ReadOnly** à **true**
- ❺ un contrôle Button
Modifier sa propriété **Name** en positionnant la valeur **btnCopier**
Modifier sa propriété **Text** en positionnant la valeur **Copier**
- ❻ un contrôle Button
Modifier sa propriété **Name** en positionnant la valeur **btnQuitter**
Modifier sa propriété **Text** en positionnant la valeur **Quitter**

Le code associé :

L'évènement **Click** sur le bouton **Copier** provoque l'exécution de la fonction :

```
private void btnCopier_Click(object sender, EventArgs e)
{
    // recopie de la valeur saisie de la source vers la cible
    txtCible.Text = txtCible.Text + " " + txtSource.Text;
    // efface la source : on utilise la méthode Clear
    txtSource.Clear();
    // positionne le focus: on utilise la méthode Focus
    txtSource.Focus();
}
```

L'évènement **Click** sur le bouton **Quitter** provoque l'exécution de la fonction :

```
private void btnQuitter_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```


III L'IDE VISUAL STUDIO

III.1 EDITER DU CODE

Commenter son code est indispensable : l'environnement de développement permet de commenter pour intégrer les commentaires dans l'Intellisense en interprétant les caractères `///` en C# (ou en Vb) comme étant des commentaires XML ; des balises XML spécifiques permettent alors de préciser des commentaires particuliers que Visual Studio utilisera :

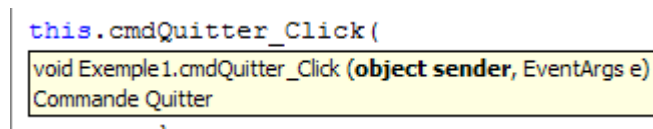
- `<summary>` : description de la classe
- `<param>` : description d'un paramètre
- `<returns>` : valeur de retour
- `<remarks>` : remarques particulières

Le travail est mâché : à l'insertion des premiers caractères `///` au dessus de la méthode à commenter, Visual Studio insère automatique un bloc complet de commentaires standards, alimentés avec les paramètres déjà existants de la méthode.

Le même résultat peut être obtenu en sélectionnant l'option Ajouter un commentaire dans le menu contextuel d'une méthode.

```
/// <summary>
/// Commande Quitter
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void cmdQuitter_Click(object sender, EventArgs e)
{
    ...
}
```

L'Intellisense affichera :



```
this.cmdQuitter_Click(
void Exemple1.cmdQuitter_Click (object sender, EventArgs e)
Commande Quitter
```

La boîte à outils offre la possibilité de stocker du code source afin de pouvoir le dupliquer autant de fois que désiré. Il suffit juste de glisser déposer du code sélectionné dans l'onglet Général de la boîte à outils.

L'opération inverse servira à insérer le bout de code dans l'éditeur.

Visual Studio suit les modifications apportées dans la fenêtre de code :

- barre verte à gauche: lignes modifiées et sauvegardées
- barre jaune : lignes modifiées non sauvegardées

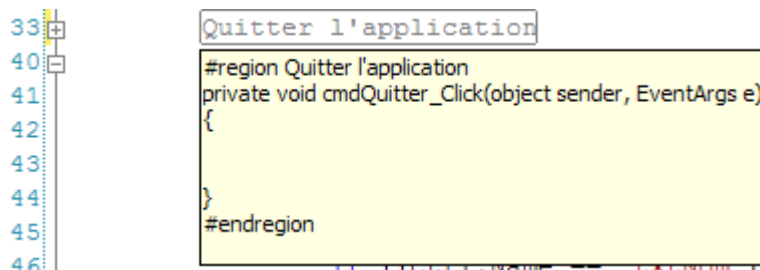
III.2 FORMATER

Des portions entières de code peuvent être regroupées en *régions* ; ce regroupement peut être très utile pour réduire le code affiché et donc faciliter la navigation.

Une *région* est une zone de code qui possède un début, une description et une fin, le début étant marqué par **#region**, et la fin par **#endregion**.

```
#region Quitter l'application
private void cmdQuitter_Click(object sender, EventArgs e)
{
...
}
#endregion
```

L'intérêt de créer des régions est de pouvoir masquer une région entière pour la réduire à une seule ligne contenant la description de la région, en cliquant sur le signe – apparaissant à gauche de la fenêtre de code devant chaque début de région. Pour déplier une région, il faudra alors cliquer sur le signe +.



Un moyen encore plus simple de créer une région est d'utiliser le menu contextuel de l'éditeur de code : après avoir sélectionné le code correspondant à la région, choisir **Entourer de ...**, dans le menu contextuel.

On remarquera par ailleurs, qu'on peut entourer un morceau de code de façon multiple (en particulier try , catch)...

Visual Studio permet d'appliquer facilement des modifications de formatage de texte à une sélection.

- Ajout ou suppression de tabulations
- Majuscules / Minuscules
- Suppression des espaces
- Commenter / Décommenter

Ces options sont accessibles par le menu **Edition / Avancé**.

L'option particulièrement intéressante de ce sous-menu est l'option **Mettre la sélection en forme**, qui permet, en fonction du paramétrage de l'éditeur et du langage utilisé, de mettre en forme un bloc de texte selon les normes.

III.3 REFACTORISER

Le refactoring est une opération qui consiste à retravailler le code source, pour améliorer la lisibilité et simplifier la maintenance.

Visual Studio fournit un ensemble d'outils accessible, depuis la fenêtre de code ou ...

Depuis le menu **Refactoriser**.

- **Renommer** permet de renommer une classe, une méthode une variable en proposant certaines options : prévisualisation des modifications, étendre le renommage aux commentaires ...
- **Extraire une méthode** permet après sélection d'une portion de code, d'en constituer une méthode.
- **Encapsuler le champ** permet
- **Transformer la variable locale** en paramètre permet d'extraire et de passer en paramètre une variable déclarée localement dans une méthode.
- **Extraire l'interface** permet, à partir d'une méthode, d'en extraire la signature, de créer l'interface contenant la définition de l'interface et de l'implémenter dans la classe initiale.
- **Réorganiser les paramètres** permet après avoir sélectionné les paramètres d'une méthode, de pouvoir en redéfinir l'ordre en prévisualisant le code obtenu.

Depuis le menu **Edition / Intellisense**.

- **Générer un stub de méthode** permet après avoir utilisé un nom de méthode non codée d'en extraire une signature possible.
- **Insérer un extrait**

Les extraits de code (ou snippets) sont des portions de code personnalisables qui correspondent à des tâches souvent répétitives : ouvrir un fichier, écrire une boucle foreach ..

Visual Studio affiche la liste des extraits qu'il connaît : après insertion du code choisi, (les paramètres sont encadrés en couleur), le passage d'un paramètre à l'autre se fait par la touche Tab. ; Il suffit de remplacer le paramètre par sa valeur et valider par Entrer

III.4 L'ENVIRONNEMENT GRAPHIQUE

On remarquera les possibilités de l'environnement graphique de Visual Studio 2005 :

- La liste des tâches (*smartag*)



le click sur la petite flèche donne des raccourcis rapides et simples vers des fonctions essentielles du contrôle, comme l'affectation de la valeur d'une propriété ou le lancement d'un assistant ou d'un éditeur personnalisé

- Les barres d'alignement (*snaplines*)

Permettent d'aligner horizontalement ou verticalement deux contrôles graphiques : une ligne bleue indique que le contrôle est correctement aligné, une ligne rouge indique que le texte est aligné, une ligne pointillée indique que l'espacement est correct par rapport aux autres contrôles ou aux bords de la feuille ;

Utilisées avec la touche Ctrl, les touches directionnelles du clavier permettent de poser le contrôle sur la prochaine *snapline*.

- Donner la même propriété à plusieurs composants

Il est possible, en une seule opération, d'affecter des propriétés identiques à plusieurs contrôles :

en sélectionnant l'ensemble des contrôles(par la touche MAJ ou CTRL), la fenêtre des propriétés affiche alors uniquement les propriétés communes qu'il est possible de modifier.

- Placement des contrôles les uns par rapport aux autres

Après avoir sélectionné les contrôles à traiter :

en utilisant, le menu *Format* → *Aligner*, on alignera l'ensemble des contrôles sur le dernier composant sélectionné.

en utilisant, le menu *Format* → *Uniformiser la taille*, on donnera la même taille à l'ensemble des contrôles que le dernier composant sélectionné.

- Positionner automatiquement le passage du focus

En utilisant *Affichage* → *Ordre de tabulation*, l'ordre de tabulation des contrôles de la fenêtre ayant leur propriété *TabStop* à true est affiché : pour modifier l'ordre de passage du focus, cliquez sur chaque composant dans l'ordre désiré et terminer par ECHAP.

- Ancrage des composants par rapport à la fenêtre mère

La propriété *Anchor* permet de redimensionner automatiquement la taille d'un composant, mais aussi de le repositionner automatiquement quand la fenêtre mère change de taille.

Par défaut, le composant s'ancre par rapport au bord supérieur et au bord gauche.

Cette propriété est positionnée graphiquement.

- Accoler un contrôle à un bord de fenêtre

La propriété *Dock* permet de fixer la position d'un contrôle par rapport à un bord de sa fenêtre mère : il peut alors être forcé à rester coller contre un bord(Left, top, Bottom, Right), ou s'étendre automatiquement de manière à occuper toute la largeur ou toute la hauteur de sa fenêtre mère(Fill).

Cette propriété est positionnée graphiquement.

IV LES FENETRES

IV.1 DEFINITION

Une fenêtre n'est pas seulement une zone rectangulaire de l'écran dans laquelle une application effectue des affichages : Une fenêtre est en effet définie par :

- Des attributs, ou caractéristiques, ou propriétés
par exemple, son icône affichée dans la barre des tâches ou son titre.
- Des méthodes pour agir directement sur la fenêtre, par exemple la fermer.
- Des méthodes pour traiter des événements signalés par Windows
par exemple, juste avant le premier affichage ou lors de la fermeture de la fenêtre.

Sachant qu'une classe C# regroupe des propriétés et des méthodes, on en déduit :

- les attributs correspondent aux propriétés de la fenêtre
- les méthodes traitent les événements signalés par Windows.

IV.2 LES PRINCIPALES PROPRIETES D'UNE FENETRE

Le tableau suivant décrit les propriétés les plus courantes, qui peuvent être modifiées au moment du design.

Il existe aussi des propriétés qui n'apparaissent pas dans la fenêtre Propriétés et qui ne peuvent être utilisées par programme qu'au moment de l'exécution (par exemple, la propriété `ActiveControl` qui indique le nom du contrôle ayant le focus).

Propriétés	Description
Name	Nom du formulaire. Deux formulaires d'un même projet ne peuvent pas avoir le même nom.
BackColor	Couleur d'arrière-plan par défaut des textes et graphiques d'un formulaire
BackgroundImage	Bitmap, icône ou autre fichier graphique à utiliser comme image de fond du formulaire. Si l'image est plus petite que le formulaire, elle s'affiche en mosaïque pour remplir la totalité du formulaire
ControlBox	Indique si le menu système est affiché ou non
Font	Police par défaut utilisée par les contrôles incorporés au formulaire et affichant du texte
ForeColor	Couleur de premier plan par défaut des textes et images du formulaire

FormBorderStyle	Contrôle l'aspect et le type de la bordure du formulaire. (par défaut : <i>Sizable</i>). D'autres options spécifient que les bordures ne sont pas redimensionnables ou qu'elles ne comportent pas les boutons du menu Système
Icon	Indique l'icône apparaissant dans le menu Système du formulaire et sur la barre des tâches Windows
Location	Précise les coordonnées du coin supérieur gauche du formulaire par rapport à son conteneur (l'écran lui-même ou un autre formulaire)
MaximizeBox	Indique si la commande Agrandir du menu Système et la barre de légende sont activées ou non. (activée par défaut)
MaximumSize	Indique la taille maximale du formulaire. La valeur par défaut (0, 0) signifie qu'il n'existe pas de taille maximale et que l'utilisateur peut le redimensionner à sa convenance
Menu	Indique le menu apparaissant dans la barre de menus du formulaire. (par défaut -aucun- indique que le formulaire n'a pas de menu)
MinimizeBox	Indique si la commande Réduire du menu Système et la barre de titre sont activées ou désactivées (activée par défaut)
MinimumSize	Indique la taille minimale du formulaire
StartPosition	Définit la position initiale du formulaire, par exemple <i>CenterScreen</i>
ShowInTaskBar	Détermine si le formulaire s'affiche dans la barre des tâches
Size	Taille par défaut du formulaire quand il est affiché pour la première fois.
Text	Contient le texte figurant sur la barre de titre du formulaire.
WindowState	Définit l'état initial du formulaire lorsqu'il est affiché pour la première fois. L'option par défaut (<i>Normal</i>) positionne le formulaire conformément aux propriétés Location Autres options : <i>Minimized</i> et <i>Maximized</i>

IV.3 LES EVENEMENTS D'UNE FENETRE

Les évènements liés à un composant sont repris dans la partie Evénements de la fenêtre des propriétés relative à ce composant.

Pour traiter un événement, il faut compléter la méthode qui lui est liée en « double-cliquant » sur le nom de l'événement dans la fenêtre des propriétés, de la même façon que nous avons traité le click du bouton, dans le chapitre précédent.

L'éditeur génère automatiquement le code suivant, représentant le gestionnaire d'événements :

```
private void frmExemple_Load(object sender, EventArgs e)
{
}
}
```

L'argument *sender* désigne une référence au composant qui est à l'origine de l'événement, ici la fenêtre principale.

L'autre argument, *System.EventArgs e*, contient l'information de ce qui s'est passé ; dans certains cas, nous aurons besoin de cette information.

Le cycle de vie d'une fenêtre passe par de nombreux événements :

Lorsque la méthode *Show ()* est appelée, des événements se produisent généralement dans l'ordre suivant :

Evènements	Description
Load	Se produit chaque fois qu'une feuille est chargée en mémoire. Utilisé pour effectuer des traitements avant que l'affichage ne se produise.
Activated	La fenêtre devient active (au démarrage) ou le redevient (après une réduction en icône ou au changement de fenêtre)..
Deactivate	La fenêtre a perdu son état de fenêtre active (l'utilisateur réduit la fenêtre en icône ou a terminé l'exécution)
FormClosing	L'utilisateur a marqué son intention de fermer la fenêtre et le programme peut encore refuser cette fermeture (c'est l'occasion de demander confirmation de l'opération). Le second argument de la fonction de traitement est de type <i>FormClosingEventArgs</i> . (<i>e.Cancel = true</i> annulera la fermeture de la fenêtre)
FormClosed	La fenêtre a été fermée par l'utilisateur (par ALT+F4, ou un clic sur la case de fermeture ou par arrêt de Windows).

IV.4 LES METHODES : FORMULAIRES MODAUX ET NON MODAUX

Cette classe propose des méthodes permettant de manipuler un formulaire :

Les formulaires peuvent être regroupés en deux catégories :

- Les formulaires modaux bloquent le reste de l'application lorsqu'ils sont affichés (utile pour obliger l'utilisateur à valider avant de poursuivre)
- Les formulaires non modaux ne bloquent pas l'application.

La nature des formulaires dépend directement de la manière dont ils sont affichés :

Pour afficher un formulaire **non modal**
que l'on pourra fermer
ou rendre invisible

```
frmExemple.Show () ;  
frmExemple.Close () ;  
frmExemple.Hide () ;
```

Pour afficher un formulaire **modal**

```
frmExemple.ShowDialog () ;
```

La méthode Close permet de fermer le formulaire, la méthode Focus définit le focus sur le formulaire.

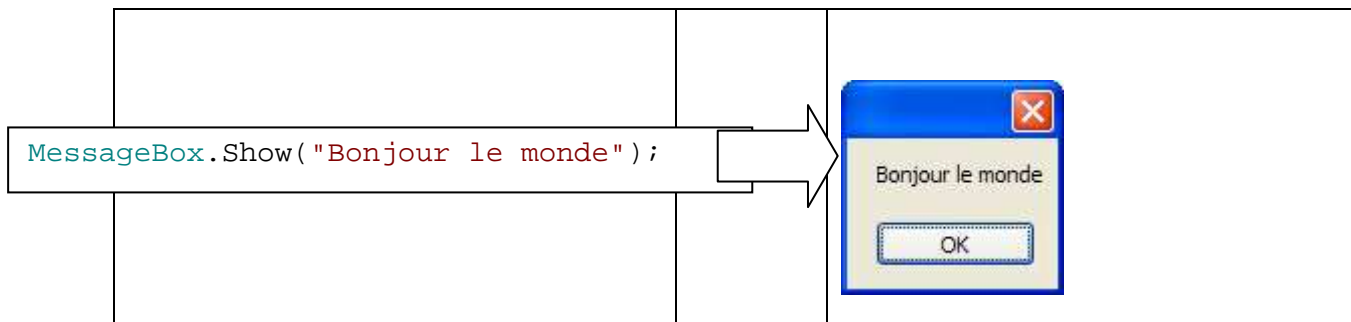
Il est plus courant d'utiliser des fenêtres modales que des fenêtres non modales : En effet, les fenêtres modales sont adaptées à la saisie de données de manière séquentielle. L'utilisation de fenêtre non modale est parfois nécessaire à l'applicatif (ex : IDE de Visual Studio)...

ATTENTION : Une application avec trop de fenêtres non modales est souvent très confuse pour l'utilisateur !

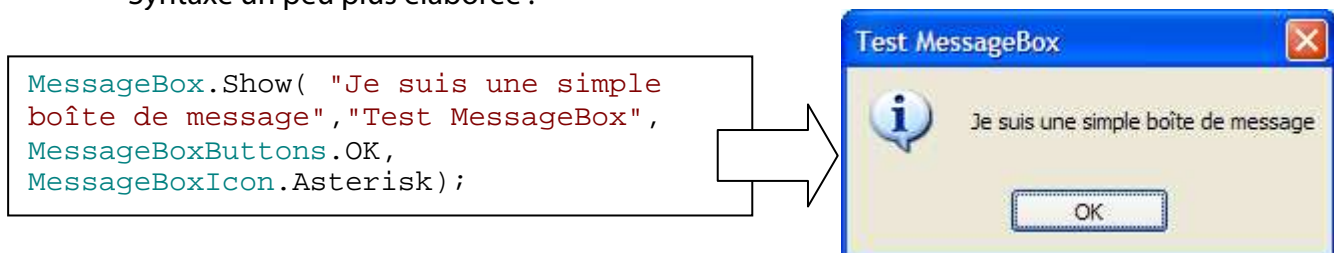
IV.5 LA CLASSE MESSAGEBOX

Elle permet, bien que ne comprenant qu'une seule fonction utile, la méthode statique Show, d'afficher une boîte de dialogue simple, pour signaler une erreur ou demander confirmation à l'utilisateur.

La syntaxe la plus simple de la méthode Show s'exprime de la façon suivante :



Syntaxe un peu plus élaborée :



La syntaxe la plus complète de la méthode Show s'exprime de la façon suivante :

```
DialogResult dr = MessageBox.Show(  
    string s,  
    string t,  
    MessageBoxButtons,  
    MessageBoxIcon,,  
    MessageBoxDefaultButton)
```

Ou

string s : Texte affiché dans la boîte de dialogue

string t : Titre de la fenêtre

MessageBoxButtons peut être l'une des valeurs suivantes :

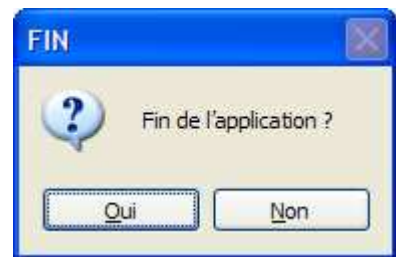
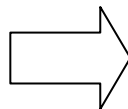
- AbortRetryIgnore : Les boutons Abandonner, réessayer et Ignorer sont affichés
- OK : Seul le bouton OK est affiché
- OKCancel : Les boutons OK et Annuler sont affichés
- RetryCancel : Les boutons Réessayer et Annuler sont affichés
- YesNo : Les boutons Oui et Non sont affichés
- YesNoCancel : Les boutons Oui, Non et Annuler sont affichés

MessageBoxIcon peut être l'une des valeurs suivantes :

Asterisk, Error, Exclamation, Hand, Information, None, Question, Stop, Warning

MessageBoxDefaultButton peut prendre la valeur Button1, Button2 ou Button3 indiquant lequel des boutons est le bouton par défaut (son contour étant souligné en gras)

```
DialogResult dr = MessageBox.Show  
( "Fin de l'application ?" , "FIN" ,  
  MessageBoxButtons.YesNo ,  
  MessageBoxIcon.Question ,  
  MessageBoxDefaultButton.Button1 ) ;  
if ( dr == DialogResult.Yes )  
  Application.Exit() ;
```



La valeur renvoyée par Show indique le bouton utilisé pour quitter la boîte de message, soit :

- Abort : Bouton Abandonner
- Cancel : Bouton Annuler ou touche ECHAP
- Ignore : Bouton Ignorer
- No : Bouton Non
- OK : Bouton OK
- Retry : Bouton essayer
- Yes : Bouton Oui

V GENERALITES SUR LES CONTROLES

Les contrôles sont des objets contenus dans des objets de type Form; on les crée, les ajoute et ensuite les positionne dans la feuille.

Chaque type de contrôle possède son propre ensemble de propriétés, de méthodes et d'événements définis dans une classe dérivée de la classe System.Windows.Forms.

Les principaux contrôles sont:

Les labels (textes non modifiables)

Les zones de saisie texte

Les zones de saisie et de mise en forme de texte

Les boutons

Les cases à cocher

Les boutons radios

Les listes déroulantes de type combo

Les listes déroulantes de type list

Les menus principaux

les menus de contexte

Les panels

Les boîtes à image

Les barres d'état

Les barres d'outils

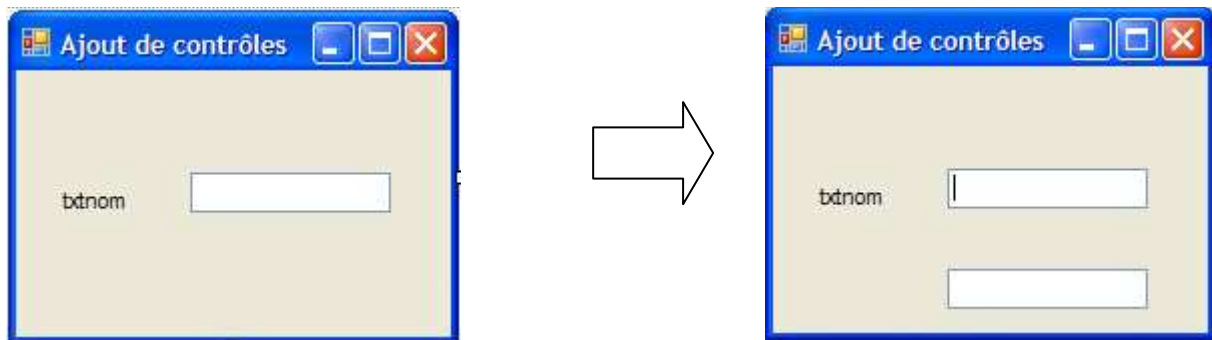
Et d'autres contrôles de liaisons aux données que nous verrons ultérieurement.

Les contrôles peuvent être insérés sur la feuille lors de sa conception, depuis la boîte à outils, comme vu dans les chapitres précédents.

Ils peuvent également être créés dynamiquement en cours d'exécution du programme.

V.1 CREATION DE CONTROLES DYNAMIQUEMENT

Par exemple la création d'une zone de texte txtPrenom au chargement de la page, positionnée en dessous de la zone de texte txtNom.



Avec le code :

```
private void frmMessage_Load(object sender, EventArgs e)
{
    TextBox newt = new TextBox();
    newt.Name = "txtPrenom";
    txtPrenom.Location = new Point(txtNom.Location.X,
    txtNom.Location.Y + 50);
    this.Controls.Add(newt);
}
```

Remarquer que la variable locale `newt` est perdue en sortie de la procédure de traitement de l'événement `Load`; cependant l'objet qui lui était rattaché est toujours dans la collection des contrôles de la forme et n'est donc pas détruit par le garbage collector.

La feuille peut alors accéder au contrôle en le recherchant dans sa collection par sa propriété **Name**

Exemple : sur un bouton de validation, on affiche le contenu des zones de texte nommées `txtNom` et `txtPrenom`:

```
private void btnValider_Click(object sender, EventArgs e)
{
    string personne = "";
    // parcourir tous les contrôles de la feuille
    foreach (Control ctrl in this.Controls)
    {
        if ((ctrl.Name == "txtNom") || (ctrl.Name == "txtPrenom"))
        {
            personne += ctrl.Text + " ";
        }
    }
    MessageBox.Show("saisie : " + personne, "",
        MessageBoxButtons.OK, MessageBoxIcon.Information);
}
```

V.2 GESTION DES EVENEMENTS

Chaque contrôle expose des événements génériques issus de la classe `Control` (clavier, souris, validation..) et aussi des événements propres à chaque classe de contrôle.

Le gestionnaire pour cet événement est créé directement en double-cliquant sur le contrôle dans la fenêtre des propriétés (affichage événement).

Il ne reste ensuite qu'à coder la réaction de l'appliquatif à l'événement ainsi généré, comme vu précédemment

Tous les contrôles lèvent des événements suite à une action de l'utilisateur.

Un contrôle a le *focus*, lorsqu'il reçoit les entrées du clavier (et si sa propriété `CanFocus` est à `true`).

Le *focus* passe d'un contrôle à un autre quand l'utilisateur active ce contrôle, avec souris ou clavier ou appuie sur *Tab*.

Un contrôle génère les événements de *focus* suivants, dans l'ordre : (voir *Introduction*)

- Enter
- GotFocus
- Leave / LostFocus
- Validating / Validated

Ces événements s'enchaînent souvent de la sorte: `MouseDown` → `Click` → `MouseUp`

- a. On peut alors souhaiter traiter ou non un événement pour un contrôle
On a vu que pour capter et traiter l'événement, on devra lui associer une procédure de traitement par le biais d'un **délégué**:
- b. On peut enfin souhaiter que plusieurs contrôles soient rattachés à un même traitement.

Exemple : Dans notre additionneur, une procédure de traitement par événement **click** était définie sur chaque bouton, de 0 à 9 :

Par exemple, pour le bouton 8 :

```
private void btn8_Click(object sender, EventArgs e)
{
    txtVisu.Text = txtVisu.Text + btn8.Text + "+";
    total = total + 8;
}
```

On peut définir la même procédure de traitement :

```
private void btnx_Click(object sender, EventArgs e)
{
    txtVisu.Text = txtVisu.Text + ((Button)sender).Text + "+";
    total = total + Convert.ToInt32(((Button)sender).Text);
}
```

Qui sera associée automatiquement, (par le choix de cette procédure de traitement dans la fenêtre des propriétés) à l'évènement click du bouton par le biais d'un délégué.

```
this.btn0.Click += new System.EventHandler(this.btnx_Click);
this.btn8.Click += new System.EventHandler(this.btnx_Click);
this.btn9.Click += new System.EventHandler(this.btnx_Click);
```

Dans la procédure de traitement, on remarquera le casting du sender, pour récupérer la propriété Text.

V.3 LA VALIDATION DES CONTROLES

Il faut toujours vérifier la validité des données entrées par un utilisateur dans un contrôle, avant de traiter ce contenu, surtout dans le cas d'une saisie numérique attendue.

La propriété Text d'une zone de texte étant par défaut une chaîne de caractères, elle peut accueillir n'importe quel caractère frappé au clavier, numérique comme alphabétique.

Pour effectuer des calculs, cette zone de texte devra être convertie en un type C# numérique (int, long, float, double ...) et pour être convertie sans générer d'erreur (voir le support de cours Initiation C#), le développeur devra s'assurer que les caractères entrés sont bien des chiffres ou séparateurs décimaux. Ces premiers contrôles sont des contrôles de validité.

Les informations ont également besoin d'être contrôlées par rapport aux règles de gestion de l'application (plages de valeur, saisie obligatoire ...)

V.3.1 Les évènements à utiliser

Sur quels évènements pouvons nous envisager d'effectuer les contrôles ?

- Sur le click d'un bouton **Valider** qui effectuerait l'ensemble des contrôles de la feuille.
- Sur l'évènement **Validating** de la zone de texte nécessitant un contrôle.
L'évènement **Validating** se produit lorsque le focus quitte un contrôle et passe à un contrôle dont la propriété **CausesValidation** est true : au cas où les données saisies seraient erronées, on définira la propriété **Cancel** du paramètre **CancelEventArgs** à true pour empêcher le passage du focus au contrôle suivant.
Pour éviter que l'évènement **Validating** se produise lorsque l'utilisateur veut quitter la feuille (bouton Quitter) ou annuler sa saisie (bouton Annuler), on positionnera la propriété **CausesValidation** de ces deux boutons à **false**.

La structure d'évènement **CancelEventArgs** propose une donnée :

- **Cancel** qui positionnée à **true** annule l'évènement

L'évènement **Validated** se déroule après l'évènement **Validating** (s'il n'a pas été annulé), mais avant que le contrôle ne perde le focus il est impossible d'annuler cet évènement

Exemple : Contrôle, en sortie de champ que le nombre entré dans la zone de texte txtAge est bien supérieur ou égal à 18.

```
private void txtAge_Validating(object sender,
System.ComponentModel.CancelEventArgs e)
{
    if (Convert.ToInt16(txtAge.Text)<18)
    {
        // interdit le passage du focus au contrôle suivant
        e.Cancel=true;
        // affiche une messagebox d'erreur
        MessageBox.Show("Age invalide") ;
    }
}
```

Lorsque txtAge.text est inférieur à 18, le focus reste dans la zone de texte txtAge, et un message d'erreur est affiché.

Attention : en amont, on aura évidemment contrôlé qu'on pouvait convertir le nombre saisi dans txtAge en entier.

- A la saisie de chaque caractère dans la zone de texte grâce à l'évènement **KeyPress** ou l'évènement **TextChanged**.

Les événements de touche se produisent dans l'ordre suivant : **KeyDown** , **KeyPress**, **KeyUp**.

L'évènement **KeyPress** n'est pas déclenché par les touches qui ne sont pas de type caractère ; cependant, ces touches déclenchent les événements **KeyDown** et **KeyUp**.

La structure d'évènement `KeyPressEventArgs` propose 2 propriétés :

- **Keychar** retourne le caractère frappé par l'utilisateur
- **Handled** positionné à `true` permet d'annuler l'évènement **KeyPress** ; dans ce cas, le caractère en erreur n'apparaît pas dans la zone de texte.

Exemple : Contrôle que le caractère entré est bien un chiffre, ou la touche `BackSpace`.

```
private void txtFact1_KeyPress(object sender,
System.Windows.Forms.KeyPressEventArgs e)
{
    if (!char.IsNumber(e.KeyChar)& e.KeyChar!= (char)Keys.Back )
    {
        e.Handled=true;
    }
}
```

Ces évènements peuvent être complémentaires : On pourra par exemple utiliser un évènement **KeyPress** pour contrôler la saisie de caractères numériques, et utiliser un évènement **Validating** pour s'assurer que le nombre de caractères entrés est conforme.

V.3.2 Les modes de contrôle

Définition et utilisation des expressions régulières

Une expression régulière est une suite de caractères qui définit des règles de contraintes sur des chaînes que l'on souhaite valider selon un certain format et un certain contenu.

Deux types de caractères peuvent être utilisés dans les expressions régulières :

les méta caractères `. \ ? * + { } () []` et les caractères normaux (tout autre caractère, y compris les symboles précédents qui devront être alors représentés par des séquences d'échappement).

Par exemple: (notes prises sur le site <http://www.aidejavascript.com/article64.html>)

1. Les chaînes alphabétiques

`[A-Za-z]` permet de vérifier qu'un caractère fait partie des lettres "A" à "Z" et "a" à "z" (en dehors de tout caractère accentué). Les slashes / délimitent l'expression régulière. Mais on peut aussi bien mettre des guillemets :

`/[A-Za-z]/` ou `"[A-Za-z]"`

Pour indiquer que nous ne voulons QUE des caractères alphabétiques, il faut écrire : `/^[A-Za-z]+$`

Les 2 caractères « ^ » et « \$ » indiquent qu'il faut établir le contrôle du début à la fin de la chaîne. Le caractère « + » indique que le caractère alphabétique doit être présent au moins une fois.

2. Les chaînes numériques

/^[0-9]*\$/ L'expression [0-9] indique qu'on n'attend que des chiffres. On peut aussi utiliser « \d » à la place de [0-9]. L'astérisque « * » signifie que le caractère peut être absent ou présent plusieurs fois (alors que « + » implique que le caractère soit présent au moins une fois). Donc si la chaîne est vide, elle sera considérée comme correcte, selon l'expression régulière.

3. Les dates

/^[0-9]+\V[0-9]+\V[0-9]+\$/ Il s'agit seulement d'une date au format numérique, dans le style "14/7/2003". Dans l'expression régulière, on a 3 parties : le jour, le mois et l'année, qui sont des chiffres répétés plusieurs fois. Ils sont séparés par le signe « \V » : on ne peut pas mettre simplement « / », parce que ce signe indique l'encadrement de l'expression régulière, donc il faut « échapper » le slash par un antislash (« \ »).

/^[0-9][0-9]?\\V[0-9][0-9]?\\V[0-9][0-9]([0-9][0-9])?\$/ Avec cette expression régulière on est obligé d'écrire le jour et le mois avec un ou 2 chiffres, et on doit écrire l'année sur 2 ou 4 chiffres.

C'est-à-dire : « 14/7/03 » ou « 14/7/2003 » ou « 14/07/2003 »

On a introduit quelques nouveautés :

le « ? » indique que le caractère précédent peut être présent 0 fois ou 1 fois, donc « [0-9][0-9] ? » signifie qu'on peut écrire un ou 2 chiffres ;

les parenthèses permettent d'affecter le quantificateur (+, *, ?) à la série de caractères entre ces parenthèses : « ([0-9][0-9]) ? » signifie que ces 2 caractères peuvent être présents 0 fois ou 1 fois.

Si l'on souhaite d'autre séparateur que le / pour la date, il faudra utiliser « (\\V|\\.|\\.) » qui autorise soit le slash, soit le tiret, soit le point par exemple

4. Les chaînes alphanumériques

/^[0-9A-Za-z]+\$/ On peut aussi utiliser le caractère « \\w », qui autorise les caractères alphanumériques ou le caractère de soulignement (_).

5. Les codes de couleur

/^#[0-9A-F]+\$/ Un code couleur est constitué d'un « # », suivi d'un nombre hexadécimal, qui ne peut contenir que des chiffres ou des lettres de A à F.

6. Une adresse e-mail

/^[_a-z0-9]+(\\.[_a-z0-9]+)*@[a-z0-9]+(\\.[a-z0-9]+)*\$/

Une adresse e-mail, par exemple « moi@mon-domaine.fr » est constituée de 3 parties (en dehors du « @ ») :

- L'utilisateur « moi »

Il peut être sous la forme « xxx_yyy-zzz.xxx_yyy-zzz.xxx_yyy-zzz... »

Il est donc contrôlé par la séquence « `[a-z0-9-]+(\.[a-z0-9-]+)*` », où « `[a-z0-9-]` » représente les caractères alphanumériques, plus le caractère de soulignement, plus le tiret. La 2e partie « `(\.[a-z0-9-]+)*` » permet d'ajouter des mots séparés par un point.

- Le nom de domaine « mon-domaine »

Il suit l'arobase et ne peut contenir que des caractères alphanumériques, le caractère de soulignement, et le tiret : « `[a-z0-9-]+` »

- Le TLD (top level domain) : « fr ».

Il est constitué seulement d'un point suivi de caractères alphanumériques, éventuellement répété plusieurs fois : « `(\.[a-z0-9-]+)*` »

Voici les principaux caractères spéciaux que vous pourrez utiliser dans vos expressions régulières.

Caractères spéciaux	Signification
\	Si le caractère suivant est n'est pas un caractère spécial, cela signifie que ce caractère doit être considéré comme un caractère spécial. Exemple : \d signifie un chiffre Si le caractère suivant est un caractère spécial, cela signifie qu'il faut prendre ce caractère de façon littérale. Exemple * cherche la présence d'un astérisque *
^	indique l'emplacement où doit commencer la chaîne de caractère à contrôler
\$	indique l'emplacement où doit finir la chaîne de caractère à contrôler
*	indique que le caractère précédent doit être présent 0 fois ou plusieurs fois (soit absent soit présent ou répété)
+	indique que le caractère précédent doit être présent 1 fois ou plusieurs fois (donc au moins une fois)
?	indique que le caractère précédent doit être présent 0 ou 1 fois (soit absent soit présent mais pas répété)
(x)	le caractère doit correspondre à x et permet de mettre ce caractère en mémoire (sert pour la fonction "exec" qui découpe la chaîne testée dans un tableau)
	ex : x y, le caractère doit correspondre à x OU à y (" " est le caractère CTL-ALT 6 ou AltGr 6 sur PC)
{n}	si n est un nombre, le caractère précédent doit être présent n fois
{n, p}	si n et p sont des nombres, le caractère précédent doit être présent au minimum n fois et au maximum p fois
[abc]	le caractère doit correspondre aux caractères entre crochets ("a", "b" ou "c")
[^abc]	le caractère ne doit pas correspondre aux caractères entre crochets ("a", "b" ou "c")
\s	le caractère doit correspondre à un espace, un retour chariot, ou un caractère de tabulation
\S	correspond à tous les caractères sauf l'espace
\d	correspond à [0-9], c'est-à-dire à un chiffre
\D	tout caractère sauf un chiffre
\w	correspond aux caractères alphanumériques + le "_" (équivalent à [A-Za-z0-9_])
\W	correspond à tous les caractères sauf les caractères alphanumériques et le "_" (équivalent à [^A-Za-z0-9_])

Le Framework fournit des classes qui vont permettre de manipuler les expressions régulières:

La classe **Regex** de l'espace de noms **System.Text.RegularExpressions** contient plusieurs méthodes statiques qui vous permettent d'utiliser une expression régulière.

Ainsi le contrôle d'une date pourra se faire de la sorte:

```
bool DateValide (string strDate)
{
    return Regex.IsMatch(strDate, @"^[0-9]+\\"/>
}
```

On pourra aussi vouloir remplacer ou supprimer des caractères dans une chaîne:

Regex.Replace (chaîne, expression, remplacement) ou **Regex.Replace (chaîne, expression, "")**

Regex.Escape convertit une chaîne de manière à ce qu'elle puisse être utilisée en toute sécurité comme constante dans une expression régulière (remplace les méta caractères par leur valeur d'échappement)

Regex.Unescape fait l'inverse

Regex.Match recherche dans une chaîne d'entrée une occurrence d'une expression régulière et retourne le résultat précis comme objet Match.

Regex.Matches recherche dans une chaîne d'entrée toutes les occurrences d'une expression régulière et retourne toutes les correspondances dans un objet MatchCollection

Regex.Split fractionne une chaîne d'entrée en un tableau de sous chaînes aux positions définies par une correspondance d'expression régulière.

La méthode TryParse

Elle convertit la représentation sous forme de chaîne en un nombre en entier 16 bits, 32 bits, 64 bits, double, décimal ... équivalent. Une valeur de retour indique si l'opération a réussi, **true** si l'opération a réussi, **false** sinon.

Cette méthode est l'équivalent de la méthode **Parse**, à la différence qu'elle ne lève pas d'exception en cas d'échec de la conversion.

Exemple : Contrôle de la validité d'un champ réel saisi

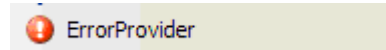
```
float montant;
if (!Single.TryParse(txtMontant.Text, out montant))
{
    MessageBox.Show("Montant invalide");
}
```

V.3.3 Affichage à l'utilisateur

Toute information invalide doit être signalée à l'utilisateur, par envoi d'un message d'information (MessageBox), ou par positionnement du focus et mise en inversion vidéo du champ en erreur (voir les propriétés spécifiques des TextBox), ou grâce à l'utilisation du contrôle ErrorProvider.

Utilisation du contrôle **ErrorProvider**

Si plusieurs messages d'erreur risquent d'apparaître à la suite, l'utilisateur devra se souvenir de tous ces messages : une meilleure solution consiste à utiliser ce contrôle.



Le contrôle sélectionné dans la boîte à outils s'affiche sous le formulaire.

La propriété **BlinkStyle**, positionnée à **BlinkIfDifferentError**, provoquera le clignotement en cas d'erreur.

La propriété **BlinkRate** permet de ralentir ou accélérer ce clignotement.

Lorsqu'une autre erreur est détectée, l'icône ne recommence à clignoter que s'il s'agit d'une erreur différente : en attribuant à la propriété **BlinkStyle**, la valeur **AlwaysBlink**, l'icône clignotera à chaque erreur.

Utilisation de ce contrôle d'erreur

```
private void txtAge_Validating(object sender,
System.ComponentModel.CancelEventArgs e)
{
```

```
    try
    {
        int age = Convert.ToInt32(txtAge.Text);
        if (Convert.ToInt32(txtAge.Text) < 18)
        {
            e.Cancel=true;
            errorProvider.SetError(txtAge, "Age invalide");
        }
        else
        {
            errorProvider.SetError(txtAge, "");
        }
    }
    catch
    {
        e.Cancel=true;
        errorProvider.SetError(txtAge, "Saisie invalide");
    }
}
```

Nom du contrôle sur lequel porte l'erreur

Message affiché
dans l'infobulle

Suppression de l'icône d'erreur

Un aperçu de l'exécution

V.3.4 Gérer le focus

Pour donner le focus à un contrôle spécifique, il suffit d'employer la méthode **Focus** exposée par la classe **Control**. La méthode renvoie **true** si le focus a bien été transmis au contrôle, et **false** dans le cas contraire.

```
if ( !txtNom.Focus() )  
{  
  
}
```

ou plus simplement, pour positionner le curseur sur le champ txtNom.

```
txtNom.Focus();
```







Un contrôle ne peut obtenir le focus que s'il fait partie d'un formulaire actif ;

Le focus peut être contrôlé grâce aux propriétés suivantes de la classe *Control*.

Noms	Description
CanFocus	Renvoie true si le focus peut être donné au contrôle
ContainsFocus	Renvoie true si le contrôle ou un de ses enfants possède le focus
Focused	Renvoie true si le contrôle possède le focus (idem ContainsFocus sans les contrôles enfants)

VI LES DIFFERENTS CONTROLES GRAPHIQUES

VI.1 LES ZONES D’AFFICHAGE ET D’EDITION

 TextBox	Contrôle permettant à l'utilisateur de saisir tous les caractères du clavier.
 MaskedTextBox	Contrôle TextBox amélioré qui prend en charge une syntaxe déclarative pour accepter ou refuser une entrée d'utilisateur
 Label	Zones d'affichage de texte
 LinkLabel	Zones d'affichage représentées comme des hyperliens
 NumericUpDown	Composant d'incrémentement et de décrémentation numériques
 DomainUpDown	Composant de sélection par flèches

VI.1.1 Les zones de texte (TextBox)

Les contrôles sont des objets contenus dans des objets de type Form; on les crée, les ajoute et ensuite les positionne dans la feuille.

Les zones de texte sont les contrôles d'édition standard de Windows, qui permettent de saisir du texte. La classe **TextBox** qui gère les zones de texte dérive de la classe **TextBoxBase**, classe de base pour les classes **TextBox** et **RichTextBox**.

Le contrôle **TextBox** est une simple zone de saisie ; la propriété **MultiLine** permet de saisir plusieurs lignes de texte, dont on peut extraire le contenu avec un tableau de **string** avec la propriété **Lines**. La propriété **ReadOnly** fait passer le contrôle en lecture seule.

Il est du rôle du développeur d'implémenter toute la logique de vérification des données saisies dans une **TextBox**.

La propriété **MaxLength** indique le nombre de caractères maximum saisissable.

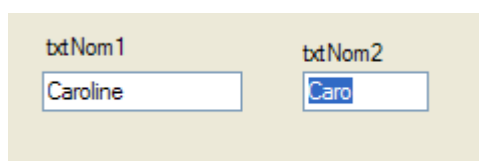
Membres remarquables des zones d'édition

Noms	Description
Propriétés	
CharacterCasing	Types de caractères que l'on peut taper (Lower, Normal, Upper)
Lines	Tableau de chaînes de caractère contenant chacune des lignes d'une zone d'édition multiligne
MultiLine	Permet de pouvoir saisir plusieurs lignes
MaxLength	Nombre maximal de caractères que l'utilisateur peut introduire dans la zone
Modified	Indique si l'utilisateur a modifié le contenu de la zone d'édition
PassWordChar	Spécifie un symbole de remplacement comme caractère de substitution pour cacher des données sensibles
ReadOnly	Indique si la zone d'édition est en lecture seule
ScrollBars	Indique quelles barres de défilement sont éventuellement affichées (si MultiLine = true)
SelectedText	Texte sélectionné
SelectionLength	Nombre de caractères sélectionnés
SelectionStart	Indice du premier caractère sélectionné
Text	Contient le texte saisi ou affiché dans la zone de texte
TextAlign	Cadrage du texte dans la zone d'édition
TextLenght	Longueur du texte saisi ou affiché
WordWrap	En cas de saisie Multiline, permet de renvoyer la saisie à la ligne au bord droit du contrôle
Méthodes	
void Clear()	Supprime le texte

Exemple : Le code suivant

```
// Pour sélectionner des caractères de la textBox
txtNom1.SelectionStart = 0;
txtNom1.SelectionLength = 4;
txtNom2.Text = txtNom1.SelectedText;
```

donne le résultat



VI.1.2 Les MaskedTextBox

Contrôle dérivé du contrôle **TextBox** et qui en reprend donc toutes ses fonctionnalités, le contrôle **MaskedTextBox** possède une propriété **Mask** qui permet de restreindre la saisie selon un modèle prédéfini :

Plusieurs types de masques sont proposés par défaut, avec la possibilité de définir ses propres masques.

La propriété **ValidatingType** permet de définir le type de données attendu à la saisie utilisateur.

```
mskDate.ValidatingType = typeof(System.DateTime);
```

Propriétés remarquables des MaskedTextBox

Noms	Description
AsciiOnly	Booleen indiquant si le contrôle MaskedTextBox accepte des caractères en dehors du jeu de caractères ASCII.
CutCopyMakFormat	Obtient ou définit une valeur qui détermine si les littéraux et les caractères d'invite sont copiés
Mask	Obtient ou définit le masque de saisie à utiliser au moment de l'exécution
MaskCompleted	Booléen indiquant si toutes les entrées requises ont été entrées dans le masque de saisie
MaskFull	Booléen indiquant si toutes les entrées requises et facultatives ont été entrées dans le masque de saisie
PromptChar	Caractère utilisé pour représenter un caractère à saisir
RejectInputOnFirstFailure	Booléen indiquant si l'analyse de l'entrée d'utilisateur doit s'arrêter après que le premier caractère non valide est atteint
ResetOnPrompt	Booléen qui détermine comment un caractère d'entrée qui correspond au caractère d'invite doit être géré
ResetOnSpace	Booléen qui détermine comment un caractère d'entrée de type espace doit être géré
SelectedText	Obtient ou définit la sélection en cours dans le contrôle
TextAlign	Obtient ou définit la façon d'aligner le texte dans le contrôle.
TextMaskFormat	Obtient ou définit une valeur qui détermine si les littéraux et les caractères d'invite sont inclus dans la chaîne mise en forme.
ValidatingType	Obtient ou définit le type de données utilisé pour vérifier les données entrées par

	l'utilisateur.
--	----------------

L'événement **TypeValidationCompleted** est déclenché pour exécuter le traitement de la validation de masque ou de type. Il reçoit un paramètre **TypeValidationEventArgs** qui contient des informations sur la conversion, par exemple, le membre **IsValidInput** qui indique si la conversion a réussi.

Exemple :

```
private void mskDate_TypeValidationCompleted(object sender,
TypeValidationEventArgs e)
{
    if (!e.IsValidInput)
    {
        errorProvider1.SetError(mskDate, "Date invalide");
        e.Cancel = true;
    }
}
```

VI.1.3 Les zones d'affichage (Labels)

Un label est une zone d'affichage pouvant être initialisée et modifiée par programme (propriété **Text**), mais ne pouvant pas être modifiée par l'utilisateur.

Propriétés remarquables des labels

Noms	Description
Autosize	La taille du contrôle s'adapte au texte à afficher
Image	Image à afficher
ImageAlign	Alignement de l'image
ImageIndex	Index de l'image sélectionnée dans la liste d'images référencée dans <i>ImageList</i>
ImageList	Liste d'images à utiliser
Text	Texte affiché dans la zone d'affichage
TextAlign	Cadrage du texte dans la zone d'affichage

VI.1.4 Les LinkLabels

Les contrôles **LinkLabel** sont des zones d'affichage, mais qui présentent une caractéristique de bouton : tout le texte du contrôle (propriété **Text**) ou une partie seulement (propriété **LinkArea**) peut servir d'hyperlien.

Le texte peut même comprendre plusieurs hyperliens (propriété **Links** utilisable par programme uniquement). L'hyperlien peut avoir n'importe quel usage, à programmer dans la fonction de traitement de l'événement **LinkClicked**.

Propriétés remarquables des LinkLabels

Noms	Description
LinkArea	Portion du texte devant être considéré comme hyperlien
LinkBehavior	Comportement de l'hyperlien (soulignement par défaut ou non)
LinkColor	Couleur d'affichage de l'hyperlien (par défaut bleu)
Links	Collection d'hyperliens définis dans Text , chaque élément étant un objet Link
LinkVisited	Indique si un hyperlien doit être affiché différemment quand il a été visité
VisitedLinkColor	Couleur d'un lien déjà visité

VI.1.5 Les UpDown

Les contrôles **Up** et **Down** associent des petites flèches à une zone d'édition. En cliquant sur l'une des flèches, on peut incrémenter ou décrémenter la valeur affichée dans la zone d'édition, qui peut contenir des nombres, éventuellement décimaux (**NumericUpDown**) ou du texte (**DomainUpDown**).

Propriétés remarquables des NumericUpDown

Noms	Description
DecimalPlaces	Nombre de décimales (0 par défaut)
Hexadecimal	Indique si la partie zone d'édition affiche les valeurs hexadécimales
Increment	Valeur d'incrément ou de décrément (à chaque fois que l'utilisateur clique sur l'une des flèches)
InterceptArrowsKeys	Indique si les touches de direction et peuvent être utilisées pour modifier le contenu de la zone d'édition
Maximum	Valeur maximale (100 par défaut)
Minimum	Valeur minimale (0 par défaut)




ThousandsSeparator	Indique si le séparateur des milliers doit être affiché.
UpDownAlign	Position des flèches par rapport à la zone d'édition
Méthodes	
void DownButton()	Simule un clic sur la flèche vers le bas
void UpButton()	Simule un clic sur la flèche vers le haut
Evènements	
ValueChanged()	L'utilisateur a cliqué sur l'une des flèches ou la valeur a été directement modifiée à partir du clavier

La classe **DomainUpDown** permet de sélectionner une chaîne de caractères : elle présente de nombreuses similitudes avec une boîte combo.

Propriétés remarquables des DomainUpDown

Noms	Description
InterceptArrowsKeys	Indique si les touches de direction et peuvent être utilisées pour modifier le contenu de la zone d'édition
Items	Collection des éléments présentés
Sorted	Indique si les éléments sont triés
Text	Libellé de l'élément sélectionné
UpDownAlign	Position des flèches par rapport à la zone d'édition
Wrap	Indique s'il y a passage automatique de la dernière valeur à la première
Méthodes	
void DownButton()	Simule un clic sur la flèche vers le bas
void UpButton()	Simule un clic sur la flèche vers le haut
Evènements	
SelectedItemChanged()	L'utilisateur a sélectionné un nouvel élément-

VI.2 BOUTONS ET CASES

 Button	Boutons de commande
 CheckBox	Cases à cocher (CheckBox)
 RadioButton	Boutons radio (RadioButton)

La plupart des méthodes et propriétés des classes **Button**, **RadioButton** et **CheckBox** sont héritées de la classe **ButtonBase**.

Le principal événement traité sur ces classes est l'événement **Click**.

Les contrôles **CheckBox** et **RadioButton** possèdent une fonction similaire : ils permettent à l'utilisateur d'effectuer son choix parmi une liste d'options. Les contrôles **CheckBox** permettent à l'utilisateur de sélectionner une combinaison d'options. Les contrôles **RadioButton** permettent par contre à l'utilisateur d'effectuer son choix parmi des options s'excluant mutuellement.

VI.2.1 Les boutons de commande (Button)

Les contrôles **Button**, sélectionnés par un click de souris, servent à déclencher une action.

Définissez la propriété **AcceptButton** ou **CancelButton** de la feuille pour permettre aux utilisateurs de sélectionner un bouton en appuyant sur la touche ENTRÉE ou ÉCHAP même si le bouton n'a pas le focus.

Propriétés remarquables

Noms	Description
ImageList	Référence à une liste d'images
ImageIndex	Numéro d'images dans la liste d'images
FlatStyle	Définit la manière dont les bords des contrôles sont affichés.

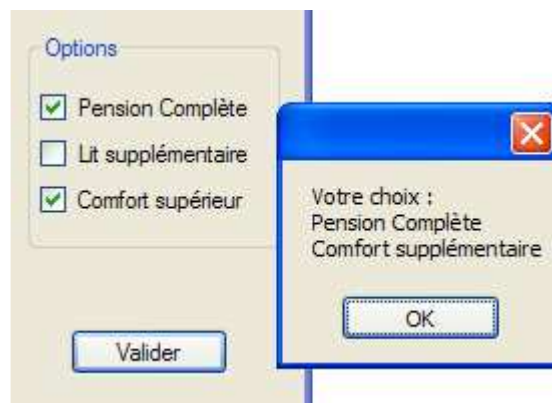
VI.2.2 Les cases à cocher (CheckBox)

La propriété **Appearance** détermine si **CheckBox** s'affiche sous la forme de **CheckBox** traditionnelle ou sous la forme d'un bouton.

La propriété **ThreeState** détermine si le contrôle prend en charge deux ou trois états. Utilisez la propriété **Checked** pour obtenir ou définir la valeur d'un contrôle **CheckBox** à deux états et utilisez la propriété **Checkstate** pour obtenir ou définir la valeur d'un contrôle **CheckBox** à trois états.

Exemple : Au click sur le bouton Valider, on affichera les options choisies par l'utilisateur, sachant que les cases à cocher se nomment respectivement `cbPensionComplete`,

cbLitsup et cbComfort.



Le code suivant permet d'afficher la messageBox résultat

```
private void btnValider_Click(object sender, EventArgs e)
{
    string strChoix = "Votre choix :\n" ;
    if (cbPensionComplete.Checked) strChoix += "Pension Complète\n";
    if (cbLitsup.Checked) strChoix += "Lit Supplémentaire\n";
    if (cbComfort.Checked) strChoix += "Comfort supplémentaire\n";
    MessageBox.Show(strChoix);
}
```

Propriétés remarquables

Noms	Description
ThreeState	Indique si la case est à 2 ou 3 états false : 2 états (checked et unchecked) true : 3 états (état supplémentaire : grisé)
Checked	Etat (coché ou non) d'une case à 2 états
CheckedState	Etat d'une case à 3 états

Généralement, un programme ne traite pas les événements d'une case à cocher : il se contente de lire l'état des cases lorsqu'il a besoin de l'information.

Toutefois, le programme peut traiter l'un des 2 événements ci-dessous :

Noms	Description
CheckedChange	Changement d'état d'une case à cocher à 2 états
CheckedStateChange	Changement d'état d'une case à cocher à 3 états

VI.2.3 Les Boutons Radio (RadioButton)

Les boutons Radio présentent les mêmes propriétés et méthodes que les cases à cocher, sauf qu'elles ne présentent pas l'état supplémentaire, et ne possèdent donc pas ni les propriétés ni les événements relatifs à cet état.

Ces contrôles permettent à l'utilisateur de choisir une option particulière parmi **plusieurs options réunies dans un groupe**.

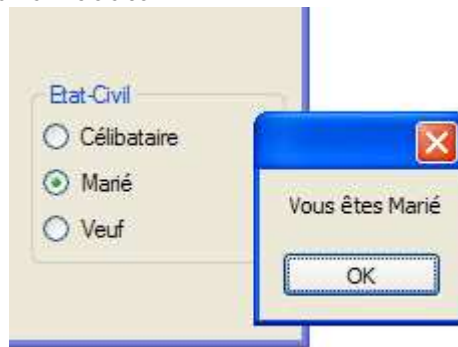
Il est impossible de sélectionner simultanément plusieurs options dans un même groupe.

Exemple : Le code suivant

```
private void rbEtatCivil_CheckedChanged(object sender, EventArgs e)
{
    if (((RadioButton)sender).Checked == true)
    {
        MessageBox.Show("Vous êtes " + ((RadioButton)sender).Text);
    }
}

}
```

donne le résultat au click sur un bouton



Les boutons radio se nomment respectivement `rbCelibataire`, `rbMarie` et `rbVeuf`.




Le traitement de leur évènement Click a été associé à la même méthode.

`rbEtatCivil_CheckedChanged`

```
this.rbCelibataire.Click += new
System.EventHandler(this.rbEtatCivil_CheckedChanged);
this.rbMarie.Click += new
System.EventHandler(this.rbEtatCivil_CheckedChanged);
this.rbVeuf.Click += new
System.EventHandler(this.rbEtatCivil_CheckedChanged);
```

VI.3 LES BOITES DE LISTE

Les boites de liste sont représentées par les contrôles suivants :

 ListBox	Boite de liste standard
 CheckedListBox	Boite de liste avec cases à cocher
 ComboBox	Boite de liste déroulante

Les boites de liste affichent un ensemble d'éléments (**items**): il s'agit en général de texte, mais la liste peut contenir des objets. L'utilisateur peut sélectionner un ou plusieurs éléments. Le contrôle affiche automatiquement des barres de défilement si le contenu dépasse le cadre du contrôle.

La liste des éléments peut être établie pendant la phase de conception, en utilisant l'éditeur de collections de la propriété **Items**; toutefois des éléments peuvent être insérés, supprimés pendant la phase d'exécution en utilisant les méthodes de la collection **Items**.

```
lstNoms.Items.Add("Jean"); // ajoute en fin de liste
lstNoms.Items.Insert(0, "Anne"); // ajoute en position 0
lstNoms.Items.Remove(strNom);
```

Sachant que les collections démarrent à l'indice 0, le libellé du (i-1) ème article de la liste de nom lstNoms est obtenu par :

```
String s = (string)lstNoms.Items[i];
```

Pour obtenir l'élément sélectionné dans une boite de liste à sélection unique, on utilisera la propriété **SelectedIndex** (pour les indices) ou **SelectedItem**(pour le libellé):

```
String s = (string)lstNoms.SelectedItem;
MessageBox.Show("Le nom sélectionné est " + s);
```

ou

```
String s = (string)lstNoms.Items[lstNoms.SelectedIndex];
MessageBox.Show("Le nom sélectionné est " + s);
```

VI.3.1 Les boites de liste standard (ListBox)

Propriétés les plus importantes des boites de liste

Noms	Description
Propriétés	
MultiColumn	Positionnée à true, la ListBox affiche ses éléments en plusieurs colonnes et une scrollBar horizontale apparaît.
ScrollAlwaysVisible	Positionnée à true, une ScrollBar apparaît selon le nombre d'éléments.
SelectionMode	Détermine le nombre d'éléments pouvant être sélectionnés à la fois.
Items	Collection des libellés des éléments de la

	liste (dont la propriété Count détermine le nombre d'éléments de la liste)
Propriétés Run-Time	
SelectedIndex	Index de l'élément sélectionné (0 pour le premier de la liste, -1 si aucune sélection)
SelectedIndices	Collection des indices des éléments sélectionnés
SelectedItem	Élément sélectionné(Etant de type Object, il peut s'appliquer à n'importe quel type ou classe- string par exemple-mais un casting est toujours nécessaire.)
SelectedItems	Collection des éléments sélectionnés
Sorted	Permet de trier les éléments automatiquement

La propriété **SelectedItem** donne l'élément sélectionné dont l'indice dans la liste est **SelectedIndex**. En cas de sélection multiple, on utilisera les collections **SelectedItems** et **SelectedIndices**.

Pour obtenir les éléments sélectionnés dans une boîte de liste à sélection multiples, on balayera la collection **SelectedIndices** (pour les indices) ou **SelectedItems**(pour les libellés):

```
foreach (string s in lstNoms.SelectedItems)
{
    ...
}
```

Méthodes les plus importantes des boîtes de liste

Noms	Description
Void ClearSelected()	Désélectionne tous les éléments.
int FindString(string str) int FindString(string str, int index)	Recherche le premier élément qui commence par la chaîne spécifiée (à partir d'une position donnée).
int FindStringExact(string str) int FindStringExact(string str, int index)	Recherche le premier élément qui correspond à la chaîne spécifiée (à partir d'une position donnée).
bool GetSelected(int index)	Retourne un booléen indiquant si l'élément spécifié est sélectionné.
void SetSelected(int index, bool value)	Sélectionne ou efface la sélection pour l'élément spécifié.

Gérer les évènements de sélection

Noms	Description
SelectedIndexChanged	Générés lorsqu'un élément est sélectionné ou désélectionné dans la zone de liste
SelectedValueChanged	
Click	Lorsque l'utilisateur clique sur un des éléments
DoubleClick	Lorsque l'utilisateur double-clique sur un des éléments

L'évènement **SelectedIndexChanged** est déclenché à chaque nouvelle sélection dans la liste.

Propriétés applicables à la propriété Items	
Count	Nombre d'objets dans la collection
Méthodes applicables à la propriété Items	
int Add (object item) ;	Ajoute un élément à la boîte de liste Renvoie la position de l'élément dans la liste
void Clear() ;	Vide le contenu de la boîte de liste
void Remove (object o)	Supprime un objet (o désigne en général une chaîne de caractères)
void Insert(int n, string)	Ajoute un élément à la n-ième position de la liste (si n=0, insertion en tête de liste)

VI.3.2 Les boîtes de liste avec cases à cocher

La classe **CheckedListBox** permet de créer une liste dont chaque élément est doté d'une case à cocher. Elle ne supporte pas la sélection multiple, mais elle permet de cocher plusieurs éléments à la fois. Les éléments cochés sont stockés dans la collection **CheckedItems**.

Etant dérivée de **ListBox**, les propriétés et méthodes de **ListBox** sont applicables à la classe **CheckedListBox** ; la classe présente toutefois des propriétés et méthodes propres.

Exemple : Ajout d'un élément dans la liste lstNoms et le cocher :

```
int n= lstNoms.Items.Add( "Raoul " ) ;  
lstNoms.SetItemChecked(n, true) ;
```

Propriétés les plus importantes des boîtes de liste avec cases à cocher

Noms	Description
Propriétés	
CheckOnClick	Indique si la case à cocher doit être basculée quand un élément est sélectionné
CheckedIndices	Collection des indices des éléments cochés.
CheckedItems	Collection des libellés des éléments cochés
Méthodes	
bool GetItemChecked(int n)	Renvoie true si l'élément en n-ième position est coché
void SetItemCheckState(int n, bool value)	Coche (true dans value) ou décoche l'élément en n-ième position

En parcourant la collection *Items*, l'état de sélection de chaque élément peut être déterminé par la méthode *GetItemCheckState()*.

Gérer les événements particuliers des zones de liste à cocher

En plus des événements classiques des zones de liste, les listes à cocher sont capables de générer un événement *ItemCheck* lorsque l'état d'une case à cocher associé à un élément est modifié.

Le gestionnaire de l'événement reçoit en paramètre un objet ***ItemCheckEventArgs***, qui possède trois propriétés en rapport avec cet événement :

- **Index** Indice de l'élément qui a généré l'événement
- **CurrentValue** état courant de la case à cocher (valeur *CheckState*)
- **NewValue** nouvel état de la case à cocher (valeur *CheckState*)

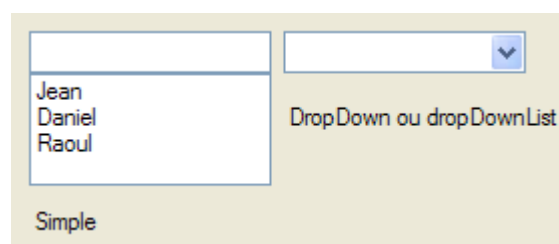
VI.3.3 Les boîtes de liste déroulantes (comboBox)

Le contrôle **ComboBox** est un contrôle hybride, combinant une **ListBox** et une **TextBox**.

Le contenu de la **TextBox** est accessible par la propriété **Text** du contrôle.

Selon le type de boîte combo (propriété **DropDownStyle**) :

- ✓ la liste est toujours affichée, ou non (Simple)
- ✓ la partie zone d'édition est réellement éditable (DropDown) ou non (dropDownList)



Lorsque la zone d'édition est éditable, la saisie du premier caractère provoque la recherche immédiate du premier élément commençant par ces caractères saisis.

La propriété *MaxDropDownItems* indique le nombre maximum d'éléments affichés en même temps dans la liste.

Etant dérivée de **ListBox**, les propriétés et méthodes de **ListBox** sont applicables à la classe **ComboBox** ; la classe présente toutefois des propriétés et méthodes propres.

Propriétés les plus importantes des boîtes de liste déroulantes

Nom	Description
DrawMode	Indique comment sont affichés les éléments de la combo
DropDownStyle	Type de Combo (DropDown / DropDownList / Simple)
DropDownWidth	Largeur de la boîte de liste
DroppedDown	Vaut true si la partie « boîte de liste » de la combo de style « dropdown » est affichée
MaxDropDownItems	Nombre d'éléments visibles dans la partie « boîte de liste » de la combo de style « dropdown » (entre 1 et 100)





Gérer les évènements

La plupart des évènements générés par les listes déroulantes sont amenés par les boîtes de liste, et par les zones de texte. Par exemple, *SelectedIndexChanged* sera généré lorsqu'un nouvel élément sera sélectionné, et *TextChanged*, lorsque le contenu de la zone de texte du contrôle sera modifié.

La classe `ComboBox` expose des évènements particuliers

:Noms	Description
DropDown	Affichage de la liste déroulante
SelectionChangeCommitted	Modification d'un élément de la liste déroulante

VI.4 LES COMPOSANTS DE DEFILEMENT

 HScrollBar  VScrollBar	Les barres de défilement(H scrollBar et VscrollBar)
 TrackBar	Les barres graduées
 ProgressBar	Les barres de progression

VI.4.1 Les barres de défilement (xxScrollBar)

On se sert souvent des barres de défilement pour incrémenter ou décrémenter une valeur :

Pour agir sur une barre de défilement, l'utilisateur peut :

1. cliquer sur l'une des flèches situées aux extrémités ce qui décrémente la propriété *Value* de la valeur de la propriété *SmallChange* (, , , avec le clavier)
2. cliquer sur la barre, mais en dehors des flèches ce décrémente la propriété *Value* de la valeur de la propriété *LargeChange* (*PageDown* et *PageUp* avec le clavier)
3. cliquer sur l'ascenseur et faire glisser la souris, bouton enfoncé

Propriétés remarquables

Noms	Description
LargeChange	Valeur d'incrémentation ou de décrémentation de la propriété <i>Value</i> lorsque l'utilisateur clique sur la barre, mais en dehors des flèches (par défaut égal à 10)
Maximum	Valeur maximale associée à la barre (100 par défaut)
Minimum	Valeur minimale associée à la barre (1 par défaut)
SmallChange	Valeur d'incrémentation ou de décrémentation de la propriété <i>Value</i> lorsque l'utilisateur clique sur l'une des flèches (par défaut égal à 1)
Value	Position de l'ascenseur, relativement aux propriétés <i>Minimum</i> et <i>Maximum</i> .

Gérer les évènements

Deux évènements sont prépondérants :

Noms	Description
ValueChanged	La propriété <i>Value</i> vient de changer, généralement parce que l'utilisateur a cliqué sur la barre de défilement. Pour savoir plus précisément quelle partie de barre a été activée, traitez l'évènement Scroll
Scroll	<p>L'utilisateur vient d'agir sur la barre. La méthode associée à cet évènement est</p> <pre>void hSB1_Scroll(object sender, ScrollEventArgs e)</pre> <p>pour une barre de nom hSB1 e contient deux informations :</p> <ul style="list-style-type: none">• <i>newValue</i> qui reflète la position de l'ascenseur (valeur que va prendre la propriété <i>Value</i>)• <i>type</i>, qui indique quelle partie de barre est concernée (<i>type</i> peut prendre l'une des valeurs de l'énumération <i>ScrollEventType</i>) : <ul style="list-style-type: none">⇒ SmallIncrement ou SmallDecrement: l'action est issue de 1⇒ LargeIncrement ou LargeDecrement: l'action est issue de 2⇒ First L'utilisateur a amené la barre au Minimum⇒ Last L'utilisateur a amené la barre au Maximum⇒ ThumbPosition L'utilisateur a relâché le bouton de la souris après avoir déplacé l'ascenseur⇒ ThumbTrack L'utilisateur déplace l'ascenseur⇒ Endscroll l'utilisateur termine une opération sur la barre

VI.4.2 Les barres graduées (TrackBar)

Il peut s'agir d'une barre horizontale ou d'une barre verticale (propriété *Orientation*). Les propriétés des barres de défilement (*Minimum*, *Maximum*, *Value*, *SmallChange*, *LargeChange*) se retrouvent dans les barres graduées, bien que la class **TrackBar** ne soit pas dérivée de **ScrollBar**.

On peut déplacer le curseur (et modifier ainsi la propriété *Value* de la barre) :

1. en cliquant sur le curseur et en le déplaçant
2. en frappant les touches de direction (ce qui incrémente ou décrémenté *Value* de *SmallChange* unités) ou les touches PageUp et PageDown (ce qui incrémente ou décrémenté *Value* de *LargeChange* unités).

Différentes possibilités sont offertes pour personnaliser les graduations (ticks) de la barre :

Propriétés remarquables

Noms	Description
TickFrequency	Indique la fréquence des graduations (une marque toutes les TickFrequency unités) (1 par défaut)
TickStyle	Indique comment les graduations sont placées par rapport à la barre concernée (<i>TickStyle</i> peut prendre l'une des valeurs de l'énumération <i>TickStyle</i>) <ul style="list-style-type: none">• Both : graduations de part et d'autre de la barre• BottomRight : graduations affichées au dessous (barre horizontale) ou à droite (barre verticale)• None : Aucune graduation• TopLeft : graduations affichées au dessus (barre horizontale) ou à gauche (barre verticale)

Le seul événement important est *Scroll*.

VI.4.3 Les barres de progression (ProgressBar)

Elles sont utilisées pour signaler l'état d'avancement d'une opération, et faire patienter l'utilisateur.





Les propriétés des barres de défilement (*Minimum*, *Maximum*, *Value*) se retrouve dans les barres de progression, bien que la class **ProgressBar** ne soit pas dérivée de **ScrollBar**.

Propriétés remarquables

Noms	Description
Maximum	Valeur maximale associée à la barre (100 par défaut)
Minimum	Valeur minimale associée à la barre (0 par défaut)
Step	Valeur d'incrément (10 par défaut) chaque fois que la méthode <i>PerformStep</i> est appelée
Style	Permet de définir (ou obtenir) le style de la progression
Value	Valeur courante associée à la barre de progression, relativement aux propriétés <i>Minimum</i> et <i>Maximum</i> .

Pour faire progresser une barre, exécutez la méthode **PerformStep**, qui ajoute à *Value* la valeur spécifiée dans la propriété **Step** (la modification de la propriété *Value* a le même effet), ce qui fait avancer la barre.

VI.5 LES CONTENEURS

 GroupBox	Les GroupBox
 Panel	Les Panel
 FlowLayoutPanel	Les FlowLayoutPanel
 TableLayoutPanel	Les TableLayoutPanel
 SplitContainer	Les SplitContainer
 TabControl	Les TabControl

Certains contrôles sont des conteneurs qui incorporent visuellement d'autres contrôles. Ils se trouvent dans la section conteneur de la boîte à outils.

VI.5.1 Contrôles **GroupBox** et **Panel**

Un conteneur du type **Panel** ou **GroupBox** contiendra plusieurs contrôles qui seront fonctionnellement regroupés dans sa collection **.Controls**.

Le contrôle **Panel** offre pratiquement les mêmes capacités que le contrôle **GroupBox**. Les différences sont les suivantes :

- Le contrôle **GroupBox** permet d'afficher un titre (propriété *Text*), pas le contrôle **Panel**
- Le contrôle **GroupBox** affiche toujours une bordure, alors que le **Panel** offre une propriété **BorderStyle**
- Le contrôle **Panel** dérive également de la classe **ScrollBarControl**. Il prend donc en charge les barres de défilement de façon automatique si sa propriété **AutoScroll** est à *true*.

Le **GroupBox** est souvent utilisé pour regrouper des boutons radios : les boutons radios sont exclusifs entre eux dans un même conteneur, il suffit de les déposer manuellement dans le conteneur pour assurer leur logique de fonctionnement

VI.5.2 Contrôles **FlowLayoutPanel** et **TableLayoutPanel**

Tous les deux héritant de **Panel**, le **FlowLayoutPanel** permet de présenter les contrôles à la manière d'un navigateur Web, disposés les uns à la suite des autres alors que le **TableLayoutPanel** les présente en les disposant à l'intérieur des cellules d'un tableau, chaque contrôle occupant une cellule.

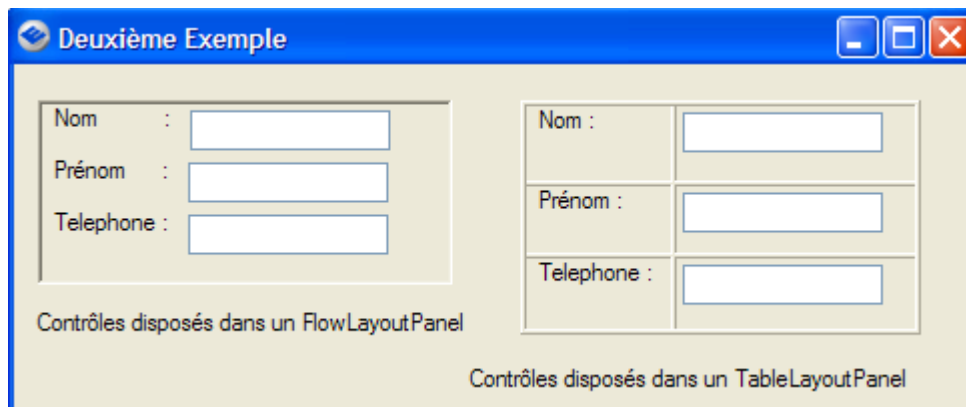
Le redimensionnement du contrôle **FlowLayoutPanel** s'accompagne d'une réorganisation des contrôles qu'il contient, afin qu'ils occupent tout l'espace disponible.

La propriété **FlowDirection** indique la direction dans laquelle les contrôles sont ajoutés au conteneur.

Le **TableLayoutPanel** est particulièrement utile en cas de création dynamique de contrôle (sa propriété **GrowStyle** permet de spécifier comment seront ajoutés les prochains contrôles en cas de tableau totalement plein).

La liste des tâches propose un éditeur pour dimensionner les cellules du tableau et donne la possibilité d'ajouter ou supprimer ligne et colonne

La propriété **CellBorderStyle** permet de spécifier l'apparence des bordures des cellules, un contrôle pouvant s'étendre sur plusieurs lignes ou plusieurs colonnes grâce à ses propriétés supplémentaires, **Rowspan** et **ColSpan**.



VI.5.3 Contrôle SplitContainer

Utilisez le contrôle **SplitContainer** pour diviser la zone d'affichage d'un feuille et autoriser l'utilisateur à redimensionner les contrôles ajoutés aux panneaux **SplitContainer**.

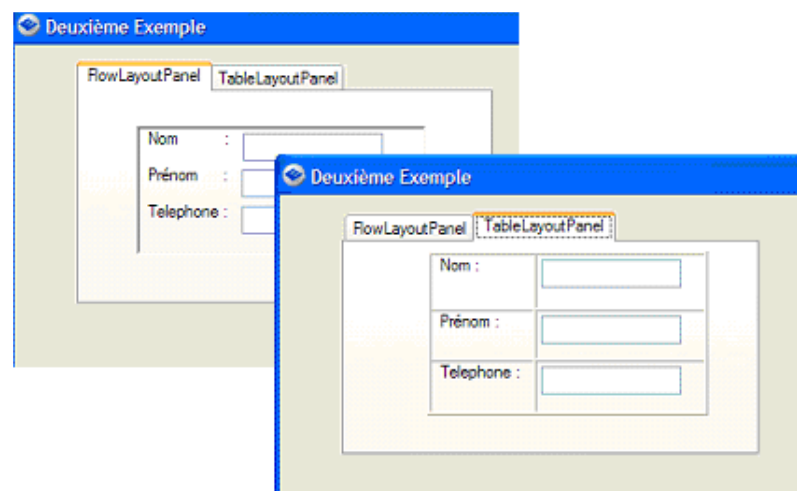
Lorsque l'utilisateur passe le pointeur de la souris sur le séparateur, le curseur se modifie pour indiquer que les contrôles situés à l'intérieur du contrôle **SplitContainer** peuvent être redimensionnés.

La propriété **SplitterDistance** spécifie l'emplacement du séparateur, en pixels, à partir du bord gauche ou supérieur, alors que **SplitterIncrement** spécifie de combien de pixels le séparateur se déplace à la fois et **Panel1MinSize** et **Panel2MinSize** représentant la taille minimum des 2 panneaux.

VI.5.4 Contrôle TabControl

Le contrôle **TabControl** contient des pages d'onglets qui sont représentées par des objets **TabPage**, ajoutés ou supprimés, en conception grâce aux options de la liste des tâches, ou par programme à travers la collection **TabPages**, modifiable directement dans la fenêtre des propriétés, grâce à un éditeur.

Lorsque l'onglet actif change, les événements suivants se produisent dans l'ordre suivant, Deselecting → Deselected → Selecting → Selected



VII LES MENUS, BARRES D'OUTILS ET D'ETAT

Les contrôles **ToolStrip**, **MenuStrip**, **StatusStrip** et **ContextMenuStrip** remplacent respectivement les contrôles **ToolBar**, **MainMenu**, **StatusBar** et **ContextMenu** qui sont encore présents pour préserver la compatibilité ascendante.

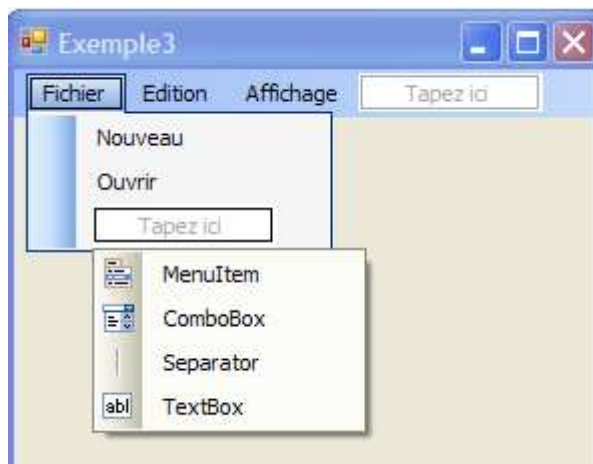
Ils sont regroupés dans l'onglet Menu et barres d'outils de la boîte à outils.



En plus d'un aspect visuel plus travaillé, de nouvelles fonctionnalités ont été ajoutées, telles que **l'organisation et l'alignement de texte et d'images.**

Il suffit de double-cliquer sur un de ces composants pour l'ajouter au formulaire.

VII.1 LE MENU DE L'APPLICATION (MENUSTRIP)

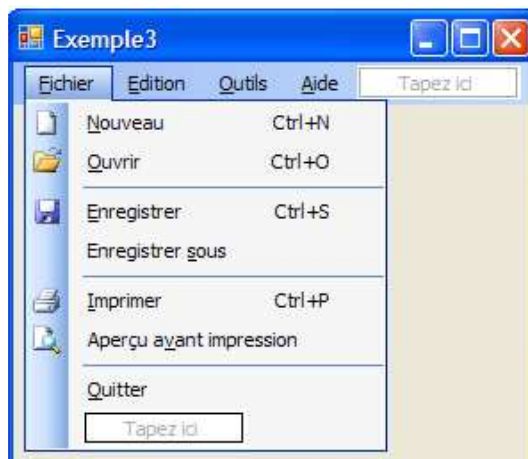


Un élément (instance de la classe **ToolStripMenuItem**) est créé en tapant son libellé directement dans l'éditeur de Menus (dans la zone *Tapez ici*).

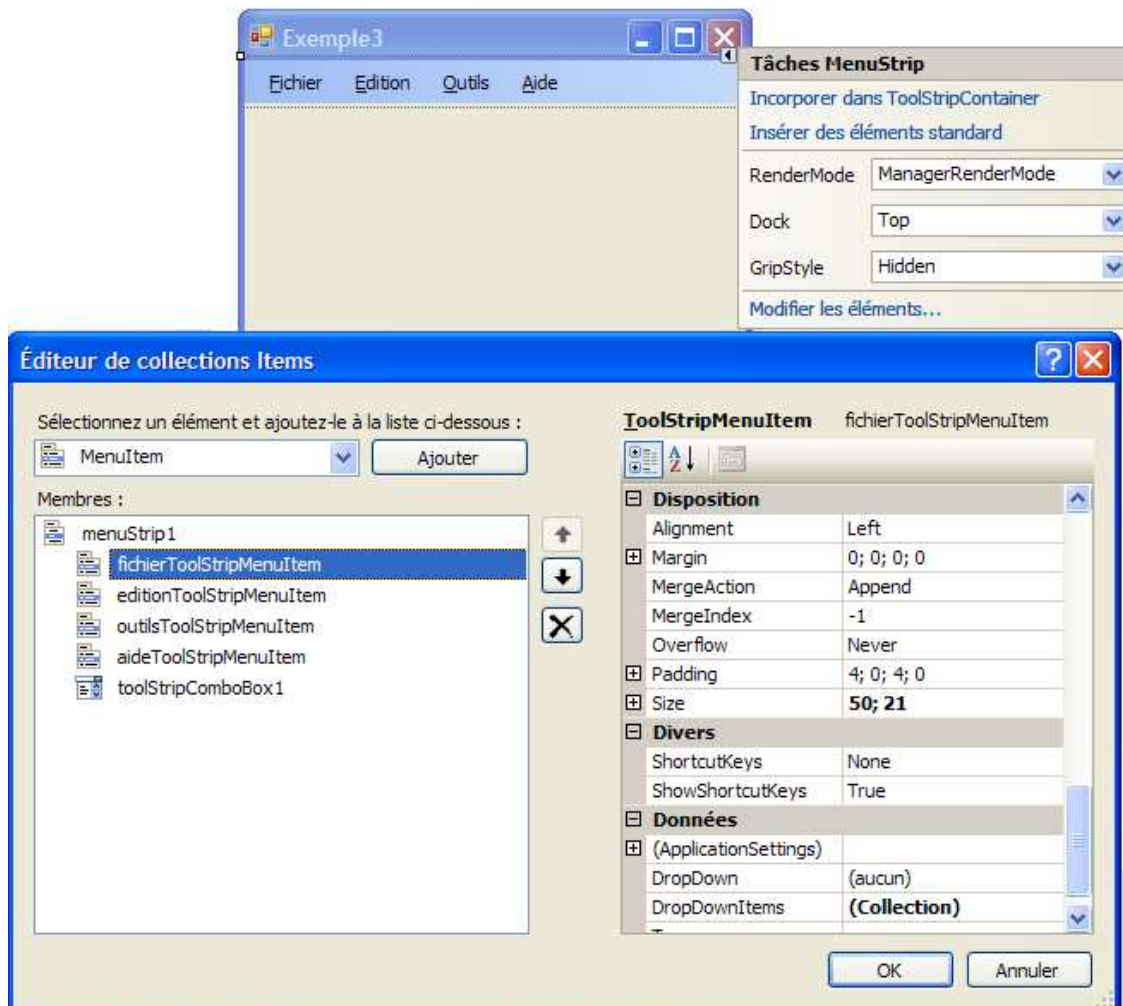
La nature de l'élément (**MenuItem**, **ComboBox**, **Separator**, **TextBox**) peut être choisie en cliquant sur l'élément

Le menu principal stocke sa collection d'élément dans une collection appelée *Items*, les sous-menus utilisent une collection **DropDownItems**.

L'option *Insérer des éléments standard* de la liste des tâches du menu principal permet d'insérer un menu standard.



L'option *Modifier les éléments* de la liste des tâches du menu principal permet de modifier les propriétés de chaque élément (faisable également dans la fenêtre de propriétés directement)



Chaque élément de menu possède les propriétés remarquables suivantes :

- **Text** Libellé de l'élément (& devant la lettre pour actionner le raccourci)
- **ShortcutKey** Touche de raccourci associée à l'élément
- **ShowShortcutKeys** Indique si la touche de raccourci est affichée sur l'élément
- **TextDirection** Permet de fixer la direction du libellé du menu
- **TextImageRelation** Spécifie la position du libellé par rapport à l'image
- **Image** Définit l'image qui sera affichée sur l'élément
- **Checked** Indique que l'élément est coché
- **Enabled** Indique que le menu est cliquable
- **Index** Position de l'élément dans le menu
- **Visible** Pour cacher l'élément

Gestion des évènements

Chaque élément de menu peut générer un évènement *Click*.

Le gestionnaire de l'évènement Click sera obtenu en double-cliquant soit sur le libellé du menu, soit sur l'évènement Click correspondant au menu à traiter dans la fenêtre des propriétés.

```
private void nouveauToolStripMenuItem_Click(object sender, EventArgs e)
{
}
```

On peut également ajouter dynamiquement des éléments de menus au menu existant :

1. Ajouter dans le programme une fonction de traitement, ou utilisez une fonction déjà créée.

```
private void mnu_Click(object sender, System.EventArgs e)
{
}
```

2. Pour ajouter l'élément Fermer en fin de menu Fichier, et faire traiter cet élément par la fonction btnQuitter_Click, il suffira de coder :

```
ToolStripMenuItem tsmi = new ToolStripMenuItem();
tsmi.Text= " Fermer ";
tsmi.Click += new System.EventHandler(this.btnQuitter_Click);
this.mnuFichier.DropDownItems.Add(tsmi);
```

Si la fonction de traitement traite plusieurs points de menus, on pourra détecter l'origine du click en écrivant :

```
ToolStripMenuItem tsmi = (ToolStripMenuItem)sender ;
if (tsmi.Text == "Fermer ") ...
```

Il est intéressant de noter qu'il n'existe aucun évènement pour signaler qu'un menu a été fermé;

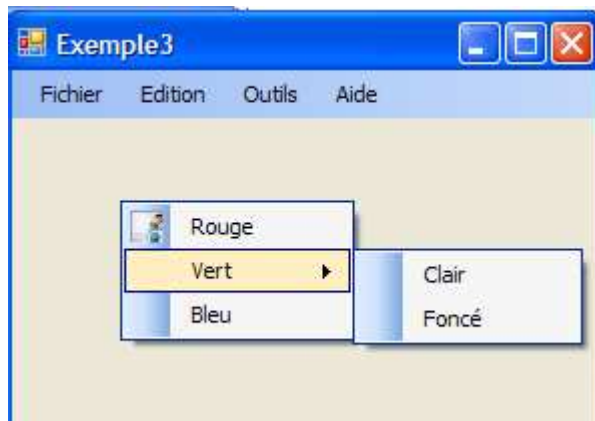
Pour cela, il faut passer par 2 évènements de la classe *Form* qui sont générés lors du premier affichage du menu et sa fermeture ;

- **MenuStart** le menu reçoit le focus pour la première fois
- **MenuComplete** le menu perd le focus

VII.2 LES MENUS CONTEXTUELS (CONTEXTMENUSTRIP)

Importé directement de la boîte à outils, c'est un objet de la classe **ContextMenuStrip**, dérivée de **MenuStrip**: il a toutes les caractéristiques d'un menu : il est créé et traité comme un menu, et permet d'associer un menu contextuel à n'importe quel contrôle.

La classe **ContextMenuStrip** représente des menus contextuels qui sont affichés lorsque l'utilisateur clique avec le bouton droit de la souris sur un contrôle ou une zone du formulaire. Les menus contextuels sont généralement utilisés pour combiner différents éléments de menu d'un **MenuStrip** d'un formulaire qui sont utiles à l'utilisateur compte tenu du contexte de l'application.



L'éditeur de collection Items dans la fenêtre de propriétés permet d'ajouter facilement tout élément à la barre d'outils et d'en modifier ses propriétés (accessible également dans la liste des tâches par l'option *Modifier les éléments*)

Les propriétés **ShowCheckMargin** et **ShowImageMargin** permettent de laisser apparaître de l'espace pour visualiser une image ou une coche lorsque l'option a été sélectionnée.

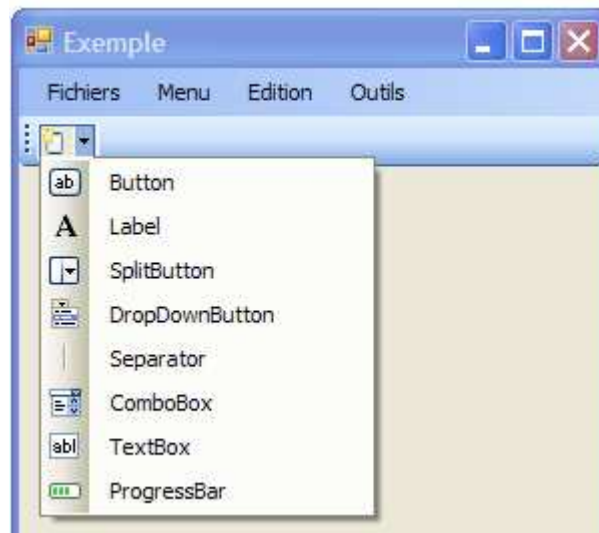
Pour afficher le menu contextuel à l'emplacement de la souris, lorsque l'utilisateur clique sur la feuille, on pourra coder :

```
private void Exemple3_MouseDown(object sender, MouseEventArgs e)
{
    contextMenuCouleur.Show(this, new Point(e.X, e.Y));
}
```

VII.3 LA BARRE D'OUTILS (TOOLSTRIP)

Le composant **ToolStrip** remplace et ajoute des fonctionnalités au contrôle **ToolBar** des versions antérieures, **ToolBar** est conservé pour la compatibilité descendante et l'utilisation éventuelle.

Le composant **ToolStrip** permet de créer rapidement une barre d'outils en incorporant des objets divers :



- Un **ToolStripButton** permettant d'incorporer texte et image. (remplace et étend le contrôle **ToolBarButton** des versions antérieures)
Utilisez les propriétés **ToolStripItem.ImageAlign** et **ToolStripItem.TextAlign** pour obtenir ou définir la position des images et du texte **ToolStripButton**
- Un **ToolStripLabel** représentant un **ToolStripItem** impossible à sélectionner qui restitue texte et images et peut afficher des liens hypertexte.
- Un **ToolStripSplitButton** représentant une combinaison d'un bouton standard à gauche et d'un bouton déroulant à droite ou inversement (si la valeur de **RightToLeft** est **Yes**).
- Un **ToolStripDropDownButton** représentant un contrôle qui, lorsque l'utilisateur a cliqué dessus, affiche un **ToolStripDropDown** associé à partir duquel l'utilisateur peut sélectionner un seul élément.
- Un **ToolStripSeparator** représentant une ligne utilisée pour grouper des éléments.
- Un **ToolStripComboBox** affichant un champ d'édition associé à un **ListBox**, permettant à l'utilisateur de sélectionner dans la liste ou d'entrer un nouveau texte.
- Un **ToolStripTextBox** représentant une zone de texte dans un **ToolStrip** qui permet à l'utilisateur d'entrer du texte
- Un **ToolStripProgressBar** représentant un contrôle de barre de progression Windows contenu dans **StatusStrip**.

L'éditeur de collection Items dans la fenêtre de propriétés permet d'ajouter facilement tout élément à la barre d'outils et d'en modifier ses propriétés (accessible également dans la liste des tâches par l'option *Modifier les éléments*)

Comme pour le **MenuStrip**, l'option *Insérer des éléments standard* de la liste des tâches de la barre d'outil permet de créer automatiquement les boutons d'une barre d'outils standard.



La barre d'outils standard générée automatiquement

Chaque élément de la barre peut générer un événement *Click*.

Le gestionnaire de l'évènement Click sera obtenu en double-cliquant soit sur le bouton de la barre, soit sur l'évènement Click correspondant au bouton à traiter dans la fenêtre des propriétés.

En cliquant sur l'icône Enregistrer, on obtient:

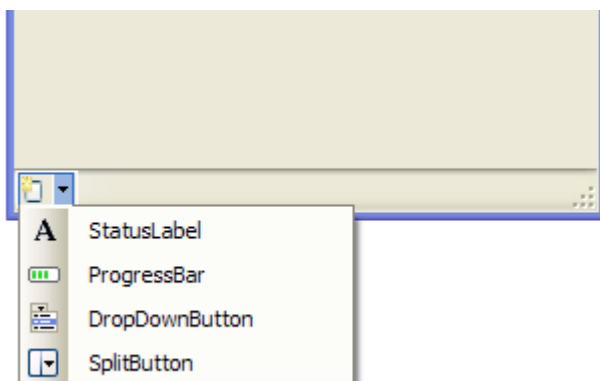
```
private void enregistrerToolStripButton_Click(object sender,
EventArgs e)
{
}
}
```

VII.4 LA BARRE D'OUTILS (STATUSSTRIP)

Le composant **StatusStrip** remplace et élargit le contrôle **StatusBar** des versions antérieures, **StatusBar** est conservé pour la compatibilité descendante et l'utilisation éventuelle

Le composant **StatusStrip** représente une barre de statut qui apparaît généralement en bas de la fenêtre. En règle générale, un contrôle **StatusStrip** est constitué d'objets **ToolStripStatusLabel**, dont chacun affiche du texte et/ou une icône.

Le **StatusStrip** peut également contenir des contrôles **ToolStripDropDownButton**, **ToolStripSplitButton** et **ToolStripProgressBar**.



nulaires

L'éditeur de collection Items dans la fenêtre de propriétés permet d'ajouter facilement tout élément à la barre de status et d'en modifier ses propriétés (accessible également dans la liste des tâches par l'option *Modifier les éléments*)

VII.5 LE TOOLSTRIPCONTAINER

Un **ToolStripContainer** a des panneaux sur ses chacun de ses quatre côtés pour le positionnement et le rafting des contrôles **ToolStrip**, **MenuStrip** et **StatusStrip**.

Plusieurs contrôles **ToolStrip** s'empilent verticalement si vous les placez dans le **ToolStripContainer** gauche ou droit.

Ils s'empilent horizontalement si vous les placez dans le **ToolStripContainer** supérieur ou inférieur.

VIII PLUSIEURS FEUILLES DANS L'APPLICATION

Plusieurs feuilles peuvent être ajoutées dans une application :
Ces feuilles peuvent être ajoutées par le menu contextuel "Ajouter un formulaire Windows" sur l'application dans la fenêtre explorateur du projet.

Elle se teste seule par la méthode Main qui la lance

```
Application.Run(new Form2());
```

VIII.1 DEFINITIONS

Les fenêtres peuvent être affichées de 2 façons: **modales** ou **non modales**.

Lorsqu'un formulaire fait l'objet d'un affichage **modal**, aucune entrée ne peut être effectuée (ni à l'aide du clavier, ni à l'aide de la souris) sauf sur les objets qu'il contient. Le programme doit masquer ou fermer le formulaire modal (généralement après une action de l'utilisateur) avant qu'une entrée puisse être effectuée dans un autre formulaire. Les formulaires à affichage modal sont généralement utilisés comme des boîtes de dialogue dans une application.

VIII.2 GESTION

Rappel : La classe Form est dérivée en une classe **Form2** qui permet de personnaliser notre fenêtre.

```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();
    }
    ...
}
```

Une feuille modale est affichée par la méthode **ShowDialog()** et ne pourra être fermée que par une action de l'utilisateur sur elle-même (par exemple, un click sur un bouton). Cette méthode peut rendre un résultat du type DialogResult.

```
private void btnOuvrir_Click(object sender, EventArgs e)
{
    Form2 frmDeux = new Form2();
    frmDeux.ShowDialog();
}
```

frmDeux est le nom de la variable de la classe Form2. Une feuille non modale est affichée par la méthode **Show()** ; si on désire fermer cette feuille depuis la feuille principale (par exemple, en cliquant sur un bouton), celle-ci devra avoir une référence sur cette feuille.

```

public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    Form2 frmDeux;
    private void btnOuvrir_Click(object sender, EventArgs e)
    {
        frmDeux = new Form2();
        frmDeux.Show();
    }
    private void btnFermer_Click(object sender, EventArgs e)
    {
        frmDeux.Close();
    }
}

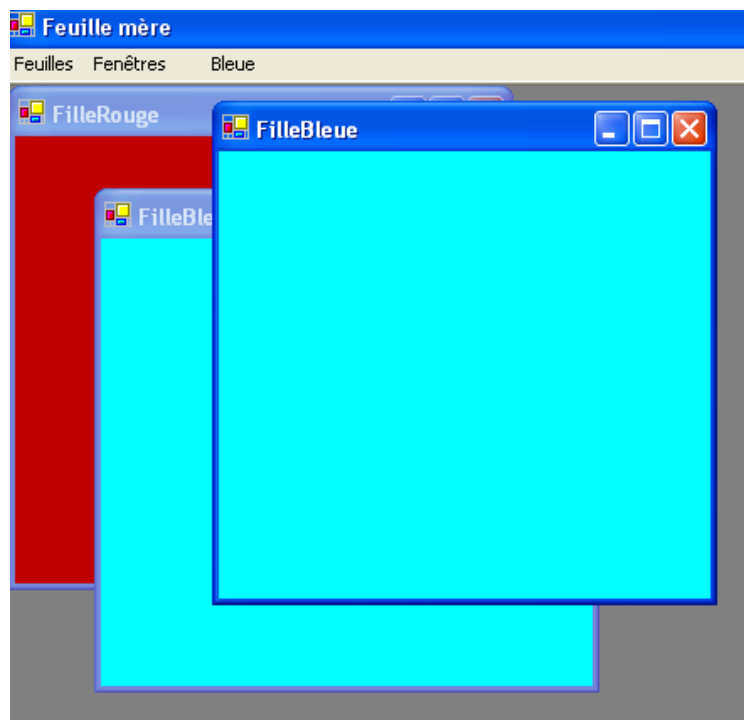
```

Référence

Nécessairement, on ne pourra cliquer sur Fermer avant d'avoir cliqué sur Ouvrir. Généralement, la fermeture de la feuille 2 sera gérée par une action utilisateur sur elle-même.

VIII.3 LES FEUILLES MDI

Elles sont composées d'une feuille principale appelée *feuille MDI*, il ne peut y en avoir qu'une par application et de *feuilles document* ou feuilles filles : la feuille mère et les feuilles filles sont non modales ; Word, mais aussi Visual Studio.Net présentent une telle interface.



Les fenêtres MDI présentent les principales caractéristiques suivantes :

- Toutes les fenêtres enfant sont affichées à l'intérieur de la fenêtre parent et ne peuvent être déplacées qu'à l'intérieur de cette fenêtre.
- Les fenêtres enfant peuvent être réduites en icône, mais les icônes sont affichées dans la fenêtre parent et pas dans la barre des tâches.
- Si une fenêtre enfant est maximisée, son titre est ajouté au titre de la fenêtre parent
- Le menu de la fenêtre enfant active s'insère dans celui de la fenêtre parent.

Créer une feuille mère

1. Créer un nouveau projet
2. Dans la fenêtre des propriétés de la feuille, positionner la propriété **IsMdiContainer** à **True**.
(Il est conseillé de positionner également la propriété **WindowState** à **Maximized**, ce qui permettra de manipuler les feuilles enfant plus facilement)
3. Ajouter un composant **MenuStrip** sur la feuille mère
(Nécessaire pour invoquer les feuilles enfants à partir de la feuille mère : par exemple, un menu Fichiers qui contiendrait deux sous-menus Nouveau et Fermer et un menu Fenêtres)
4. Positionner dans la propriété **MdiWindowListItem** du composant **MenuStrip** de la feuille mère le nom du menu dans lequel Windows se chargera de gérer la liste des fenêtres actives, ici le menu Fenêtres.
5. Ajouter une nouvelle feuille dans le projet, de nom frmEnfant.

Générer une feuille enfant depuis la feuille mère, sur le clic du menu Nouveau

```
private void mnuNouveau_Click(object sender, System.EventArgs e)
{
    frmEnfant newfeuille = new frmEnfant();
    newfeuille.MdiParent=this;
    newfeuille.Show();
}
```

6. Pour arranger la disposition des feuilles enfant créées par de multiples « clic » sur le menu Nouveau, on pourra utiliser la méthode `LayoutMdi` avec les quatre valeurs différentes de l'énumération `MdiLayout`
 - `ArrangeIcons` :
les icônes des fenêtres enfants sont réarrangées dans l'espace de fenêtre parent
 - `Cascade`
Réorganisation en cascade
 - `TileHorizontal`
en bandes horizontales
 - `TileVertical`
en bandes verticales

```
private void mnuCascade_Click(object sender, System.EventArgs e)
{
    this.LayoutMdi(MdiLayout.Cascade);
}
```

Gestion du menu de la feuille enfant

Les propriétés **AllowMerge** des composants **MainStrip** de la feuille mère et de la feuille fille doivent être positionnées à true(valeur par défaut).

Le menu de la fenêtre enfant active **se fond** dans celui de la feuille mère, suivant la valeur de la propriété **MergeAction** des objets **ToolStripMenuItem**. Le type de fusion d'un élément de menu indique la façon dont l'élément se comporte quand il a le même ordre de fusion qu'un autre élément de menu fusionné. Vous pouvez fusionner des menus pour créer un menu consolidé basé sur deux ou plusieurs menus existants

La propriété **MergeIndex** définit la position d'un élément fusionné.

MergeAction indique ce qui doit se passer lors de la fusion :

- **Append**
Ajoute l'élément à la fin de la collection
- **Insert**
Insère l'élément à la position indiquée par **MergeIndex**
- **MatchOnly**
La correspondance est nécessaire pour fusionner les menus
- **Replace**
les éléments du menu de la fenêtre enfant remplacent ceux de la fenêtre parent à la même position
- **Remove**
le menu n'est pas inclus dans le menu fusionné

VIII.4 COMMUNIQUER ENTRE FEUILLES

Un objet affiche une feuille à l'écran et souhaite échanger des données avec elle.

Les règles suivantes seront appliquées:

Si l'objet créateur a besoin de communiquer des informations en entrée à la feuille, il les lui donne:

- soit lors de sa construction en paramétrant le constructeur
- soit par le biais de méthodes de type **set** ou de propriétés au sens C# (données membres associées à des méthodes)

L'une ou l'autre de ces techniques permettent à la feuille de contrôler la qualité des données qu'elle reçoit

De même si la feuille doit renseigner l'objet en sortie, ce sera par le biais de méthodes de type **get** que l'objet pourra utiliser avant de la fermer.

Exemple:

On souhaite créer une instance de la classe Form2 à partir d'un bouton Ouvrir codé sur Form1, en lui spécifiant un titre à afficher dans la barre de fenêtre.

« Technique » du constructeur:

Tout ce que l'on souhaite rajouter à la construction d'une feuille sera fait par une procédure séparée de la procédure *InitializeComponent()* que l'on réserve à l'EDI Visual Studio.

Aussi, on créera une surcharge du constructeur qui recevra en paramètre, la variable *strTitre*, contenant le titre de la fenêtre à afficher. Ce constructeur appellera une méthode privée, *InitComponentBis()* par exemple, qui fera l'affectation du titre dans la propriété *Text* de la feuille et sera appelée dans le constructeur à la suite *InitializeComponent()*

Dans Form2 :

```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();
    }

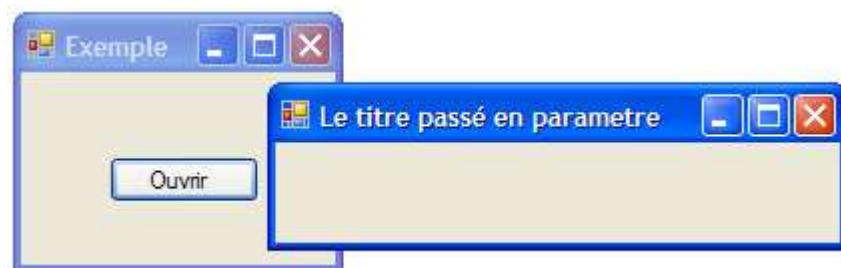
    public Form2(string strTitre)
    {
        InitializeComponent();
        InitializeComponent2(strTitre);
    }

    public void InitializeComponent2(string strTitre)
    {
        // affectation du titre dans la propriété Text
        this.Text = strTitre;
    }
}
```

Constructeur surchargé

Dans Form1 :

```
private void btnOuvrir_Click(object sender, EventArgs e)
{
    string strtitre = "Le titre passé en parametre";
    Form2 frmDeux = new Form2(strTitre);
    frmDeux.Show();
}
```



« Technique » des propriétés:

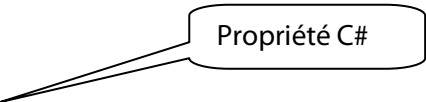
On va alors créer dans Form2, la propriété titre, au sens C# du terme, et l'affecter dans Form1 au moment de l'affichage de Form2. La propriété Text de Form2 sera positionnée sur l'évènement Load.

Dans Form2 :

```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();
    }

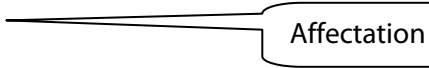
    private string titre;
    public string Titre
    {
        get { return titre; }
        set { titre = value; }
    }

    private void Form2_Load(object sender, EventArgs e)
    {
        this.Text = titre;
    }
}
```



Dans Form1 :

```
private void btnOuvrir_Click(object sender, EventArgs e)
{
    string strtitre = "Le titre passé en parametre";
    Form2 frmDeux = new Form2();
    frmDeux.Titre = strtitre;
    frmDeux.Show();
}
```



Etablissement référent

Marseille Saint Jérôme

Equipe de conception

Elisabeth Cattaneo

Remerciements :

Aux formateurs de St Brieuc et Clermont

Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.

« toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconques. »

Date de mise à jour 24/03/2009
afpa © Date de dépôt légal mars 09



afpa / Direction de l'Ingénierie 13 place du Général de Gaulle / 93108 Montreuil Cedex
association nationale pour la formation professionnelle des adultes
Ministère des Affaires sociales du Travail et de la Solidarité