

Utiliser les types et les structures du langage C#

Sommaire

1	INTRODUCTION.....	3
1.1	CONTEXTE FONCTIONNEL	3
1.2	CONTEXTE TECHNIQUE.....	4
2	DEFINIR L'INTERFACE UTILISATEUR DE L'APPLICATION	5
2.1	CREER LE PROJET ET LA FENETRE PRINCIPALE	5
2.2	DEFINIR LES MENUS	11
2.3	POSITIONNER LES CONTROLES DE DONNEES	32
2.4	LOCALISER L'APPLICATION.....	41
3	ALIMENTER LA GRILLE AVEC LE CONTENU D'UN FICHIER CSV	48
3.1	CREER ET UTILISER LA TABLE EN MEMOIRE	49
3.2	ALIMENTER LA TABLE EN MEMOIRE AVEC LE FICHIER CSV	58
3.3	FINALISEZ LE COMPORTEMENT DE LA FENETRE	80
4	POUR ALLER PLUS LOIN...	91
4.1	LES LIENS UTILES	91
4.2	IDEE/RESSOURCE.....	91

1 Introduction

Cet atelier s'inscrit dans le cadre du tutorial du coach C# dont l'objectif est la découverte et l'utilisation du langage C# 2.0 avec Visual Studio 2005 pour la construction d'applications avec une approche orientée objet.

Les exercices de l'atelier 2 se proposent de vous présenter l'utilisation des types et des structures de bases du langage C# en développant une application simple à base de formulaire Windows.

1.1 Contexte fonctionnel

Rappel du contexte fonctionnel du tutorial du coach C#

L'objectif du tutorial du C# est d'accompagner les développeurs à la prise en main du langage C# dans le cadre de projets de développement. Il s'adresse à tous ceux qui, connaissant un autre langage ou ayant une expérience de développement, souhaitent aller vers la programmation C#. Ce tutorial se concentre donc sur le langage, en l'abordant plus du point de vue des besoins des développeurs que de celui d'une liste exhaustive des caractéristiques techniques.



Le guide de référence complet du langage C# est disponible en ligne sur le site MSDN de Microsoft à l'adresse suivante :
[http://msdn2.microsoft.com/fr-fr/library/67ef8sbd\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/67ef8sbd(VS.80).aspx)

Contexte fonctionnel du deuxième atelier

Cet atelier décrit la grammaire du langage C# et les différents types que vous allez nécessairement utiliser dans vos tâches de programmation. De plus, est abordée la gestion des blocs de programmation, qui est importante tant du point de vue de la portée des variables que de celui de la gestion de la mémoire.

Vous allez donc développer une application à base de formulaire pour afficher une liste d'informations, lue dans un premier temps à partir d'un fichier au format « Texte délimité » ou CSV.

Cette application présente une fenêtre avec une grille de travail sur les données :

Menu de sauvegarde et d'ouverture des fichiers CSV

Grille de données

Barre de navigation

TextBox de résultat de calcul

Id	Entreprise	Contact	Titre	Adresse	Ville	Region
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Represent...	Obere Str. 57	Berlin	
ANATR	Ana Trujillo Empa...	Ana Trujillo	Owner	Ávda. de la Cons...	México D.F.	
ANTON	Antonio Moreno ...	Antonio More...	Owner	Maladeros2312	México D.F.	
AROUT	Around the Horn	Thomas Hardy	Sales Represent...	120 Hanover Sq	London	
BERGS	Berglunds snabb...	Christina Berglund	Owner Administrator	Berguvsvägen8	Luleå	
	Blauer See Delik...	Hanna Moos	Represent...	Forsterstr. 57	Mannheim	
	Blondesdél père...	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg	
	Bólido Comidas p...	Martín Sommer	Owner	C/ Araquil, 67	Madrid	
	Bon app'	Laurence Leblhan	Owner	12, rue des Bouc...	Marseille	
	Bottom-Dollar Ma...	Elizabeth Lincoln	Accounting	832 Sawassen Bl...	Tsawassen	BC
	B's Beverages	Victoria Ashworth	Sales Represe...	285 Baker Street	London	
	Cactus Comidas ...	Patricio Simpson	Sales Agent	Avda. de la Libertad	Buenos Aires	
	Centro comercial ...	Francisco Chang	Marketing Manager	Gran Vía, 1	México D.F.	
	CHOPPS	Yang Wang	Owner	Hauptstr. 31	Berlin	

Pour la programmation de cette partie, vous allez utiliser une approche « à l'ancienne », c'est-à-dire que vous allez utiliser des objets de données définis à la main en lisant un contenu au format texte-délimité. La programmation que nous allons utiliser confère à ce que l'on faisait dans un temps pas si ancien, lorsque nous programmions en client/serveur à deux niveaux.

1.2 Contexte technique

A la fin de cet atelier, vous saurez comment :

- Créer une application simple à base de formulaire ;
- Différencier les types par valeurs et les types par référence ;
- Définir la portée des variables ;
- Organiser votre code en bloc, afin de limiter les portées des variables ;
- Utiliser les principales instructions de bouclage et de tests ;
- Utiliser les principaux opérateurs ;
- Utiliser des objets fournis par le Framework

2 Définir l'interface utilisateur de l'application

Dans cet exercice, vous allez apprendre à :

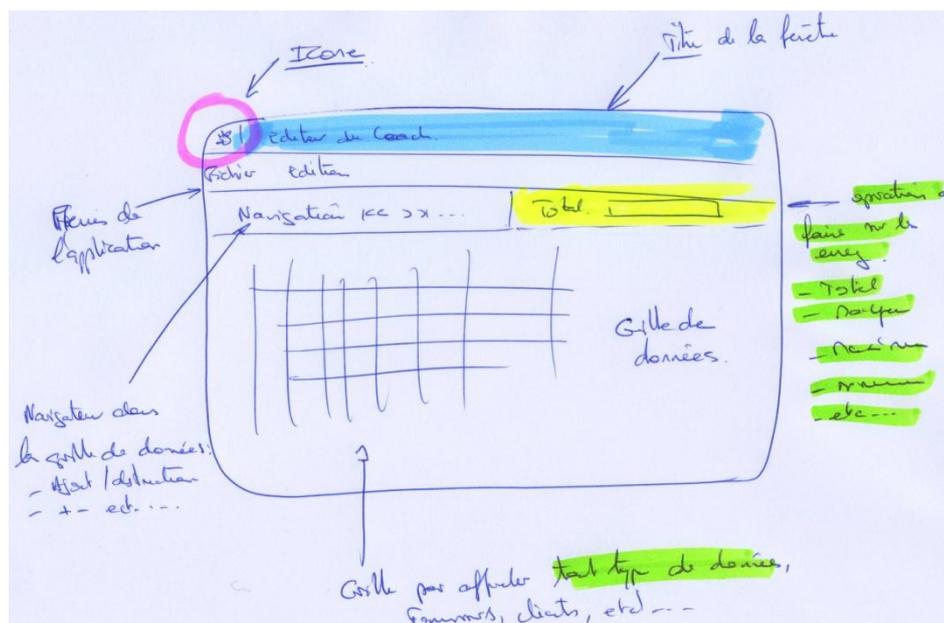
- Créer un projet de développement type formulaire Windows ;
- Positionner les différents contrôles graphiques de l'interface utilisateur ;
- Localiser l'application dans différentes langues ;

Objectif

L'objectif de cet exercice est de débiter le développement d'une application à base de formulaire en positionnant les éléments et contrôles de l'interface utilisateur, tout en abordant les concepts de base du langage C#.

Contexte fonctionnel

Vous êtes développeur, et vous avez à implémenter des écrans qui ont été décrits dans l'analyse fonctionnelle, ou mieux qui ont été conçus par un designer d'application à base de formulaire (on peut rêver un peu, non ?). Plus probablement, on vous a donné un bout de feuille issu d'une réunion et qui décrit ce qu'il faudrait faire :

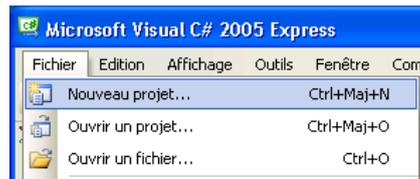


Votre première tâche est de réaliser cette interface utilisateur.

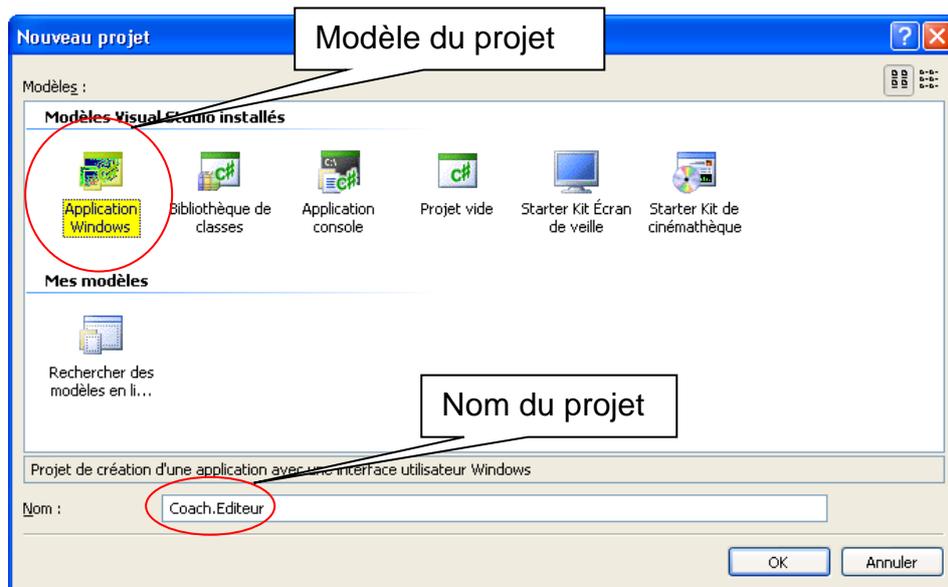
2.1 Créer le projet et la fenêtre principale

Déroulement de l'exercice :

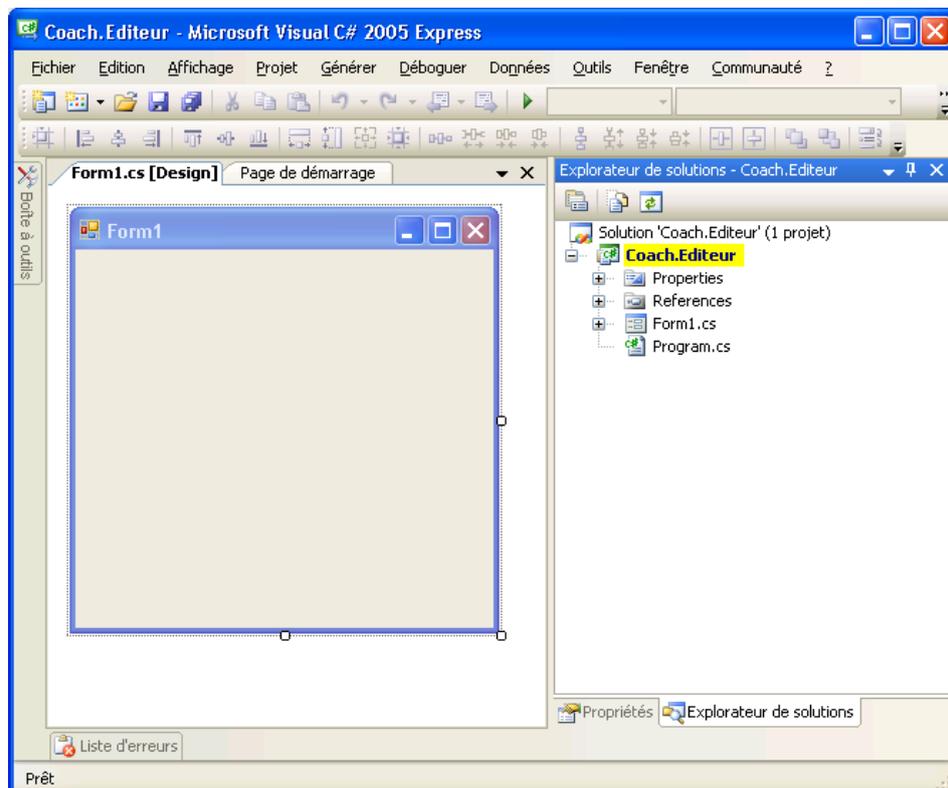
1. Créez la solution :
 - Lancez **Visual C#**
 - Dans Visual C# Express, cliquez le menu **Fichier > Nouveau projet ...**



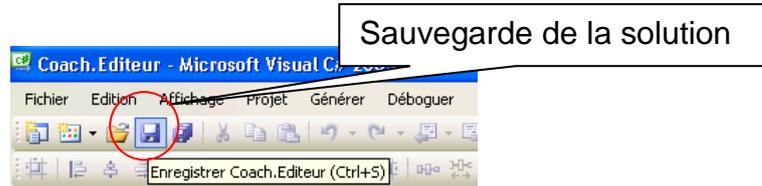
- La **boîte** de dialogue **Nouveau Projet** s'affiche. Dans cette **boîte** de dialogue, sélectionnez le modèle de projet **Application Windows** et indiquez un nom de projet **Coach.Editeur** ;



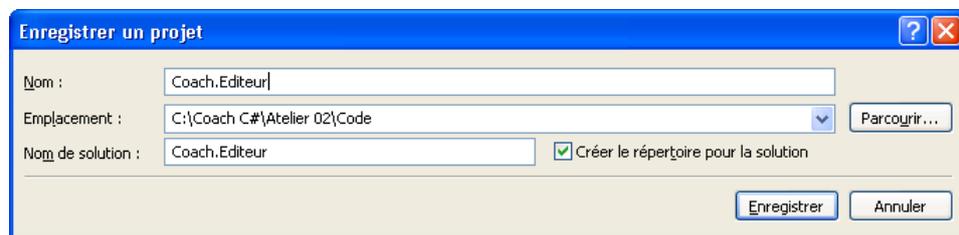
- Cliquez le bouton **OK**. Un projet avec une fenêtre vide s'affiche ;



- Si l'explorateur de solution n'est pas visible, pour l'afficher dans Visual C# Express cliquez le menu **Affichage > Explorateur de solutions** ;
- Dans la barre d'outils, cliquez la disquette pour sauvegarder votre solution ;



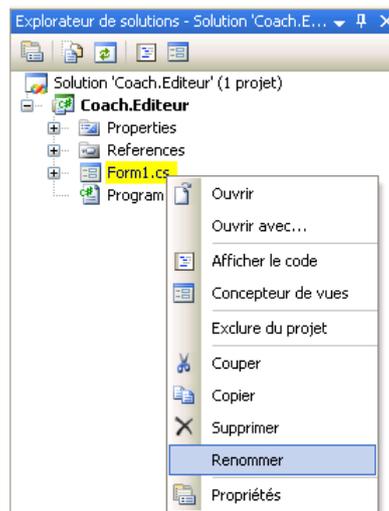
- Dans la boîte de dialogue **Enregistrer un projet**, indiquez votre répertoire de travail, en cochant la case à cocher **Créer le répertoire pour la solution** ;



- Cliquez sur **Enregistrer** ;

2. Renommez la fenêtre principale :

- Dans l'Explorateur de Solutions, sur le fichier **Form1.cs**, faites un **clic-droit** et sélectionnez le menu **Renommer** ;



- Changez le nom de **Form1.cs** en **Main.cs** ;
- Appuyez sur la touche **Entrée** ; La fenêtre suivante s'affiche :



Visual Studio vous propose ici de chercher automatiquement dans le projet en cours tous les endroits où il y a une référence au nom de la fenêtre **Form1** et de les remplacer par le nouveau nom donné **Main**. Il va s'occuper pour vous de renommer les variables et autres éléments, ce qui est très utile donc.

- Cliquez le bouton **Oui** ;



En fait, un formulaire Windows est avant tout une classe d'objet (d'ailleurs tout est classe d'objet dans .Net 😊). En **double cliquant** sur le nom de la fenêtre **Main.cs** dans l'**Explorateur de solutions**, vous verrez le designer de formulaire qui affiche l'interprétation du code de définition de la classe. Ce code peut aussi être affiché en **cliquant** dans la barre d'outil de l'explorateur de solutions l'icône ;

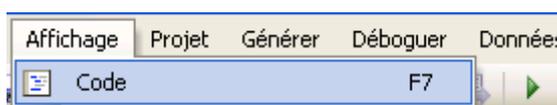


Il est possible d'afficher le code d'un formulaire en suivant d'autres « chemins » :

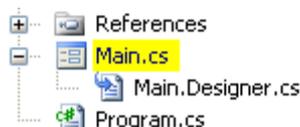
- a. Vous faites un **clic-droit sur le fichier** du formulaire et sélectionnez le menu **Afficher le code** ;



- b. Vous **sélectionnez le fichier** du formulaire et vous cliquez le menu Visual C# Express **Affichage > Code** ;



Un formulaire Windows est composé d'au moins deux fichiers dans Visual Studio. Ils sont visibles en cliquant sur l'icône précédent le nom du fichier.



Le fichier **Main.cs** contient tous les codes que vous allez programmer, et le fichier **Main.Designer.cs** contient tous les codes qui sont générés par Visual Studio. Le code généré est exactement celui que nous devrions développer à la main si nous voulions faire l'équivalent de ce que fera le générateur. Il n'y a ainsi aucun code caché et la totalité du code est accessible ! N'hésitez donc pas à aller y jeter un coup d'œil, mais ne vous lancez pas à le modifier si vous ne savez pas ce que vous faites, sauf

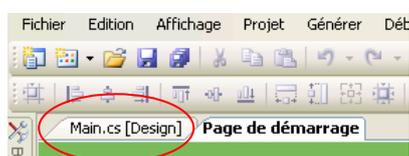
pour corriger des éventuelles erreurs de compilation liées à des destructions intempestives de contrôles par exemple.

3. Paramétrez la fenêtre principale :

- Affichez le formulaire **Main.cs** en **double-cliquant** sur son fichier dans l'explorateur de solutions ;

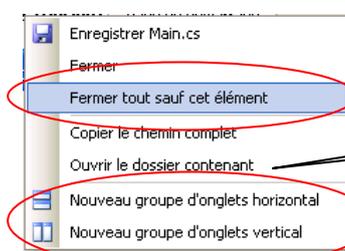


Il y a de fortes chances que le formulaire ait déjà été ouvert, et dans ce cas vous pouvez cliquer directement sur l'onglet **Main.cs [Design]** de la zone de travail ;



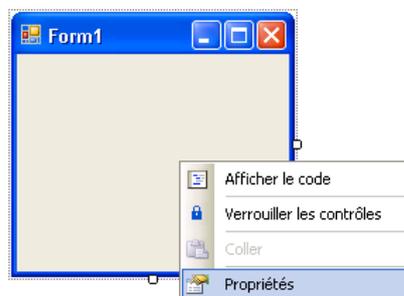
En faisant un clic-droit sur ces onglets, vous avez aussi un menu contextuel qui propose des options intéressantes comme fermer toutes les documents ouverts sauf celui en cours (utiles lorsqu'il y a tellement de fichiers ouverts qu'on est un peu perdu), ou comme ouvrir deux documents en visualisation simultanée sur la zone de travail (utile pour comparer et/ou copier-coller facilement du code).

Fermeture des documents

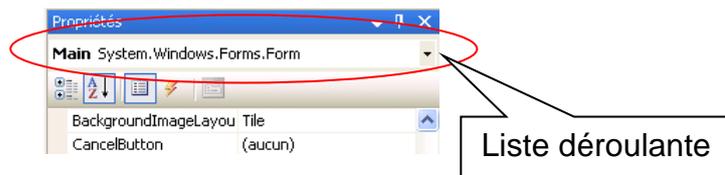


Visualisation simultanée

- Sur le formulaire, faites un **clic-droit** et sélectionnez le menu **Propriétés** ;



- Dans la fenêtre **Propriétés** qui est apparue sur la droite, vérifiez tout d'abord que vous êtes bien sur le bon objet, dont le nom et le type s'affiche dans la liste déroulante en haut ;



Si vous avez un formulaire complexe, cette liste déroulante est à utiliser pour sélectionner le bon élément. D'une manière générale, c'est une bonne pratique que de vérifier le nom du contrôle dont vous modifiez les propriétés.



La barre d'outils de la fenêtre de **Propriétés** possède les boutons suivant :

- Le bouton  affiche les éléments en les triant par catégorie ;
- Le bouton  affiche les éléments en les triant par nom ;
- Le bouton  définit que les éléments à afficher sont les propriétés de l'objet sélectionné ;
- Le bouton  définit que les éléments à afficher sont les événements de l'objet sélectionné – cette vue sert à ajouter facilement des méthodes de réponses aux événements (nous verrons cela plus tard) ;
- Le bouton  affiche une page de propriétés complémentaires de l'objet, s'il en existe une bien sûr.



Dans la fenêtre de **Propriétés**, les propriétés affichées en **gras** sont celles que vous avez modifiées. Les autres indiquent les valeurs par défaut. Pour chacune des propriétés dont la valeur indiquée est différente de la valeur par défaut, Visual C# Express va générer une (ou plusieurs) ligne(s) dans le fichier **Main.Designer.cs** ;

- Dans la fenêtre **Propriétés**, sélectionnez la propriété **Text** et tapez la valeur **Editeur du Coach C#** ; Vous remarquerez que le titre de la fenêtre a changé



Dans Microsoft .net, la propriété **Text** référence toujours ce qui est affiché par le contrôle à l'écran, même pour les labels ou les boutons.

- Dans la fenêtre **Propriétés**, sélectionnez la propriété **Size** (dimension), ouvrez la en cliquant sur , et indiquez la valeur **727** pour **Width** (largeur) et **427** pour **Height** (hauteur) ;

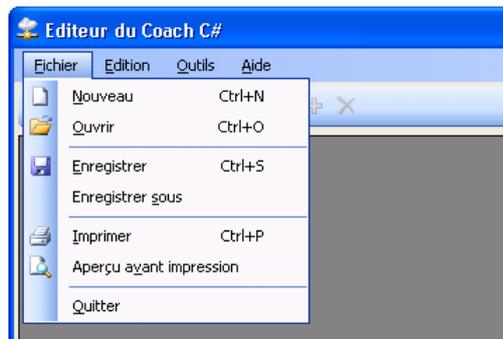
Size	727; 427
Width	727
Height	427

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder toutes les modifications réalisées ;

2.2 Définir les menus

L'objectif de cet exercice est d'ajouter à l'application deux types de menus :

- Une barre de menu standard qui s'affiche sous le titre de la fenêtre ;



- Un menu contextuel, qui va s'afficher quand l'utilisateur fera un clic droit sur un icône s'affichant dans la zone de notification (en bas à droite de l'écran)



Nous en profiterons pour programmer nos premières lignes de code dans cette application afin de connecter quelques sous menus (comme **Quitter** par exemple ☺).

Déroulement de l'exercice :

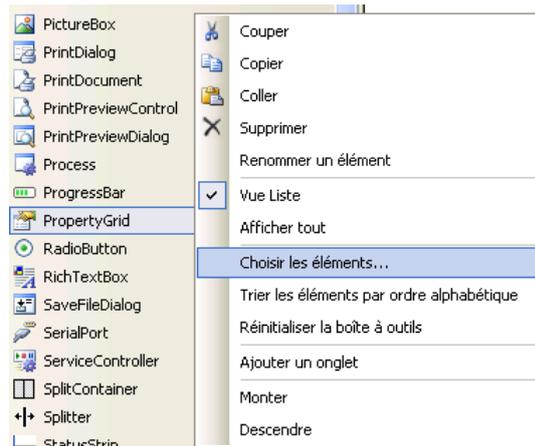
1. Ajoutez la barre de menu standard :



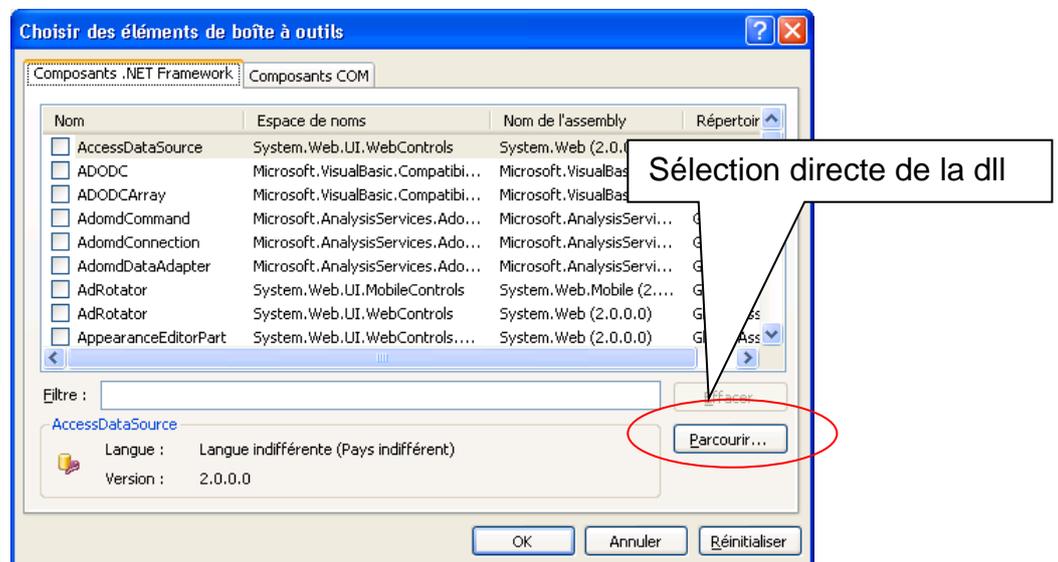
Visual Studio C# Express propose beaucoup de contrôles pour vous aider à construire vos écrans. Ces contrôles sont disponibles dans la **Boîte à outils** qui s'affiche en standard sur la gauche de votre écran. Si elle n'est pas visible, vous pouvez l'afficher en cliquant le menu **Affichage > Boîte à outils** dans Visual C# Express.



Vous pouvez ajouter (ou retirer) des contrôles à la liste qui est proposée. Pour ce faire, faites un **clic-droit** dans la **Boîte à outils**, et le menu suivant s'affiche :



Sélectionner le menu **Choisir les éléments ...** et la boîte de dialogue **Choisir des éléments de boîte à outils** vous aide à sélectionner les composants que vous souhaitez ;



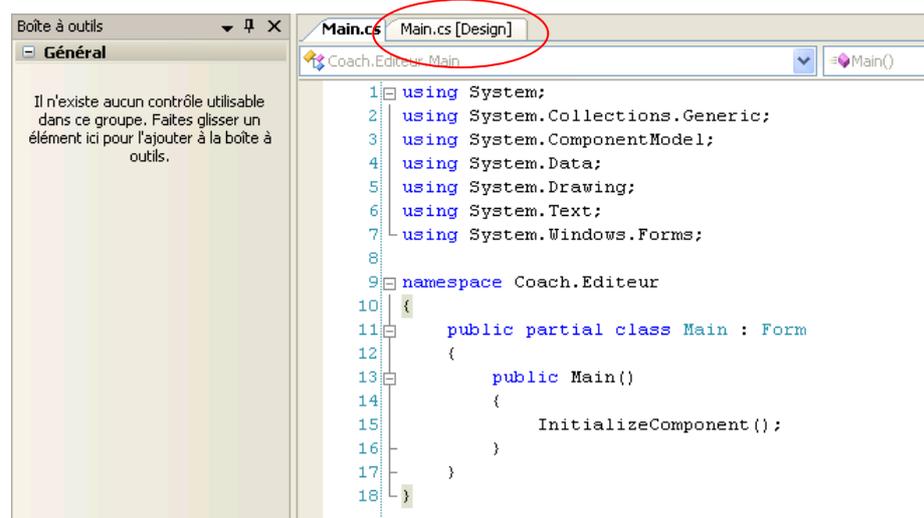
Le bouton **Parcourir** permet de sélectionner directement un assemblage (.dll) qui contiendrait des contrôles voulus.



Vous pouvez ajouter des composants issus de nombreuses sociétés tierces à Microsoft, ou issus de sites communautaires comme <http://www.codeplex.com/>. Sur la page d'accueil de ce dernier, sélectionnez la galerie de contrôles et vous avez une liste de contrôles et d'exemples téléchargeables.



La **Boîte à outils** affiche uniquement les composants qui sont disponibles compte tenu du contexte en cours. Si vous êtes dans une page de code, la **Boîte à outils** sera donc vide.

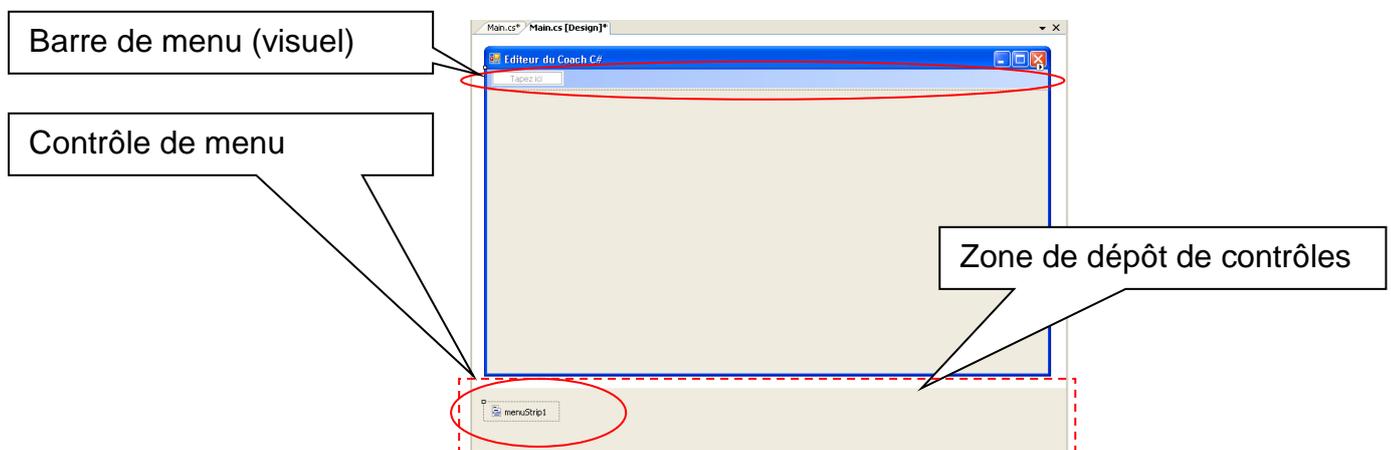


Pour afficher le contenu de la **Boîte à outils**, vérifiez donc bien que vous avez sélectionné le formulaire en mode **[Design]**.

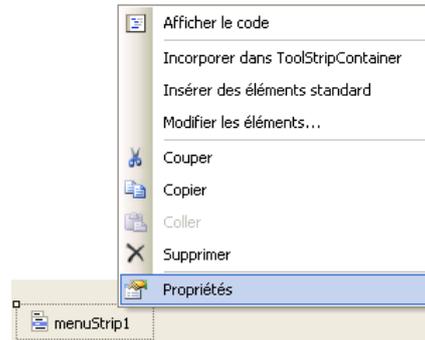
- Ouvrez le formulaire **Main.cs** en mode **[Design]** en double-cliquant sur le fichier **Main.cs** dans l'**Explorateur de solutions** ;
- Dans la boîte à outils, ouvrez l'onglet **Menus et barre d'outils** en cliquant sur **+** qui précède le nom de l'onglet ;



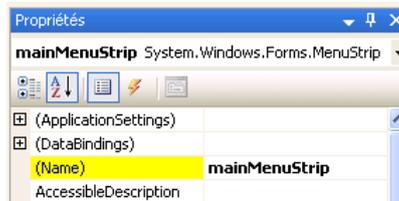
- Faites un glisser-déplacer du contrôle **MenuStrip** sur le formulaire ; Vous devez voir apparaître deux choses sur votre formulaire : une barre de menu vide sous le titre et un contrôle de menu (**menuStrip1**) dans une nouvelle zone de dépôt de contrôles en bas de la zone de travail ;



- Faites un **clic-droit** sur le contrôle **menuStrip1** et sélectionnez le menu **Propriétés** ;



- Dans les propriétés du contrôle, changez son nom par **mainMenuStrip** en changeant la valeur de la propriété **(Name)** ; Cette propriété apparaît parmi les premières si vous êtes en classement alphabétique (bouton  de la barre d'outils des propriétés) ;



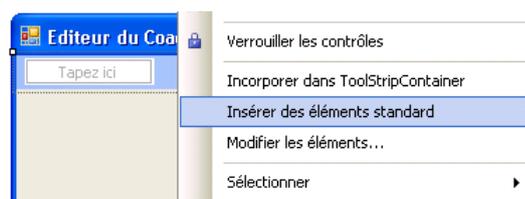
Le langage C# est sensible à la casse des mots. Autrement dit MainMenuStrip ne représente pas la même chose que mainMenuStrip. Pensez donc à systématiquement faire attention à la façon dont vous nommez vos variables et composants afin d'éviter les erreurs de compilation. Le meilleur moyen est de suivre un plan de nommage et de notation précis.



Pour tout savoir sur les bonnes pratiques et les conventions concernant le nommage, voici un lien très intéressant :

[http://msdn2.microsoft.com/fr-fr/library/ms229002\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/ms229002(VS.80).aspx)

- Dans la fenêtre **Main.cs** en mode **[Design]**, faites un clic-droit sur la barre de menu que vous venez d'ajouter ;
- Sélectionnez le menu **Insérer des éléments standard** ; Magique Non ? Bon seuls les menus ont été ajoutés (c'est déjà bien), et il reste à ajouter le code ;



- Dans la fenêtre **Main.cs** en mode **[Design]**, faites un double-clic sur le menu **Fichier > Quitter** ;



- Une méthode `quitterToolStripMenuItem_Click` de réponse au menu **Quitter** a été ajoutée au code du formulaire **Main.cs** ;

Code C#

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

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

- Dans la méthode `quitterToolStripMenuItem_Click`, ajoutez le code de fermeture du formulaire `this.Close()` ;

Code C#

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

    private void quitterToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // Fermeture du formulaire
        this.Close();
    }
}
```

- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour sauvegarder toutes les modifications réalisées ;

- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; **L'Editeur du coach C#** se lance, et en cliquant son menu **Fichier > Quitter**, il se ferme ! Félicitation, vous venez de réussir la première ligne de code 😊 (bon certes, nous n'avons pas encore affiché « Hello World », mais c'est un bon début).



C'est quoi **this** ?

Le mot clé **this** référence l'instance courante d'un objet. Dans notre cas il s'agit donc de l'instance de la fenêtre **Main** qui sera créée lors de l'exécution du programme.



Pour tout savoir sur le mot clé **this**, veuillez aller sur le lien :

[http://msdn2.microsoft.com/fr-fr/library/dk1507sz\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/dk1507sz(VS.80).aspx)



Mais comment fait le système pour retrouver la bonne méthode à exécuter quand on clique sur **Quitter** ?

Ce n'est pas le nom de la méthode qui fait foi. En fait, la fonction de prise en charge de l'événement (ou la méthode de réponse à l'événement, qui se dit « *Event Handler* » en anglais) a été dans notre cas automatiquement paramétrée par Visual C# Express dans le fichier **Main.Designer.cs** lorsque nous avons fait un double clic sur le menu **Quitter**. Nous allons aller voir la définition (faites attention de ne rien modifier 😊) :

- Ouvrez le fichier **Main.Designer.cs** en double cliquant dessus dans l'**Explorateur de solutions** ;
- Dans ce fichier, cherchez la partie qui est automatiquement générée par Visual C# Express ;

```
22:|
23:|
318:|
```

Code généré par le Concepteur Windows Form

- Cliquez sur le signe  pour visualiser les lignes de code ;
- Avec la barre de défilement vertical, survolez le code et localisez la définition du menu **quitterToolStripMenuItem** ;

C'est ici qu'est définie la relation entre l'événement et la fonction de prise charge.

```
// quitterToolStripMenuItem
//
this.quitterToolStripMenuItem.Name = "quitterToolStripMenuItem";
this.quitterToolStripMenuItem.Size = new System.Drawing.Size(203, 22);
this.quitterToolStripMenuItem.Text = "&Quitter";
this.quitterToolStripMenuItem.Click += new System.EventHandler(this.quitterToolStripMenuItem_Click);
```

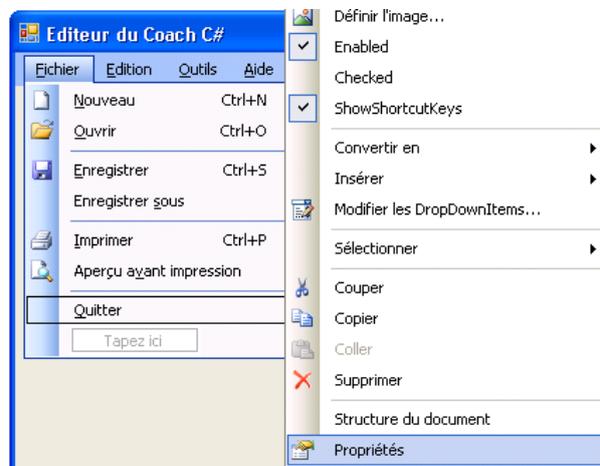
Nous reviendrons plus dans le détail sur ces fonctions de prise en charge des événements (ou « *Event Handler* »), notamment sur le formalisme à utiliser. Ce qu'il faut retenir est qu'il n'y a aucun code caché, et que nous pourrions si nous le désirerions nommer comme nous le voudrions la fonction de réponse à l'événement (avec un beau et original nom comme « toto » par exemple ! mais cela ne facilite pas la maintenance par la suite).

- Fermez le fichier de code **Main.Designer.cs** en cliquant sur le bouton  de l'onglet du fichier ;

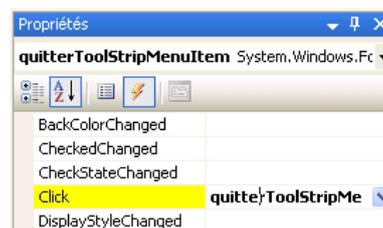


Comment visualiser l'ensemble des événements disponibles sur le menu **Quitter** ?

- Dans la fenêtre **Main.cs** en mode **[Design]**, faites un **clic-droit** sur le menu **Quitter** et sélectionnez le menu **Propriétés** ;



- Dans les **Propriétés** du menu **Quitter**, cliquez le bouton de la barre d'outils pour afficher l'ensemble des événements ; Vous devez voir que seul l'événement **Click** possède une méthode de réponse ;



- Par la suite, pensez à cliquer sur le bouton de la barre d'outils des **Propriétés** pour revenir à la liste des propriétés ;



Pour ajouter une fonction de réponse à un événement, il suffit d'aller visualiser la liste des événements disponibles sur le contrôle sélectionné (cf. ci-dessus) et de double-cliquer sur le nom de l'événement. Cela va ajouter automatiquement la fonction dans votre fichier de code (par exemple **Main.cs**), et le paramétrage de l'événement dans le fichier utilisé par le designer de Visual C# Express (par exemple **Main.Designer.cs**). Au lieu de double-cliquer, vous pourriez aussi indiquer un nom quelconque pour la fonction de prise en charge (« toto » par exemple), et au moment où vous allez taper **Entrée**, Visual C# Express va tout créer pour vous.

Si vous détruisez la fonction de réponse de votre fichier de code, il faudra aussi penser à retirer le paramétrage de l'événement dans le fichier utilisé par le designer, Visual C# Express ne pouvant le faire pour vous car il ne sait pas si la destruction (ou le changement de nom) est consécutif à une erreur ou à un choix décidé du programmeur. De toute manière, si vous ne le faites pas, vous obtiendrez une erreur de compilation (et oui, le compilateur C# vérifie un maximum de chose).

2. Ajouter un menu contextuel :



Nous allons maintenant ajouter un menu contextuel, qui va s'afficher quand l'utilisateur fera un clic droit sur un icône s'affichant dans la zone de notification (en bas à droite de l'écran)

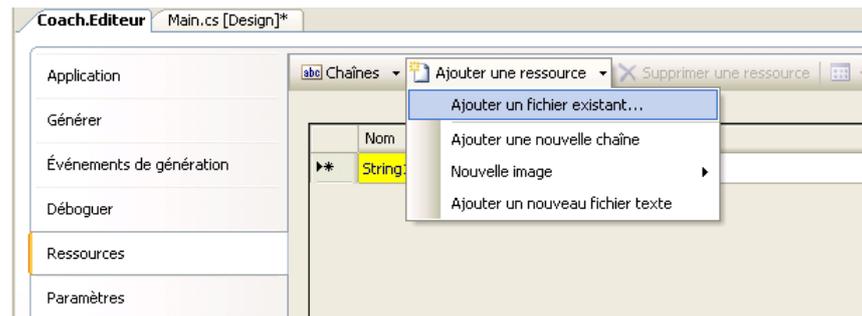


Pour ce faire nous devons suivre quatre étapes :

- Ajouter un icône s'affichant lors de l'exécution dans la zone de notification, et le paramétrer ;
 - Ajouter un menu contextuel, et le paramétrer ;
 - Lier l'icône de notification et le menu contextuel ;
 - Ajouter le code correspondant aux options du menu contextuel.
- Ouvrez le formulaire **Main.cs** en mode **[Design]** en double-cliquant sur le fichier **Main.cs** dans l'**Explorateur de solutions** ;
 - Dans la boîte à outils, ouvrez l'onglet **Contrôles communs** en cliquant sur  qui précède le nom de l'objet ;
 - Faites un **glisser-déplacer** du contrôle **NotifyIcon** sur le formulaire ; Vous devez voir apparaître le contrôle **notifyIcon1** dans la zone de dépôt de contrôles en bas de la zone de travail ;

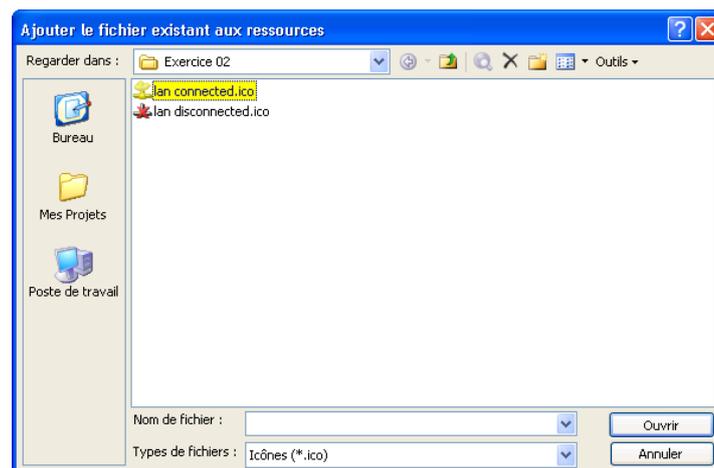


- Faites un **clic-droit** sur le contrôle **notifyIcon1** dans la zone de dépôt de contrôles, et sélectionnez le menu **Propriétés** ;
- Dans les propriétés du contrôle, changez son nom par **mainNotifyIcon** en changeant la valeur de la propriété **(Name)** ; Cette propriété apparaît parmi les premières si vous êtes en classement alphabétique (bouton  de la barre d'outils des propriétés) ;
- Dans les propriétés du contrôle, changez la propriété **Text** avec **Editeur du Coach C#** ; Ce texte apparaîtra en aide rapide (« tooltip ») lorsque le pointeur de souris sera au dessus de l'icône ;
- Dans l'**Explorateur de solutions**, faites un clic-droit sur le projet **Coach.Editeur** et sélectionnez le menu **Propriétés** ;
- Sélectionnez l'onglet **Ressources** ;
- Sur l'écran de gestion des ressources, cliquez le menu **Ajouter une ressource > ajouter un fichier existant...** ;

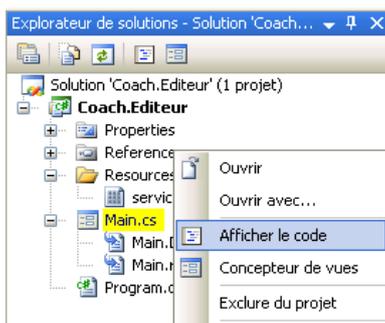


Par rapport au premier atelier, nous n'avons pas à créer les ressources associées au projet car elles ont été automatiquement créées par le modèle que nous avons utilisé pour créer le projet.

- Dans la boîte de dialogue **Ajouter un fichier existant aux ressources**, naviguez jusqu'au répertoire des fichiers utiles de l'exercice 01 de l'atelier 02 (par exemple **C:\Coach C#\Atelier 02\Fichiers Utiles\Exercice 01**) ;
- Dans la boîte de dialogue **Ajouter le fichier existant aux ressources**, sélectionnez le fichier **lan_connected.ico** et cliquez le bouton **Ouvrir** ;



- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Fermez l'onglet des ressources du **Coach.Editeur** en cliquant sur son bouton de fermeture  ;
- Dans l'explorateur de solutions, faites un **clic-droit** sur le fichier **Main.cs** et sélectionnez le menu **Afficher le code** ;



- **Localisez** dans la définition de la classe Main la méthode `public Main()` ; Il s'agit du constructeur de la classe ;

Code C#

```
namespace Coach.Editeur
{
    public partial class Main : Form
    {
        public Main()
        {
            InitializeComponent();
        }

        (...)
    }
}
```

Constructeur de la classe



C'est quoi un constructeur ?

Le constructeur est la méthode membre d'une classe que vous allez appeler à chaque fois que vous instanciez un nouvel objet :

Type de variable

Nom de variable

Méthode membre appelée pour l'instanciation

```
Main main = new Main();
```

L'objectif du constructeur est de créer convenablement une instance d'objet, en le configurant dans un état valide. En C#, le constructeur répond aux caractéristiques suivantes :

- Le nom du constructeur est rigoureusement le même que le nom de la classe ;
- Le constructeur n'a aucun type de retour spécifié (pas même `void`).



Comme toute méthode membre, il est possible de passer des paramètres au(x) constructeur(s) d'une classe. Il est ainsi possible de définir plusieurs façons convenables de créer une instance d'objet. C'est notamment visible dans l'IntelliSense qui vous propose de sélectionner un des différents constructeurs disponible, le cas échéant :

Il y a ici trois façons de créer cet objet

```
List<Main> = new List<Main>()
// 1 sur 3 List<Main>.List()
Initialise une nouvelle instance de la classe System.Collections.Generic.List<T> qui est vide et possède la capacité initiale par défaut.
```



Le constructeur qui ne possède aucun paramètre passé est appelé le constructeur par défaut.

- Dans le constructeur de **Main**, ajoutez le code pour initialiser l'icône du contrôle **mainNotifyIcon** avec celui chargé en ressource ; Bon, c'est le même code que celui fait à l'atelier N°1 (exercice 4.2.2), mais on va se répéter un peu :

Code C#

```
namespace Coach.Editeur
{
    public partial class Main : Form
    {
        public Main()
        {
            InitializeComponent();
            // Assignment de l'icone de mainNotifyIcon
            mainNotifyIcon.Icon = Properties.Resources.lan_connected;
        }

        (...)
    }
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Revenez sur le formulaire **Main.cs** en mode **[Design]** en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail ;
- Dans la boîte à outils, ouvrez l'onglet **Menus et barre d'outils** en cliquant sur  qui précède le nom de l'onglet ;

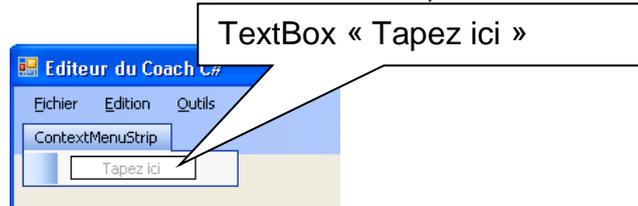


- Faites un **glisser-déplacer** du contrôle **ContextMenuStrip** sur le formulaire ; Vous devez voir apparaître deux choses sur votre formulaire : une barre de menu contextuel vide sous le menu standard et un contrôle de menu (**contextMenuStrip1**) dans la zone de dépôt de contrôles en bas de la zone de travail ;



Un menu contextuel ne s'affichant pas hors de son contexte (c'est une Lapalissade), dès que le contrôle du menu contextuel n'est plus sélectionné dans la zone de dépôt de contrôles, il disparaît de l'écran. Pour l'afficher de nouveau, il vous suffit de sélectionner le contrôle dans la zone de dépôt de contrôles.

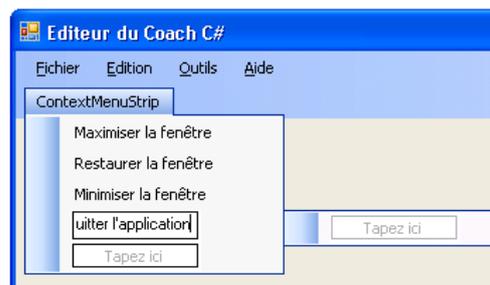
- Faites un **clic-droit** sur le contrôle **contextMenuStrip1** dans la zone de dépôt de contrôles, et sélectionnez le menu **Propriétés** ;
- Dans les propriétés du contrôle, changez son nom par **mainNotificationIconContextMenuStrip** en changeant la valeur de la propriété **(Name)** ; Cette propriété apparaît parmi les premières si vous êtes en classement alphabétique (bouton  de la barre d'outils des propriétés) ;
- Sur la fenêtre, juste en dessous des menus standards, **cliquez** sur la boîte de texte « **Tapez ici** » qui apparaît dans le **ContextMenuStrip** juste en dessous des menus standards du formulaire ;



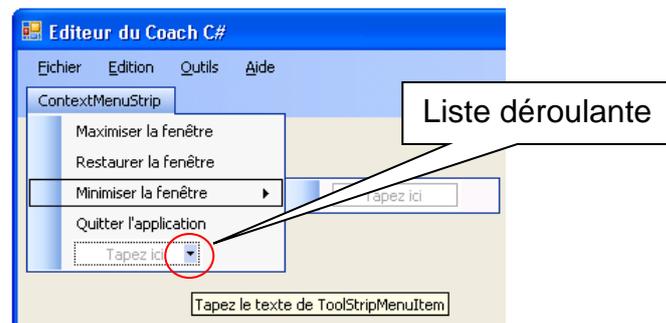
- Saisissez dans la boîte de texte **Maximiser la fenêtre** ;



- Recommencez l'opération avec les options **Restaurer la fenêtre**, **Minimiser la fenêtre**, et **Quitter l'application** ;

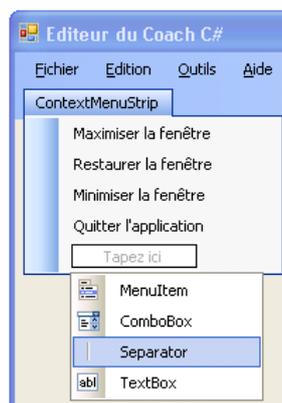


- Afin de sortir du mode de saisie des options du menu, faites un **simple clic** sur une des options déjà saisie ;
- Déplacez le pointeur de souris sur la dernière boîte de texte « **Tapez ici** », et vous devez voir apparaître la flèche d'une liste déroulante sur la droite du contrôle ;

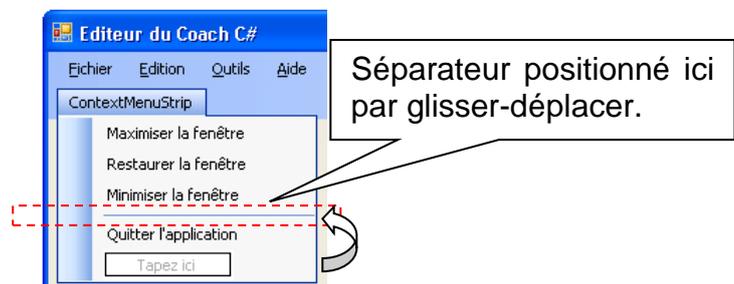


La liste déroulante vous aide à insérer un autre type de ligne de menu : le **MenuItem** correspond à une ligne de menu similaire à celles que vous venez de saisir, une **ComboBox** ajoute une liste déroulante, et le **TextBox** vous permet de saisir directement une valeur de menu lors de l'exécution de l'application. Il reste le **Separator**, que nous allons utiliser, et qui ajoute simplement une ligne grise horizontale dans la liste des menus.

- Cliquez sur **Separator** pour insérer une ligne de séparation dans la liste des menus ;

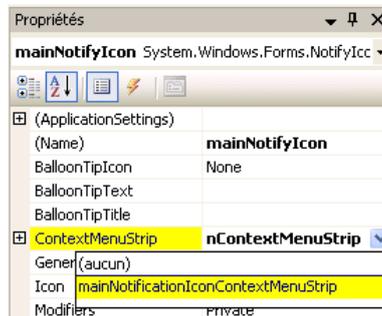


- Faites un **glisser-déplacer** de la ligne de séparation afin de la positionner avant le menu **Quitter l'application** ;



- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Faites un **clic-droit** sur le contrôle **mainNotifyIcon** dans la zone de dépôt de contrôles, et sélectionnez le menu **Propriétés** ;

- Dans les propriétés du contrôle, changez la valeur de **ContextMenuStrip** en utilisant la liste déroulante proposée, et sélectionnez **mainNotificationIconContextMenuStrip** ;



- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; **L'Editeur du coach C#** se lance, et vous devez voir votre icône dans la zone de notification en bas à droite de votre écran. Si vous faites maintenant un clic-droit sur votre icône, vous devez voir apparaître le menu contextuel. Reste donc à programmer les options du menu.

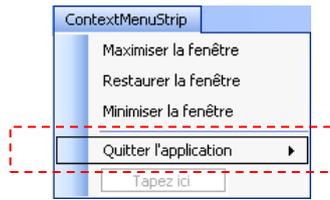


Bon, si vous avez des erreurs de compilation, c'est probablement lié à la casse car C# est case-sensitive, ou encore à un ' ; ' qui manque en fin de ligne. Servez-vous des messages d'erreurs de compilation renvoyés par Visual C# Express pour trouver les bonnes corrections.



Nous allons maintenant développer les méthodes de réponse aux événements **Click** de chacun des menus.

- Arrêtez le mode de débogage en cliquant le bouton  dans la barre d'outils **Débuguer** de **Visual C# Express** (vous auriez tout aussi bien quitter l'application de **L'Editeur du Coach C#** en utilisant les menus **Fichier > Quitter**)
- Faites un **clic** sur le contrôle **mainNotificationIconContextMenuStrip** dans la zone de dépôt de contrôles, afin de l'afficher sur le formulaire ;
- Faites un **double-clic** sur l'option du menu contextuel **Quitter l'application** ;



- Une méthode `quitterLapplicationToolStripMenuItem_Click` de réponse au menu **Quitter l'application** a été ajoutée au code du formulaire **Main.cs** ; Dans cette méthode, ajoutez le code de fermeture du formulaire ; C'est exactement la même ligne de code que celle que nous avons ajouté précédemment pour répondre au menu **Fichier > Quitter** du formulaire **Main.cs** (rien de difficile donc) ;

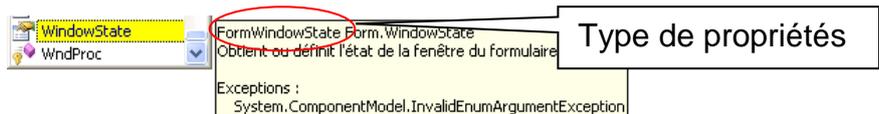
Code C#

```
private void quitterLapplicationToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Fermeture du formulaire
    this.Close();
}
```

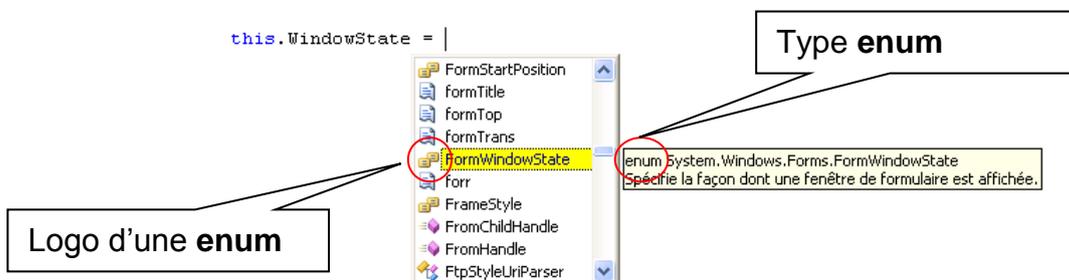
- Revenez sur le formulaire **Main.cs** en mode **[Design]** en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail ;
- Faites un **double-clic** sur l'option du menu contextuel **Maximiser la fenêtre**, pour ajouter la méthode de réponse à ce MENU `maximiserLaFenetreToolStripMenuItem_Click` ;



La propriété que nous allons utiliser est **WindowState** de l'instance de la fenêtre **Main** en cours (renvoyée par **this**). Cette propriété, accessible en lecture et en écriture obtient ou définit l'état de la fenêtre du formulaire.



Le type de cette propriété est **FormWindowState**, qui est en réalité une énumération de variables à valeurs constantes (**enum** en C#) ;



Les trois valeurs possibles proposées par cette énumération de valeurs sont : **Maximized**, **Minimized** et **Normal** (en fait, chacune de ces valeurs correspond à un entier unique dans la liste des valeurs, mais c'est bien plus lisible avec une énumération ☺) :

Liste des valeurs disponibles dans l'énumération

```
this.WindowState = FormWindowState.
```



C'est quoi un Type?

Le *Type* indique globalement la taille qu'un objet va utiliser en mémoire, la façon dont on va accéder à cette mémoire, et tout ce que cet objet peut faire. C# est un langage fortement typé. Autrement dit, tous les objets que vous allez utiliser ou que vous allez créer en C# doivent avoir un type. Le système de types C# comporte principalement des types valeurs (ou types intrinsèques) et des types références. Les variables de type valeur stockent directement des données (comme un entier ou une date) car le système sait exactement la taille à allouer pour ces données (un int32 fait toujours 32 octets !), alors que les variables de type référence stockent les références aux données (en gros, l'adresse des données en mémoire), car le système ne sais pas nécessairement la taille à allouer (quelle est la taille d'une liste de clients ?). Les types références sont également considérés comme des objets.



C'est quoi une variable ?

Pour les puristes, une variable est une instance d'un type par valeur (type intrinsèque). Bon, de mon côté, je désigne (comme la plupart des développeurs) une variable comme une instance d'un type quelconque, par valeur ou par référence (voire par pointeur si vous devez intégrer des librairie existante en C++ ☺). Une variable est composée d'un type et d'un identificateur (aussi dit *nom de la variable*) :



Assigner une variable, c'est lui donner une valeur en mémoire, soit à sa création, soit ultérieurement lors de l'exécution du programme. Une variable est donc fondamentalement en lecture/écriture (d'où son caractère variable [cqfd]).

```
int compteur = 5 ;
compteur = 6;
```



A la différence d'autres langages du marché, l'assignation des variables est obligatoire en C#. Autrement dit, vous devez nécessairement définir la valeur par défaut que va avoir votre variable, le compilateur ne pouvant le décider pour vous – cela évite bien des bogues difficiles à détecter lors de l'exécution.

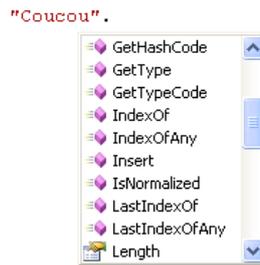


C'est quoi une constante ?

Une constante est une variable dont la valeur ne peut pas être changée lors de l'exécution du programme (elle est en lecture uniquement).

Une constante littérale est juste une valeur, comme 6, 12 ou "Coucou" ; D'ailleurs, tapez un point juste après la constante littérale "Coucou" , et

vous obtenez par l'IntelliSense tout ce qu'il est possible de faire sur une chaîne de caractères (magique, non ?).



Une constante symbolique est l'assignation d'un nom à une valeur. La déclaration d'une constante symbolique est réalisée par la ligne suivante :

```
const int valeurMaximumDuCompteur = 42;
```

Type

Identificateur

Assignation de la valeur



C'est quoi une énumération ?

Une énumération est un ensemble de constantes symboliques nommées.

Code C#

```
public enum FormWindowState
{
    Normal,
    Minimized,
    Maximized
}
```

La valeur des constantes de la liste n'étant pas précisée, le compilateur affecte des valeurs entières en commençant à 0 ; le code ci-dessus est donc similaire à :

Code C#

```
public enum FormWindowState
{
    Normal = 0,
    Minimized = 1,
    Maximized = 2
}
```

- Bon, revenons à nos moutons. En faisant le **double-clic** sur l'option du menu contextuel **Maximiser la fenêtre**, nous venons d'ajouter la méthode `maximiserLaFenetreToolStripMenuItem_Click`, qui est en fait la méthode de réponse à l'événement **click** du menu contextuel;
- Dans cette méthode, ajoutez le code pour maximiser la fenêtre en cours ;

Code C#

```
private void maximiserLaFenetreToolStripMenuItem_Click(object sender,
EventArgs e)
{
    // Maximise la fenêtre en cours
    this.WindowState = FormWindowState.Maximized;
}
```

- Revenez sur le formulaire **Main.cs** en mode **[Design]** en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail ;
- Faites maintenant un **double-clic** sur l'option du menu contextuel **Minimiser la fenêtre**, pour ajouter la méthode de réponse à ce menu, dans laquelle vous indiquez que l'état de la fenêtre est minimisé (`FormWindowState.Minimized`);
- Répétez encore l'opération pour l'option du menu contextuel **Restaurez la fenêtre**, et indiquez maintenant que l'état de la fenêtre est normal (`FormWindowState.Normal`);
- Le code de réponse des options du menu contextuel doit être le suivant :

Code C#

```
private void quitterLapplicationToolStripMenuItem_Click(object sender,
EventArgs e)
{
    // Fermeture du formulaire
    this.Close();
}

private void maximiserLaFenetreToolStripMenuItem_Click(object sender,
EventArgs e)
{
    // Maximise la fenêtre en cours
    this.WindowState = FormWindowState.Maximized;
}

private void restaurerLaFenetreToolStripMenuItem_Click(object sender,
EventArgs e)
{
    // Restaure la fenêtre en cours dans son état normal
    this.WindowState = FormWindowState.Normal;
}

private void minimiserLaFenetreToolStripMenuItem_Click(object sender,
EventArgs e)
{
    // Minimise la fenêtre en cours
    this.WindowState = FormWindowState.Minimized ;
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; **L'Editeur du coach C#** se lance, et vous devez voir votre icône dans la zone de notification

en bas à droite de votre écran. Si vous faites maintenant un clic-droit sur votre icône, vous devez voir apparaître le menu contextuel, que vous pouvez tester complètement. Magnifique, non ?



Un truc sympathique à faire serait de griser les menus contextuels qui sont inutiles en fonction du contexte de la fenêtre. Par exemple, le menu **Maximiser la fenêtre** pourrait être **grisé** quand la fenêtre est en état maximisé.



La propriété que nous allons utiliser est **Enabled** sur chacun des menus contextuel, en indiquant une valeur booléenne **true** ou **false**, pour respectivement le rendre actif ou le griser ;



Où faut-il connecter le code ? Si on regarde le besoin fonctionnel, on a : le menu **Maximiser la fenêtre** doit être grisé **quand** la fenêtre est en état maximisé ; Bon, on va faire un peu d'analyse ...



L'utilisation de la conjonction **quand** dans l'expression fonctionnelle indique l'utilisation d'un événement déclencheur : **quand** la fenêtre est en état maximisé (mais depuis quand les analystes-programmeurs doivent connaître la grammaire française – tout fout le camps, ffff !). Nous allons donc utiliser une fonction de réponse à un événement (« event handler ») ;



La conjonction **quand** dans l'expression fonctionnelle est généralement suivi de l'objet sur lequel l'événement va porter : quand **la fenêtre** est en état maximisé. La fonction de réponse à l'événement va donc porter sur la fenêtre en cours (i.e. sur l'objet **Main**) ;



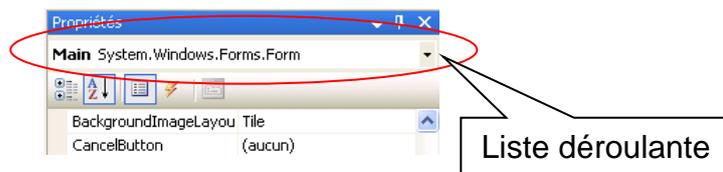
L'auxiliaire de l'expression fonctionnelle est généralement (dans le cas des événements) suivi du nom francisé (au sens large) de l'événement : quand la fenêtre est **en état maximisé**. C'est utile lorsqu'on ne connaît pas, à priori, le nom de l'événement à utiliser.



Où faut-il connecter le code, donc ? Dans notre cas, nous allons donc chercher s'il existe, sur l'objet fenêtre (« **Form** ») un événement comme « StateMaximized », ou « Maximized », ou encore un autre en relation avec la dimension de la fenêtre (« **Size** » en anglais), car maximiser une fenêtre revient à en changer la taille ; Et c'est là que nous allons

connecter le code !

- Revenez sur le formulaire **Main.cs** en mode **[Design]** en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail ;
- Faites un **clic-droit** à l'intérieur de la fenêtre (en dehors de tout autre contrôle, sur la zone grise de la fenêtre par exemple) et sélectionnez le menu **Propriétés** ;
- Dans la barre d'outils de la fenêtre de **Propriétés**, cliquez le bouton  pour afficher la liste des événements disponibles sur l'objet **Main** ; Pensez à vérifier que vous êtes bien sur le bon objet, dont le nom et le type s'affiche dans la liste déroulante en haut ;



- Dans la liste des événements disponibles, cherchez l'événement qui pourrait correspondre à notre besoin ; aidez-vous de l'aide succincte de chaque événement qui apparaît en bas de la fenêtre de propriétés quand vous faites un simple clic sur un événement ; Bon, malgré le nombre d'événements disponibles, il n'y en a aucun avec « State » - nous allons donc nous rabattre sur **SizeChanged** ...



Lorsqu'un événement est au **prétérit** (c'est-à-dire avec **ed** à la fin), alors il est déclenché après que l'action ait eu lieu. Par exemple, **SizeChanged** est déclenché après le changement de taille de la fenêtre, ou encore **FormClosed** est déclenché après la fermeture de la fenêtre. Si l'événement est au **présent progressif** (c'est-à-dire avec **ing** à la fin), l'événement est déclenché en tout début d'action, et il est généralement possible d'interagir avec cette action. Par exemple, **FormClosing** est déclenché en début de processus de fermeture de la fenêtre et il est possible d'annuler ce processus en laissant la fenêtre ouverte (ce qui est pratique si vous avez des données non sauvegardées ...) ;

- Faites un **double-clic** sur l'événement **SizeChanged** dans la fenêtre de propriétés ; Vous venez d'ajouter la méthode de prise en charge de l'événement `Main_SizeChanged` au code de l'application ;



L'idée de cette méthode est de configurer la propriété **Enabled** de chacun des menus contextuels à vrai (**true**) ou faux (**false**), en fonction de l'état de la fenêtre.

- Dans la méthode `Main_SizeChanged`, ajoutez le code suivant ;

Code C#

```
private void Main_SizeChanged(object sender, EventArgs e)
{
    // Grise les menus en fonction de l'état de la fenêtre
    this.maximiserLaFenetreToolStripMenuItem.Enabled =
```

```
!(this.WindowState == FormWindowState.Maximized);

this.minimiserLaFenetreToolStripMenuItem.Enabled =
    !(this.WindowState == FormWindowState.Minimized);

this.restaurerLaFenetreToolStripMenuItem.Enabled =
    !(this.WindowState == FormWindowState.Normal);
}
```



Le code `this.WindowState == FormWindowState.Minimized` utilise l'opérateur d'égalité `==`. Pour les types valeurs, l'opérateur d'égalité `==` retourne **true** si les valeurs des opérandes sont égales et **false** dans le cas contraire. Pour les types références autres que `string`, `==` retourne **true** si ses deux opérandes font référence au même objet. Pour le type `string`, `==` compare les valeurs des chaînes.



Le code `this.WindowState == FormWindowState.Minimized` retourne donc **true** dans le cas où l'état de la fenêtre est minimisé. Mais nous aimerions que le menu contextuel **Minimiser la fenêtre** soit grisé quand l'état de la fenêtre est minimisé. Nous allons donc utiliser un autre opérateur, celui de négation logique.



L'opérateur de négation logique `!` est un opérateur qui applique une négation sur l'opérande. Il est défini pour un opérande de type booléen et retourne **true** si, et seulement si, l'opérande est **false**, et inversement.



Le code `!(this.WindowState == FormWindowState.Minimized)` retourne donc **false** dans le cas où l'état de la fenêtre est minimisé, et c'est justement ce que nous aimerions afin que le menu contextuel **Minimiser la fenêtre** soit grisé quand l'état de la fenêtre est minimisé. Il nous reste donc à assigner la propriété **Enabled** du contrôle correspondant avec cette valeur, en utilisant l'opérateur d'assignation `=` ;



L'opérateur d'assignation `=` stocke la valeur de l'opérande de droite dans l'emplacement de stockage, la propriété (ou l'indexeur, mais cette notion est pour plus tard dans le coach) stipulé par l'opérande de gauche, et retourne la valeur comme résultat.

Les opérandes doivent être de même type (ou l'opérande de droite doit être implicitement convertible au type de l'opérande de gauche).



Et voilà, la ligne de code :

```
this.minimiserLaFenetreToolStripMenuItem.Enabled =
    !(this.WindowState == FormWindowState.Minimized);
```

grise donc le menu contextuel **Minimiser la fenêtre** quand la fenêtre est en état minimisé !



A la différence d'autres langages du marché, en C# l'opérateur d'assignation `=` est différencié de l'opérateur d'égalité `==`, ce qui réduit

fortement les risques de mauvaise interprétation des expressions.



C# propose un large éventail d'opérateurs. Ces derniers sont des symboles qui spécifient les opérations à effectuer dans une expression. C# prédéfinit les opérateurs arithmétiques et logiques habituels, ainsi que de nombreux autres, comme illustré à l'adresse suivante :

[http://msdn2.microsoft.com/fr-fr/library/6a71f45d\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/6a71f45d(VS.80).aspx)

- Reste maintenant à griser le menu **Restaurez la fenêtre** au lancement de l'application car la fenêtre est alors dans un état normal ; Pour ce faire, nous avons au moins deux moyens : changer la valeur par défaut de la propriété **Enabled** du menu contextuel **Restaurer la fenêtre**, ou le faire en une ligne de code (bon, on est là pour coder ?) ;
- Dans le constructeur de **Main**, ajoutez le code pour griser le menu contextuel :

Code C#

```
public Main()
{
    InitializeComponent();
    // Assignment de l'icône de mainNotifyIcon
    mainNotifyIcon.Icon = Properties.Resources.lan_connected ;

    // Griser le menu Restaurer la fenêtre
    this.minimiserLaFenêtreToolStripMenuItem.Enabled = false;
}
```

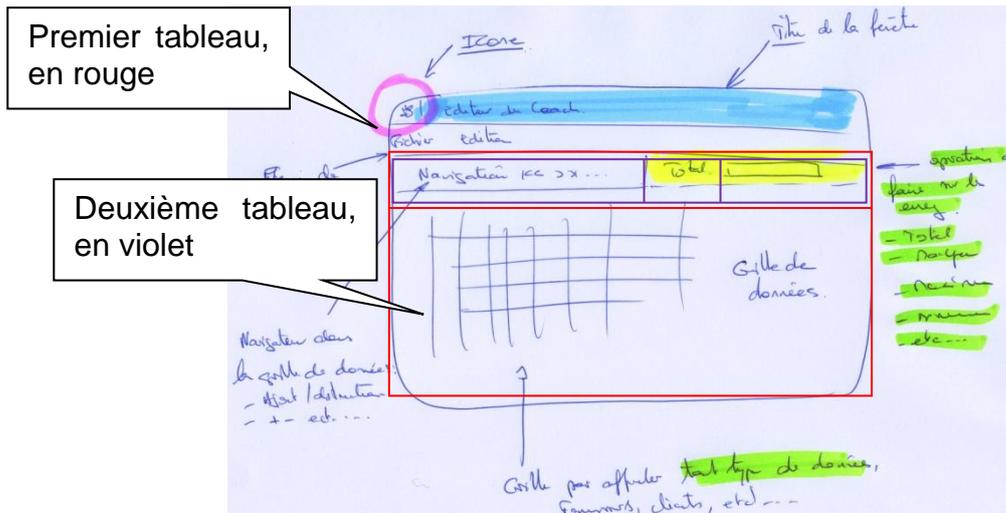
- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; Testez le tout pour voir comment cela fonctionne bien. *Good job guys !*

2.3 Positionner les contrôles de données

Il s'agit dans cette exercice de positionner sur le formulaire les différents contrôles de données que nous allons alimenter pour manipuler les données. Mais avant de déposer les contrôles de données, nous allons tout d'abord structurer le formulaire à l'aide de tableaux.



Si on regarde le document d'analyse fonctionnel, on peut (avec de l'imagination, certes, mais nous sommes des créatifs 😊) voir une structure de tableaux imbriqués :



Le premier tableau contient une colonne et deux lignes ; la première ligne de ce tableau contient un deuxième tableau d'une seule ligne mais avec trois colonnes. Bref, tout ce beau monde est imbriqué pour structurer l'interface utilisateur et proposer un ensemble de cellules destinées à contenir les contrôles d'interface (grille, boîte de texte, libellé, barre de navigation, etc.)



Pour structurer l'interface, nous allons utiliser le contrôle **TableLayoutPanel** dont l'objectif est de manipuler la disposition des composants et de les organiser automatiquement sous forme de tableau.

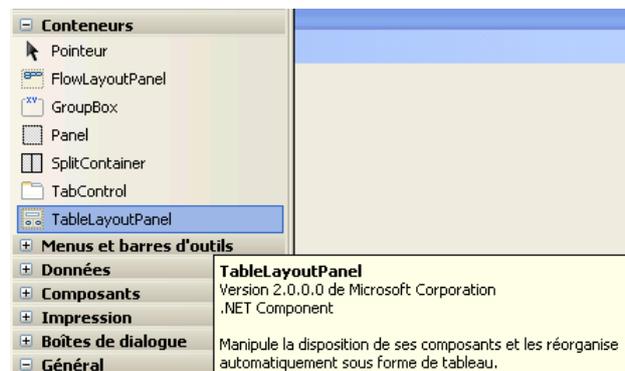


Pour tout savoir sur le contrôle **TableLayoutPanel**, voici le lien : [http://msdn2.microsoft.com/fr-fr/library/3a1tbfd\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/3a1tbfd(VS.80).aspx)

Déroulement de l'exercice :

1. Structurez le formulaire sous forme de tableaux :

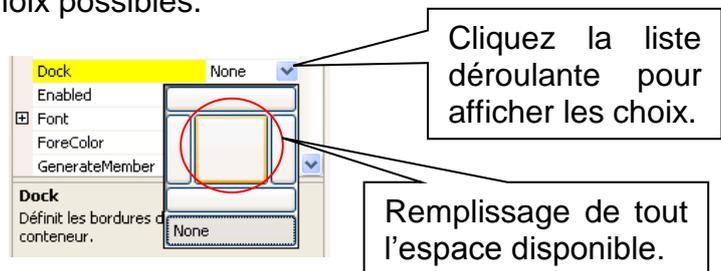
- Revenez sur le formulaire **Main.cs** en mode **[Design]** (soit en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail, soit en double-cliquant sur le fichier **Main.cs** dans l'**Explorateur de solutions**) ;
- Dans la boîte à outils, ouvrez l'onglet **Conteneurs** en cliquant sur  qui précède le nom de l'onglet ;



- Faites un **glisser-déplacer** du contrôle **TableLayoutPanel** sur le formulaire ; un tableau apparaît à la surface du formulaire ;



- Affichez la fenêtre de propriété du contrôle que vous venez d'insérer, dont le nom doit être **tableLayoutPanel1** ;
- Dans la fenêtre de propriétés, configurez la propriété **Dock** de façon à ce que le contrôle remplisse toute la surface disponible du formulaire (**Fill**). L'éditeur de propriété affiche une boîte de dialogue de sélection visuelle des différents choix possibles.



En fait, la mise en forme de base comporte deux fonctions : l'ancrage (**Anchor**) et le docking (**Dock**). Par exemple, le contrôle **TableLayoutPanel** a une propriété **Anchor** qui est un type énuméré dont les valeurs (qui peuvent être traitées par un **ou** logique) indiquent de quel(s) bord(s) du contrôle parent le contrôle **TableLayoutPanel** gardera une distance constante. Par exemple, en configurant la propriété **Anchor** à `AnchorStyles.Bottom`, le tableau restera toujours à la même distance du bord inférieur du formulaire, lorsque ce dernier sera redimensionné.

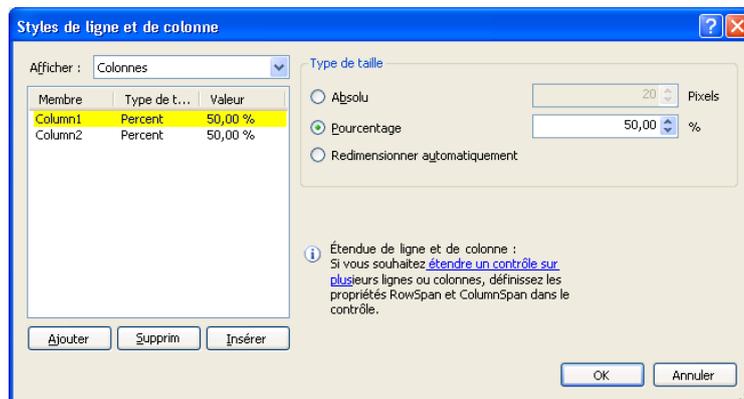
Le docking n'est en fait qu'un cas spécial de l'ancrage. La propriété **Dock** décrit à quel bord du contrôle parent un contrôle doit se relier. Il y a le docking Haut (**Top**), Gauche (**Left**), Droit (**Right**), Bas (**Bottom**), ou Plein (**Fill**). Dans chaque cas, le contrôle est déplacé aussi près que possible du bord spécifié et dimensionné de manière à remplir ce bord, où il reste lorsque le parent est redimensionné. Le docking est une sorte d'ancrage avec une distance de zéro par rapport au bord.

Les deux propriétés **Anchor** et **Dock** possèdent un éditeur de propriétés visuel.

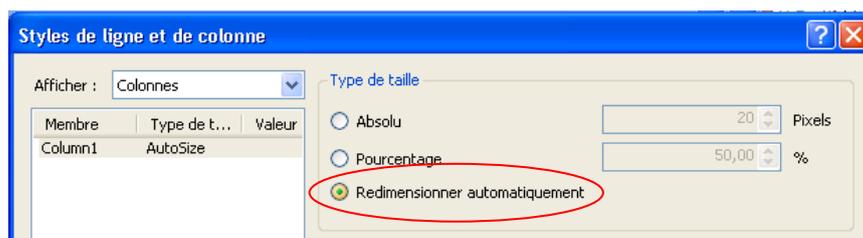
- Dans la fenêtre de propriétés du contrôle **tableLayoutPanel1**, sélectionnez la propriété **Columns**, qui permet de travailler avec la collection de colonnes et de lignes du tableau ; Un bouton  s'affiche en face de la propriété afin d'ouvrir l'éditeur de propriétés correspondant ;



- Cliquez sur le bouton  pour afficher l'éditeur de propriétés ;



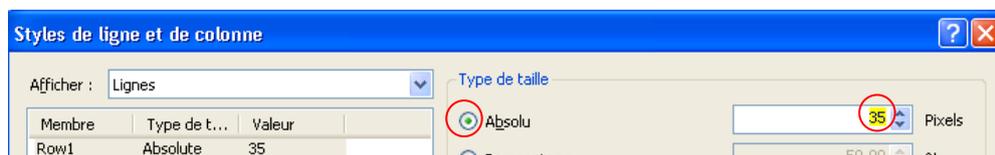
- Sélectionnez la colonne **Column2**, et cliquez le bouton **Supprimer** de l'éditeur de propriétés ;
- Sélectionnez la colonne **Column1**, et indiquez que sa dimension est automatiquement calculée en cliquant le radio-bouton **Redimensionner Automatiquement** ;



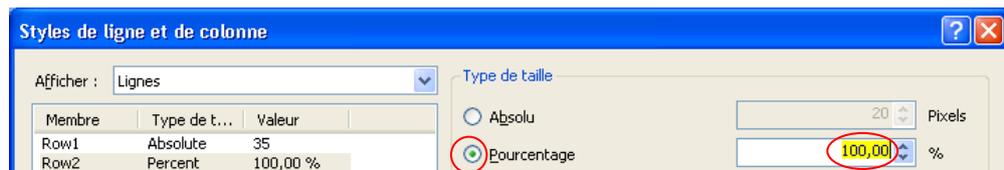
- Dans la liste déroulante **Afficher**, sélectionnez le choix **Lignes** ;



- Pour la première ligne, dont le nom est **Row1**, indiquez une taille de type **Absolu** et de **35 pixels** ;



- Pour la deuxième ligne, dont le nom est **Row2**, indiquez une taille de type **Pourcentage** et de **100 %**, afin d'occuper tout l'espace restant disponible ;



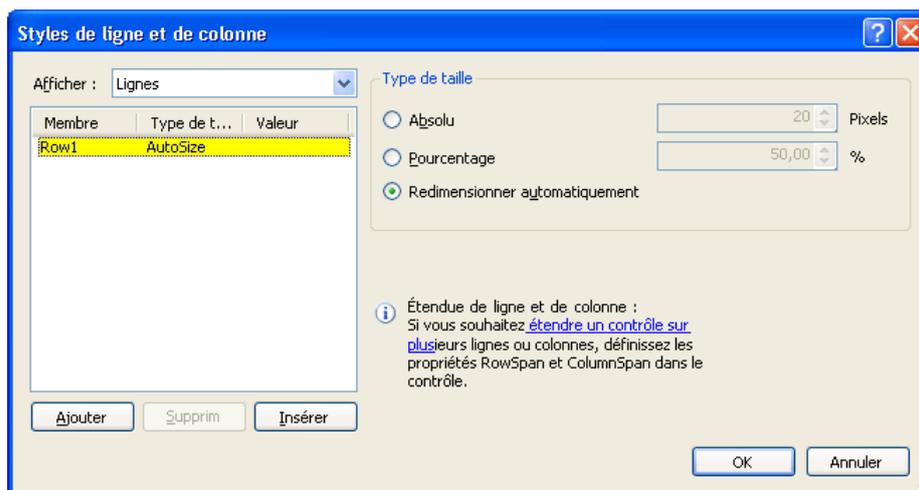
- Cliquez maintenant le bouton **Ok** afin de fermer la boîte de dialogue **Styles de ligne et de colonne** ;
- Faites un **glisser-déplacer** d'un deuxième contrôle **TableLayoutPanel** à l'intérieur de la première ligne du tableau précédemment ajouté ;
- Affichez la fenêtre de propriétés du contrôle que vous venez d'insérer, dont le nom doit être **tableLayoutPanel2** ;
- Dans la fenêtre de propriétés, configurez la propriété **Dock** de façon à ce que le contrôle remplisse toute la surface disponible de la ligne qui le contient (**Fill**) ;
- Sélectionnez le contrôle **tableLayoutPanel2** que vous venez d'insérer sur la surface du formulaire, et vous devez remarquer une petite flèche en haut à droite du contrôle ;



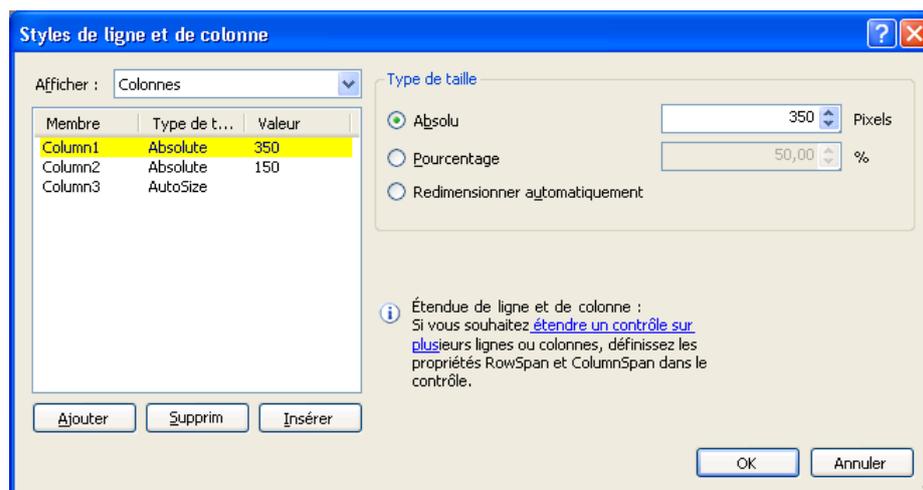
- Cliquez sur cette flèche  ; Elle vous propose les tâches les plus courantes à faire sur ce contrôle (cela marche pour la plupart des contrôles .Net 2.0) ;



- Sélectionnez la tâche **Modifier les lignes et les colonnes...** et la boîte de dialogue **Styles de ligne et de colonne** s'affiche à l'écran. C'est exactement la même que celle de l'éditeur de propriété **Columns** que nous venons juste d'utiliser ;
- Dans la liste déroulante **Afficher**, sélectionnez **Lignes** ;
- **Détruisez** la ligne **Row2** ;
- **Indiquez** un redimensionnement automatique pour le ligne **Row1** ;



- Dans la liste déroulante **Afficher**, sélectionnez **Colonnes** ;
- Ajoutez une nouvelle colonne en cliquant sur le bouton **Ajouter** ;
- Indiquez une taille absolue de **350 pixels** pour la colonne **Column1** ;
- Indiquez une taille absolue de **150 pixels** pour la colonne **Column2** ;
- Indiquez un redimensionnement automatique pour la colonne **Column3** ;



- Cliquez maintenant le bouton **Ok** afin de fermer la boîte de dialogue **Styles de ligne et de colonne** ;
- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; Testez le tout pour vérifier que tout continue de bien fonctionner. Vous constatez notamment que les contrôles de tableau n'ont aucun impact visuel (bordure, etc.) sur le formulaire (circulez, il n'y a rien à voir !).



Visual C# Express est livré avec de nombreux contrôles. La description et l'utilisation de chacun d'entre eux est sur le lien suivant :

[http://msdn2.microsoft.com/fr-fr/library/3xdhey7w\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/3xdhey7w(VS.80).aspx)



Pour les développeurs qui viendraient de Visual Basic 6.0, de la programmation MFC ou encore de FoxPro, le lien suivant explique les correspondances entre les anciens contrôles et les contrôles .Net :

[http://msdn2.microsoft.com/fr-fr/library/0061wezk\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/0061wezk(VS.80).aspx)



Pour tout ce qui concerne l'utilisation des contrôles, de la boîte à outils ou encore de l'ajout de contrôles ActiveX à une application .Net :

[http://msdn2.microsoft.com/fr-fr/library/3deasc0e\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/3deasc0e(VS.80).aspx)



Et enfin, un lien pour tout savoir sur la disposition des contrôles dans un formulaire Windows, notamment pour éviter de passer des heures à essayer d'aligner deux boutons et une boîte de texte :

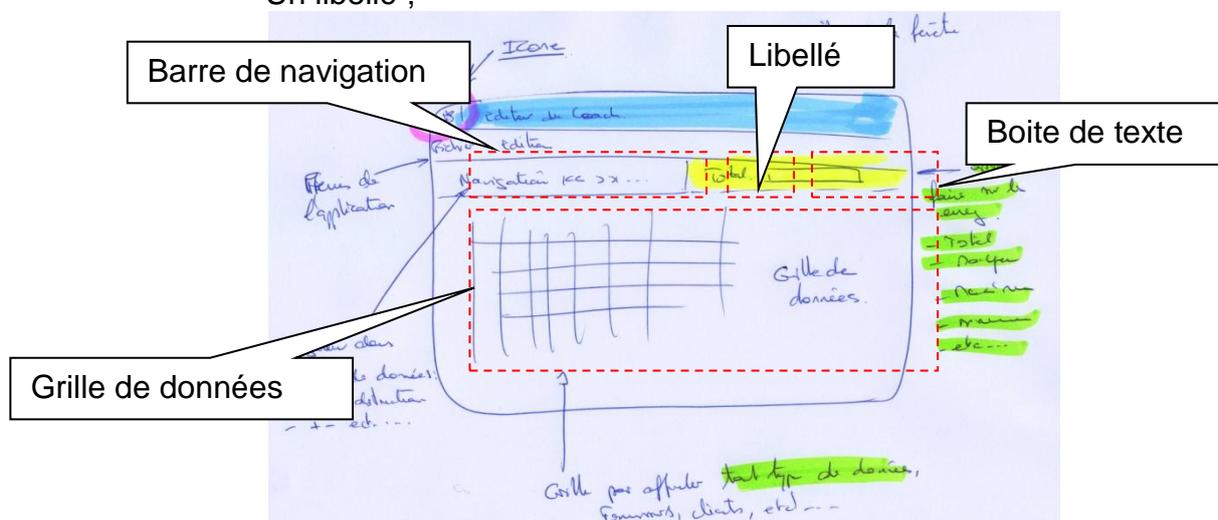
[http://msdn2.microsoft.com/fr-fr/library/ty26a068\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/ty26a068(VS.80).aspx)

2. Ajoutez les contrôles de données :



Bon, nous avons maintenant une structure de tableau avec 4 cellules. A priori, nous allons donc maintenant ajouter 4 contrôles, un par cellule (Cqfd !). En regardant le document d'analyse fonctionnelle (comme quoi, cela sert un peu de faire des documents d'analyse), l'interface utilisateur est composée des contrôles suivants :

- Une barre de navigation,
- Une grille d'affichage des données,
- Une boîte de texte d'affichage de résultat,
- Un libellé ;



Outre le libellé et la boîte de texte qui sont des contrôles de base de tout bon outil de développement, Visual C# Express propose deux contrôles de données répondants à nos besoins : le **DataGridView** et le **BindingNavigator**. Ce sont ces contrôles que nous allons utiliser :

- Le contrôle **DataGridView** offre un moyen puissant et flexible pour afficher des données sous forme de tableau. Il est possible d'étendre le contrôle **DataGridView** afin de générer des comportements personnalisés. Par exemple, vous pouvez spécifier par programme vos propres algorithmes de tri, et vous pouvez créer vos propres types de cellules ;

- Le contrôle **BindingNavigator** propose une façon standardisée pour parcourir et manipuler des données sur un formulaire. C'est en fait une barre d'outils qui a été complétée par les équipes de développement de Microsoft afin de gérer automatiquement la navigation dans les données ;

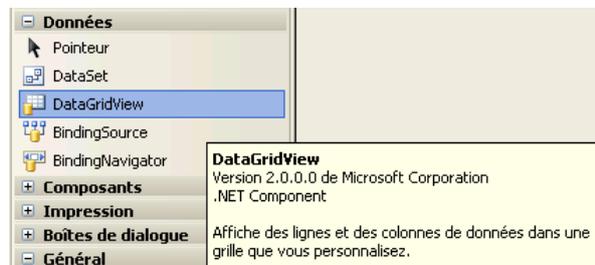


Pour tout ce qui concerne l'utilisation du contrôle **DataGridView** :
[http://msdn2.microsoft.com/fr-fr/library/e0ywh3cz\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/e0ywh3cz(VS.80).aspx)

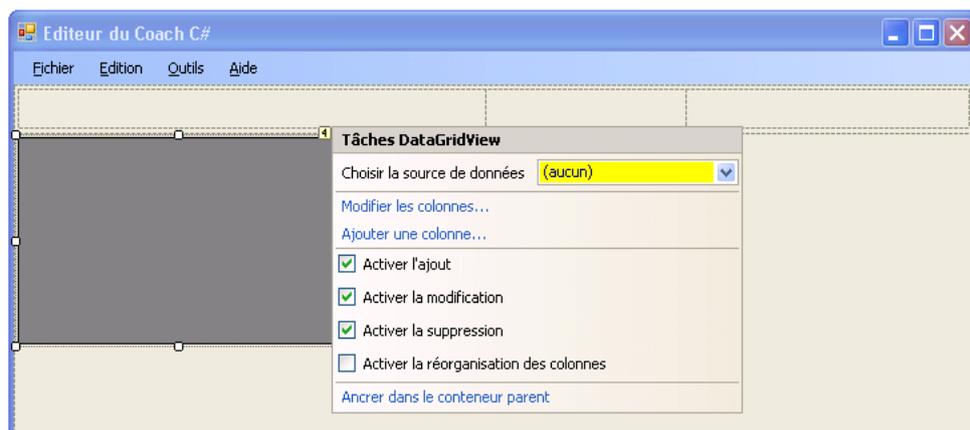


Pour tout ce qui concerne l'utilisation du contrôle **BindingNavigator** :
[http://msdn2.microsoft.com/fr-fr/library/ms158105\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/ms158105(VS.80).aspx)

- Revenez sur le formulaire **Main.cs** en mode **[Design]** (soit en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail, soit en double-cliquant sur le fichier **Main.cs** dans l'**Explorateur de solutions**) ;
- Dans la boîte à outils, ouvrez l'onglet **Données** en cliquant sur **+** qui précède le nom de l'onglet ;

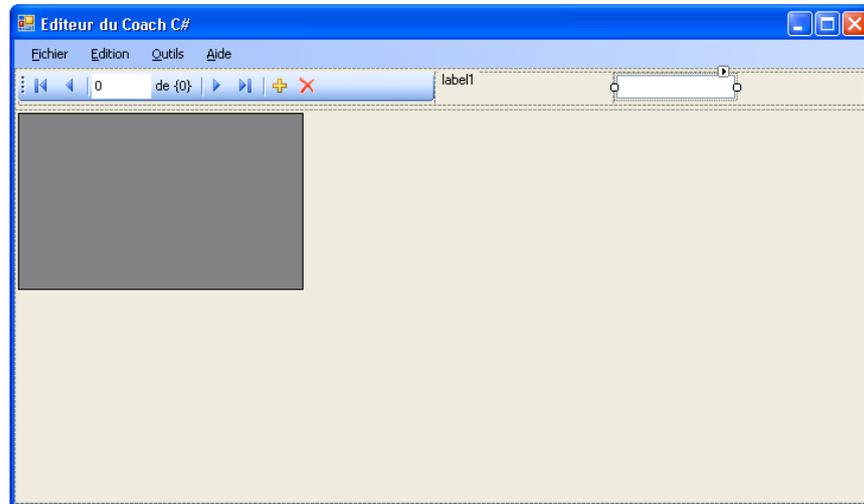


- Faites un **glisser-déplacer** du contrôle **DataGridView** sur le formulaire, dans la plus grande cellule du tableau (la dernière) ; une grille (vide bien sûr) apparaît à la surface du formulaire ;

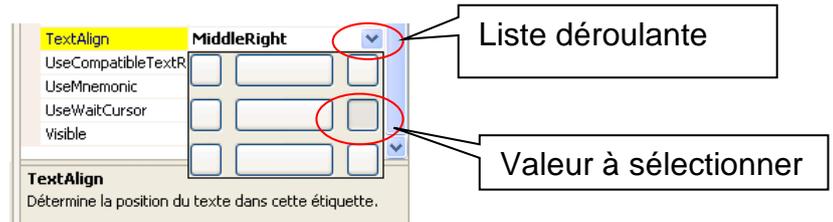


- Faites un **glisser-déplacer** du contrôle **BindingNavigator** sur le formulaire, dans la première cellule du tableau ;
- Dans la boîte à outils, ouvrez l'onglet **Contrôles communs** en cliquant sur **+** qui précède le nom de l'onglet ;
- Faites un **glisser-déplacer** du contrôle **Label** sur le formulaire, dans la deuxième cellule du tableau ;

- Enfin, faites un **glisser-déplacer** du contrôle **TextBox** sur le formulaire, dans la troisième cellule du tableau ; Vous obtenez un formulaire comme ci-dessous. Il reste maintenant à personnaliser les propriétés de chacun des contrôles ;

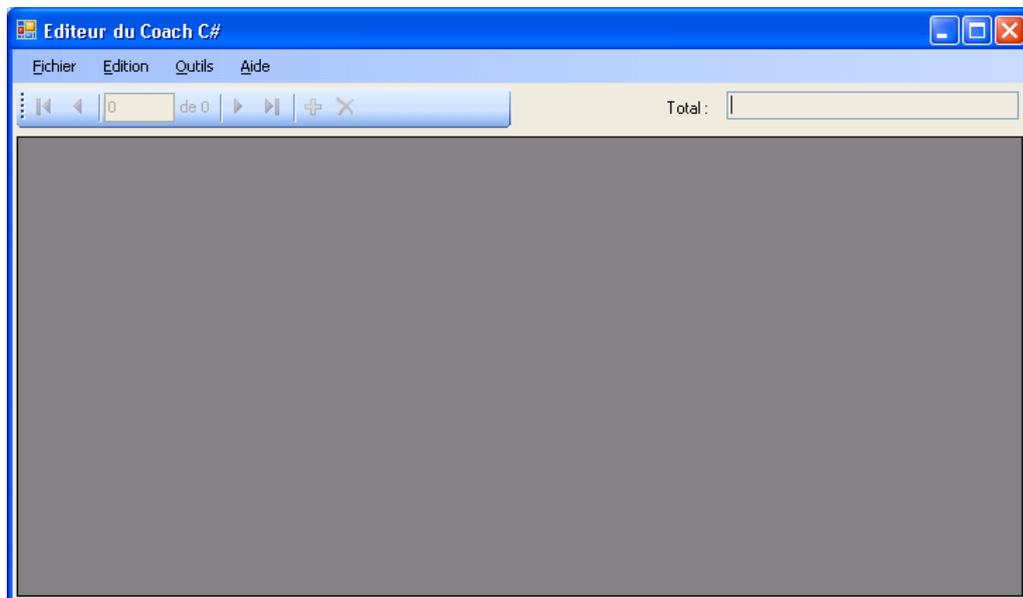


- **Appuyez** sur la touche **Shift**  et maintenez la enfoncée ;
- Avec la souris, faites un **clic** sur chacun des quatre contrôles que vous venez d'ajouter, tout en maintenant la touche **Shift** appuyée, afin de faire une sélection multiple de contrôles ;
- **Lâchez** maintenant la touche **Shift** ;
- Affichez la fenêtre de propriétés de la sélection, qui vous propose toutes les propriétés communes des contrôles de votre sélection ;
- Dans la fenêtre de propriétés, configurez la propriété **Dock** de façon à ce que les contrôles remplissent toute la surface disponible de la cellule dans lesquels ils sont (**Fill**) ;
- **Sélectionnez** maintenant uniquement la grille de données **dataGridView1** (désélectionnez l'ensemble des contrôles en cliquant sur la barre de titre du formulaire, par exemple, puis cliquez sur la grille pour la sélectionner) ;
- Dans la fenêtre de **Propriétés**, configurez le nom du contrôle en changeant la propriété (**Name**) avec **mainDataGridView** (avec C#, attention à la casse des noms !) ;
- **Sélectionnez** maintenant uniquement la barre d'outils de navigation **bindingNavigator1** ;
- Dans la fenêtre de **Propriétés**, configurez le nom du contrôle en changeant la propriété (**Name**) avec **mainBindingNavigator** ;
- Ensuite, **sélectionnez** uniquement le libellé **label1** ;
- Dans la fenêtre de **Propriétés**, configurez le nom de ce contrôle en changeant la propriété (**Name**) avec **lblOperation** ;
- Dans la fenêtre de **Propriétés**, configurez la façon dont le texte va être positionné dans le libellé, en changeant la propriété **TextAlign** avec **MiddleRight** (alignement du contenu à droite du libellé, avec un centrage vertical) ; Utilisez au besoin l'éditeur de propriétés proposé en cliquant sur la liste déroulante de la propriété ;



- Toujours dans la fenêtre de **Propriétés**, configurez le texte à afficher en changeant la propriété **Text** avec **Total** ;
- Pour finir, **sélectionnez** uniquement la boîte de texte **TextBox1** ;
- Dans la fenêtre de **Propriétés**, configurez le nom de ce contrôle en changeant la propriété (**Name**) avec **txtOperation** ;
- Enfin, dans la fenêtre de **Propriétés**, indiquez que cette boîte de texte va être en lecture uniquement, en changeant la propriété **ReadOnly** avec **True** (valeur booléenne Vrai) ;
- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; Testez le tout pour vérifier que tout fonctionne encore bien.

La fenêtre suivante apparaît :



Aucune donnée n'est affichée, mais tout fonctionne correctement. C'est super (surtout si vous n'avez aucun bogue de compilation à résoudre).

2.4 Localiser l'application

L'objectif de cet exercice est de rendre disponible l'interface utilisateur de l'application dans une autre langue que le français, par exemple l'anglais.



Si vous envisagez de distribuer votre application à un public international, vous devrez garder à l'esprit certaines considérations lors des phases de design et de développement. Et, même si vous ne pensez pas immédiatement distribuer votre application à un public international, de simples préparatifs pourront vous faciliter considérablement la tâche dans l'éventualité où les versions ultérieures de votre application aient des objectifs différents.



La localisation est le processus de personnalisation de votre application pour une culture ou des paramètres régionaux donnés. La localisation consiste principalement à traduire l'interface utilisateur.



La globalisation est le processus de conception et de développement d'un produit logiciel adapté à plusieurs cultures.



Pour tout savoir sur la localisation et la globalisation :

[http://msdn2.microsoft.com/fr-fr/library/1021kkz0\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/1021kkz0(VS.80).aspx)



C'est quoi une ressource ?

Lorsque vous localisez une application, l'ensemble des données localisées, comme des chaînes et des images qui sont adaptées à chaque culture, sont stockées avec Visual C# Express dans des fichiers séparés d'extension **.resx** ; Ces différentes données qui vont être chargées au moment de l'exécution en fonction de la culture sont appelées les ressources de l'application.



Dans Visual Studio, les ressources localisées (des données comme des chaînes et des images qui sont adaptées à chaque culture) sont stockées dans des fichiers séparés et chargées en fonction de la configuration de la culture de l'interface utilisateur. Pour comprendre comment les ressources sont chargées, imaginez qu'elles sont organisées sous la forme d'une hiérarchie.

- En haut de la hiérarchie se trouvent les ressources par défaut, c'est-à-dire celles correspondant à la langue de développement de l'application. Dans notre cas, il s'agit du français. Ce sont les seules ressources qui n'ont pas leur propre fichier. Au moment de la compilation, elles sont stockées dans l'assemblage principal de l'application (par exemple **nom.exe**) ;
- Sous ces ressources par défaut se trouvent les ressources des cultures neutres. Une culture neutre est associée à une langue, mais pas à une région. Par exemple, Français (« fr ») ou l'anglais (« en ») sont des cultures neutres. Lors de la compilation, elles sont stockées dans un assemblage satellite lui-même stocké dans un sous répertoire ayant un nom référençant la culture neutre (par exemple **fr**) ;
- En dessous de ces ressources de cultures neutres se trouvent les ressources des cultures spécifiques. Une culture spécifique est associée à une langue et à une région. Par exemple, Français

(Canada) ("fr-CA") est une culture spécifique. Lors de la compilation, elles sont stockées dans un assemblage satellite lui-même stockée dans un sous répertoire ayant un nom référençant la culture spécifique (par exemple **fr-CH**) ;

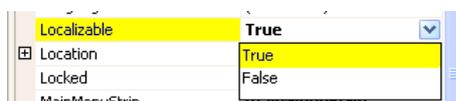


Si une application essaie de charger une ressource localisée, telle qu'une chaîne, et ne la trouve pas, elle remontera dans la hiérarchie jusqu'à ce qu'elle trouve un fichier de ressources contenant la ressource demandée.

Déroulement de l'exercice :

1. Configurez la localisation de votre formulaire :

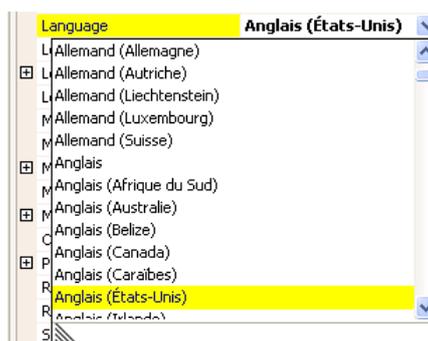
- Revenez sur le formulaire **Main.cs** en mode **[Design]** (soit en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail, soit en double-cliquant sur le fichier **Main.cs** dans l'**Explorateur de solutions**) ;
- Dans la fenêtre de **Propriétés** du formulaire **Main**, indiquez que ce dernier supporte la localisation en configurant la propriété **Localizable** à **True** (Vrai) ;



- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;

2. Localisez votre application en anglais :

- Revenez sur le formulaire **Main.cs** en mode **[Design]** (soit en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail, soit en double-cliquant sur le fichier **Main.cs** dans l'**Explorateur de solutions**) ;
- Dans la fenêtre de **Propriétés** du formulaire **Main**, indiquez que ce dernier est maintenant affiché par le designer Visual C# Express en localisation Anglais (Etats-Unis), en configurant la propriété **Language** à **Anglais (Etats-Unis)** ;



- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Affichez maintenant l'**Explorateur de solutions** ; Un nouveau fichier **Main.en-US.resx** a été ajouté à la solution, en dépendance de **Main.cs** ; Ce fichier contient les ressources de l'application pour la nouvelle langue ajoutée ;



- Revenez sur le formulaire **Main.cs** en mode **[Design]** ;
- Pour chacun des contrôles (formulaire, libellé, menus, etc.), changez la propriété **Text** par un contenu en Anglais ;
- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; L'application fonctionne (normalement) mais est toujours en français ! C'est normal, votre poste étant configuré avec un système d'exploitation français, la culture qui est lu est donc celle par défaut.

3. Définissez la culture de l'interface utilisateur :

- Affichez le code du formulaire **Main.cs**, en faisant un clic-droit sur **Main.cs** dans l'**Explorateur de solutions**, et en sélectionnant le menu **Afficher le code** ;
- Dans le constructeur de la fenêtre **Main.cs**, ajoutez le code pour changer la culture du processus exécutant l'application ;

Code C#

```
public Main()
{
    // Configure la culture à Anglais (Etats-Unis)
    System.Threading.Thread.CurrentThread.CurrentCulture =
        new System.Globalization.CultureInfo("en-US");
    // Configure la culture de l'interface à Anglais (Etats-Unis)
    System.Threading.Thread.CurrentThread.CurrentUICulture =
        new System.Globalization.CultureInfo("en-US");

    InitializeComponent();
    // Assignment de l'icone de mainNotifyIcon
    mainNotifyIcon.Icon = Properties.Resources.lan_connected ;

    // Griser le menu Restaurer la fenêtre
    this.minimiserLaFenetreToolStripMenuItem.Enabled = false;
}
```



Pensez à configurer la culture d'exécution avant l'appel à `InitializeComponent()` ; en effet, cette méthode va configurer les différents contrôles du formulaire, dont leurs propriétés **Text**, et donc elle va charger les ressources dont elle a besoin ; si la culture du processus

est configurée après l'appel de cette méthode, ce sera trop tard car les fichiers de ressources auront déjà été lus ...



C'est quoi **CultureInfo** ?

La classe **CultureInfo** rend des informations spécifiques à une culture, par exemple la langue, la sous-langue, le pays et/ou la région ou le calendrier associés à la culture, ainsi que les conventions applicables à cette dernière. Elle spécifie un nom unique pour chaque culture, basé sur la norme RFC 3066 pour Windows Vista et les versions ultérieures, et sur la norme RFC 1766 pour les systèmes d'exploitation antérieurs à Windows Vista. Le nom est une combinaison d'un code de culture à deux lettres minuscules ISO 639 associé à une langue et d'un code de sous-culture à deux lettres majuscules ISO 3166 associé à un pays ou une région. Le format du nom de culture est <languagecode2>-<country/regioncode2>, où <languagecode2> est le code de langue et <country/regioncode2> est le code de sous-culture.



Pour tout savoir sur la classe **CultureInfo**, notamment les codes de chacune des cultures disponibles :

[http://msdn2.microsoft.com/fr-fr/library/kx54z3k7\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/kx54z3k7(VS.80).aspx)



Quelle est la différence entre `CurrentThread.CurrentCulture` et `CurrentThread.CurrentUICulture` ?

- **CurrentCulture** obtient ou définit la culture du thread en cours, c'est-à-dire la façon dont l'application va traiter en interne le format des nombres, des dates ou de chaînes de caractères par exemple ;
- **CurrentUICulture** obtient ou définit la culture actuelle utilisée par le gestionnaire de ressources pour rechercher des ressources spécifiques à la culture au moment de l'exécution ;

C'est pour cette raison qu'il faut changer les deux propriétés dans le code.



Pour tout savoir sur la classe **Thread**, qui fournit plein de renseignements sur le processus en cours notamment au travers de la propriété `Thread.CurrentThread` :

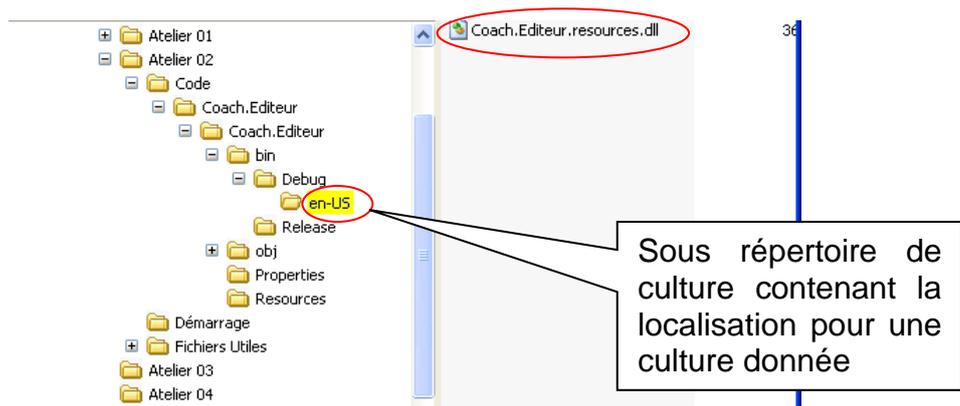
[http://msdn2.microsoft.com/fr-fr/library/h158zycw\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/h158zycw(VS.80).aspx)

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; L'application fonctionne maintenant en anglais ;



Allez voir dans le répertoire de compilation de l'application (par exemple `C:\Coach C#\Atelier 02\Code\Coach.Editeur\Coach.Editeur\bin\Debug`), et vous verrez que le compilateur a généré automatiquement un sous répertoire contenant l'assemblage satellite des ressources en anglais. Lors de la distribution de l'application, il faudra penser à distribuer l'assemblage principal, mais aussi tous ses assemblages satellites de

localisation.



Pour tout savoir sur la classe la gestion des ressources dans les application Windows :

[http://msdn2.microsoft.com/fr-fr/library/f45fce5x\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/f45fce5x(VS.80).aspx)



Pour les traducteurs qui localisent les applications, il existe un outil spécifique dans le SDK de .Net 2.0 : **Winres.exe**. **Winres.exe** est une application graphique qui recrée une version en mode WYSIWYG d'un formulaire Windows simplement à partir du fichier de ressources, sans devoir accéder au code source. **Winres.exe** héberge le concepteur de formulaires Windows et sa fenêtre Propriétés. Ces fonctionnalités permettent une modification visuelle d'un fichier **.resources** ou **.resx** contenant un formulaire Windows Forms.



Pour tout savoir sur **Winres.exe** :

[http://msdn2.microsoft.com/fr-fr/library/8bxdx003\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/8bxdx003(VS.80).aspx)



Pour télécharger le SDK .Net 2.0 :

<http://www.microsoft.com/downloads/details.aspx?FamilyID=fe6f2099-b7b4-4f47-a244-c96d69c35dec&displaylang=fr>



Les développeurs utilisent généralement aussi des ressources directement codées « en dur » dans le code, comme par exemple les chaînes de caractères de messages. La création et l'utilisation de fichiers de ressources pour ces chaînes de caractères est une tâche plutôt délicate, si elle est réalisée à la fin de la programmation car elle nécessite une relecture complète du code. Il existe un outil, le **Resource Refactoring Tool**, qui peut aider à résoudre ce problème.



Pour télécharger le le **Resource Refactoring Tool**,:

<http://visualstudiogallery.msdn.microsoft.com/39ae29d3-81e1-43d4-9c48-fc9644869d84>

4. Configurez de nouveau la culture de l'interface utilisateur en français :

- Affichez le code du formulaire **Main.cs**, en faisant un clic-droit sur **Main.cs** dans l'**Explorateur de solutions**, et en sélectionnant le menu **Afficher le code** ;
- Dans le constructeur de la fenêtre **Main.cs**, mettez en commentaire le code pour changer la culture du processus exécutant l'application ;

Code C#

```
public Main()
{
    // Configure la culture à Anglais (Etats-Unis)
    // System.Threading.Thread.CurrentThread.CurrentCulture =
    //     new System.Globalization.CultureInfo("fr-FR");
    // Configure la culture de l'interface à Anglais (Etats-Unis)
    // System.Threading.Thread.CurrentThread.CurrentUICulture =
    //     new System.Globalization.CultureInfo("fr-FR");

    InitializeComponent();
    // Assignation de l'icone de mainNotifyIcon
    mainNotifyIcon.Icon = Properties.Resources.lan_connected ;

    // Griser le menu Restaurer la fenêtre
    this.minimiserLaFenêtreToolStripMenuItem.Enabled = false;
}
```

- Revenez sur le formulaire **Main.cs** en mode **[Design]** (soit en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail, soit en double-cliquant sur le fichier **Main.cs** dans l'**Explorateur de solutions**) ;
- Dans la fenêtre de **Propriétés** du formulaire **Main**, indiquez que ce dernier est maintenant affiché par le designer Visual C# Express en localisation par défaut, en configurant la propriété **Language** à **(Par défaut)** ;
- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; L'application fonctionne maintenant en français, qui est notre langue par défaut ;

Félicitations, vous venez de passer une première étape importante du Coach C# (nous venons de finaliser l'essentiel de l'interface utilisateur que nous allons utiliser tout au long des différents ateliers), et il nous reste maintenant à alimenter cet écran avec des données.

3 Alimenter la grille avec le contenu d'un fichier CSV

Dans cet exercice, vous allez apprendre à :

- Lire un fichier au format CSV (Texte délimité) ;
- Utiliser les principales boucles du langage C# ;
- Alimenter les contrôles de données avec les informations lues ;

Objectif

L'objectif de cet exercice est de manipuler des données au format CSV, tout en abordant la structuration d'un programme et les ordres de boucles et de tests du langage C#.

Contexte fonctionnel

Nous allons implémenter un accès aux données à l'ancienne, c'est-à-dire que nous allons utiliser des données stockées au format CSV, en les transformant (pour faciliter l'alimentation de la grille de données) en un objet table en mémoire. Bon, c'est pas exactement ce qu'il faudrait faire de nos jours (c'est beaucoup de lignes de code pour pas grand chose), mais cela va nous aider dans notre démarche progressive de découverte de C#

Le fichier de données que nous allons utiliser est le fichier **Clients.coach** situé dans les fichiers utiles de l'exercice Il est possible de l'ouvrir avec le bloc-notes de Windows, car c'est juste un fichier au format texte.

Le principe à implémenter est le suivant



Let's go, folks !



Le Framework .Net propose en standard de nombreuses classes à utiliser dans vos développements. Ces classes sont regroupées en **espaces de nommage** (Namespace, cf. ci-dessous). Nous allons donc créer un objet de type **DataTable**, qui représente une table de données, se trouvant dans l'espace de nommage **System.Data**.



C'est quoi un **espace de nommage** (Namespace) ?

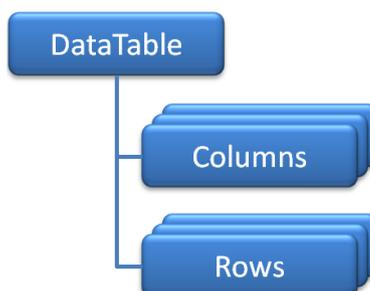
Les espaces de nommages aident à organiser logiquement et clairement les types et autres membres fournis par le Framework .Net (notamment), ou par tout autre application et/ou assemblage .Net, et ce indépendamment du nom de l'assemblage .net fournissant ces types ou ces membres. Cependant, souvent les noms des assemblages du Framework .Net réfèrent à l'espace de nommage principalement enrichie

par l'assemblage : par exemple **system.xml.dll** enrichie principalement l'espace de nom **System.Xml** (manipulation de données XML), mais pas seulement. L'espace de nommage **System.Data** contient tous les types requis pour manipuler des données ; De la même manière, **System.Configuration** contient tous les types pour gérer la configuration d'une application. Nous reviendrons durant le troisième atelier sur l'organisation des espaces de nommages.



C'est quoi une **DataTable** ?

Une **DataTable** représente une table de données en mémoire. C'est exactement ce que l'on souhaite faire ! (le hasard fait bien les choses). La structure globale d'une **DataTable** est la suivante :



- La propriété **Columns** réfère à la collection des colonnes qui appartiennent à la table ; les objets de cette collection sont de type **DataColumn** ;
- La propriété **Rows** réfère à la collection des lignes qui appartiennent à cette table ; les objets de cette collection sont de type **DataRow** ;

Autrement dit, nous allons définir la structure de la table en mémoire en utilisant la collection **Columns**, puis nous manipulerons les données à l'aide de la collection **Rows**.



Pour tout savoir sur l'espace de nom **System.Data** :
[http://msdn2.microsoft.com/fr-fr/library/ax3wd0k9\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/ax3wd0k9(VS.80).aspx)



Pour tout savoir sur des **DataTable** :
[http://msdn2.microsoft.com/fr-fr/library/9186hy08\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/9186hy08(VS.80).aspx)

3.1 Créer et utiliser la table en mémoire

Dans ce premier exercice, vous allez développer le code pour créer une table vide en mémoire, et connecter cette table vide aux contrôles de données présents sur le formulaire. La structure de la table à créer, dont le nom va être **Coach**, est composée de douze champs dont la description est la suivante :

Nom	Description	Type
Id	Code d'indentification du client	string

Entreprise	Raison sociale de l'entreprise	string
Contact	Nom du contact principal du client	string
Titre	Fonction du contact principal du client	string
Adresse	Adresse de l'entreprise	string
Ville	Ville de résidence de l'entreprise	string
Region	Région de résidence de l'entreprise	string
CodePostal	Code postal du bureau postal distributeur	string
Pays	Pays de résidence de l'entreprise	string
Telephone	Numéro de téléphone du standard de l'entreprise	string
Telecopie	Numéro de la télécopie principale de l'entreprise	string
CA	Chiffre d'affaire arrondi, en millier d'euros, que vous réalisez avec cette entreprise	int

Bon, pour les puristes de la troisième forme normale, ce n'est pas très intègre, mais cela devrait suffire pour notre besoin.



Pour ceux qui souhaitent comprendre la troisième forme normale :
[http://msdn2.microsoft.com/fr-fr/library/aa200276\(offic.11\).aspx](http://msdn2.microsoft.com/fr-fr/library/aa200276(offic.11).aspx)

Déroulement de l'exercice :

1. Créez une fonction de création d'une table vide :
 - Affichez le code du formulaire **Main.cs**, en faisant un clic-droit sur **Main.cs** dans l'**Explorateur de solutions**, et en sélectionnant le menu **Afficher le code** ;
 - Positionnez le curseur juste après l'accolade } fermante du constructeur de la classe **Main** ;
 - Ajoutez deux lignes vides en tapant sur la touche **Entrée** ;
 - Créez une fonction **CreerTable** en ajoutant le code suivant :

Code C#

```
private DataTable CreerTable()
{
}

```



C'est quoi une fonction ?

Une fonction encapsule une séquence de code définissant un comportement de la classe, et en retournant une valeur/référence de retour.



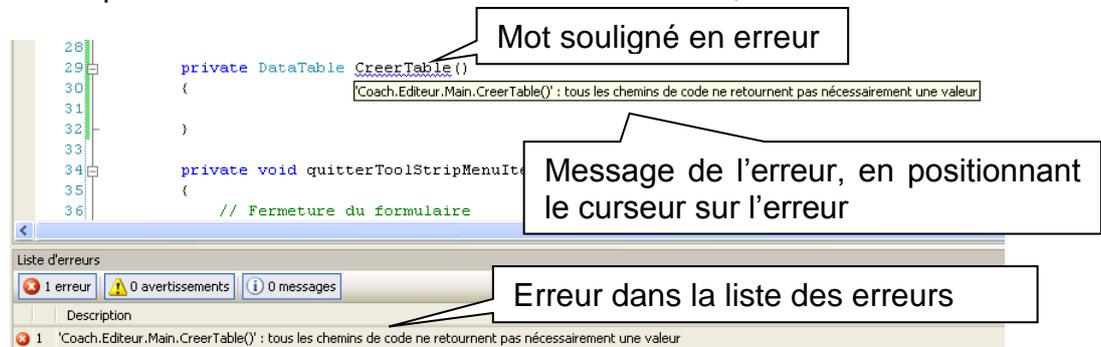
Cette fonction **CreerTable** renvoie une valeur de type **DataTable**, et est indiquée privée (**private**) car elle ne sera utilisée que dans le cadre de la classe **Main** ; l'objectif de cette fonction est donc de créer une nouvelle table et d'en retourner la référence.



Une bonne pratique pour éviter les bogues est de systématiquement sauvegarder votre travail et de lancer une compilation à chaque fois que vous travaillez sur la structure d'une classe, ou encore, à chaque fois que

vous ajoutez un bloc de code dans le contenu d'un membre de classe !

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Une erreur de compilation est affichée dans la **Liste d'erreurs** ;



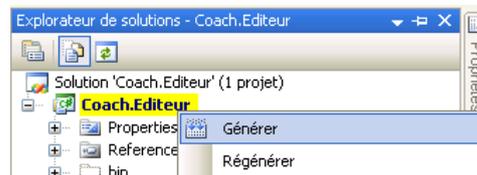
Si la fenêtre **Liste d'erreurs** ne s'affiche pas, sélectionnez le menu de Visual C# Express **Affichage > Liste d'erreurs** ;



Pour compiler l'application, il est aussi possible de sélectionner le menu de Visual C# Express **Générer > Générer la solution** ;



Pour compiler l'application, il est encore possible de faire un **clic-droit** sur le projet **Coach.Editeur** dans l'**Explorateur de solutions** et de sélectionner le menu **Générer** ;



Pour localiser une erreur dans le code, c'est simple : c'est le(s) mot(s) qui apparaît(ssent) en souligné ...



Pour aller sur la ligne de code ayant généré une erreur dans la **Liste d'erreurs**, c'est simple : faites un double-clic sur l'erreur de la liste ...



Elle vient d'où cette erreur ?

En fait le compilateur analyse chacun des membres de la classe et va marquer toutes les fonctions qui ne retournent aucune valeur assignée, voire qui ne retournent rien du tout ! Cela évite d'avoir des bouts de code pour lesquels le développeur a oublié d'indiquer la valeur de retour. Le compilateur va d'ailleurs aussi marquer les codes qui ne seront jamais exécutés (si, si cela arrive souvent d'ailleurs) en analysant les chemins possibles pour parcourir le code à l'exécution.



Dans notre cas, pour corriger l'erreur il faut donc ajouter le code de renvoi de la fonction.



Une règle de bonne programmation est de systématiquement ajouter le code de renvoi de fonction avec la création de la fonction. Pour ma part, j'utilise toujours une variable nommée **result**.

- Ajoutez à la fonction **CreerTable** les lignes de création d'une nouvelle table de nom **Coach**, et de retour de cette nouvelle table ;

Code C#

```
private DataTable CreerTable()
{
    // Création d'une table vide avec le nom Coach
    DataTable result = new DataTable("Coach");

    // retour de la valeur
    return result;
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout se compile correctement maintenant.



Nous allons maintenant créer une colonne pour chacune des colonnes à ajouter à la table en précisant le type, et l'ajouter à la liste des colonnes de la table. Le type que nous allons utiliser pour créer la colonne est **DataColumn** ;

- Ajoutez à la fonction **CreerTable** les lignes de création d'une nouvelle colonne, juste avant la directive **return** ;

Code C#

```
private DataTable CreerTable()
{
    ...

    // Création de la colonne ID
    DataColumn idColumn = new DataColumn();

    ...
}
```

- Définissez maintenant le nom et le type de cette colonne, en ajoutant les lignes suivantes juste après la création de la colonne ;

Code C#

```
private DataTable CreerTable()
{
    ...

    // Création de la colonne ID
    DataColumn idColumn = new DataColumn();
    idColumn.ColumnName = "Id";
}
```

```
idColumn.DataType = typeof(string);  
...  
}
```



Pensez à utiliser l'IntelliSense pour aller plus vite dans le codage.



C'est quoi **typeof** ?

L'opérateur **typeof** est utilisé pour obtenir l'objet correspondant à un type. En fait, la propriété **DataType** doit référencer une instance de *quelque chose* qui définit le type, mais pas le type en lui-même qui n'est pas une instance, mais juste le type de l'instance. C'est pour cela qu'en passant par **typeof(string)**, on obtient directement l'objet correspondant au type **string**. Cqfd ☺ - bon, souvenez vous que si vous avez un type à indiquer en paramètre d'une fonction, utiliser l'opérateur **typeof** ...



Pour tout savoir sur l'opérateur **typeof** :

[http://msdn2.microsoft.com/fr-fr/library/58918ffs\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/58918ffs(VS.80).aspx)



Pour tout savoir sur les autres opérateurs par mot clé de C# :

[http://msdn2.microsoft.com/fr-fr/library/bewds7kc\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/bewds7kc(VS.80).aspx)

- Ajoutez maintenant la colonne nouvellement créée **idColumn** à la liste des colonnes de la table **result**, en ajoutant les lignes suivantes ;

Code C#

```
private DataTable CreerTable()  
{  
    ...  
  
    // Création de la colonne ID  
    DataColumn idColumn = new DataColumn();  
    idColumn.ColumnName = "Id";  
    idColumn.DataType = typeof(string);  
    result.Columns.Add(idColumn);  
    ...  
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout se compile correctement.



Bon, on a du taper quatre lignes de code et une ligne de commentaire juste pour créer une colonne, ainsi que créer une variable par référence qui n'est utilisée que pour faire un ajout à la liste. Avec C#, il est possible de créer une nouvelle instance d'objet en utilisant l'opérateur **new** à tout endroit où le code attend une instance. C'est d'ailleurs pour cette raison que les types d'objets proposent généralement différents constructeurs afin de faciliter la construction des instances. Ainsi nos quatre lignes peuvent être remplacées par une seule (ce que je préfère d'ailleurs).

- Dans la fonction **CreerTable**, remplacez les quatre lignes de création de la colonne **idColumn** par la ligne suivante ;

Code C#

```
private DataTable CreerTable()
{
    ...

    // Création de la colonne ID
    DataColumn idColumn = new DataColumn();
    idColumn.ColumnName = "Id";
    idColumn.DataType = typeof(string);
    result.Columns.Add(idColumn);
    result.Columns.Add(new DataColumn("Id", typeof(string)));
    ...
}
```

- Dans la fonction **CreerTable**, ajoutez maintenant le code de création des onze autres colonnes ;

Code C#

```
private DataTable CreerTable()
{
    ...

    // Création de la colonne ID
    result.Columns.Add(new DataColumn("Id", typeof(string)));
    result.Columns.Add(new DataColumn("Entreprise", typeof(string)));
    result.Columns.Add(new DataColumn("Contact", typeof(string)));
    result.Columns.Add(new DataColumn("Titre", typeof(string)));
    result.Columns.Add(new DataColumn("Adresse", typeof(string)));
    result.Columns.Add(new DataColumn("Ville", typeof(string)));
    result.Columns.Add(new DataColumn("Region", typeof(string)));
    result.Columns.Add(new DataColumn("CodePostal", typeof(string)));
    result.Columns.Add(new DataColumn("Pays", typeof(string)));
    result.Columns.Add(new DataColumn("Telephone", typeof(string)));
    result.Columns.Add(new DataColumn("Telecopie", typeof(string)));
    result.Columns.Add(new DataColumn("CA", typeof(int)));
    ...
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement.

2. Liez la table aux contrôles de données :



L'idée est de lier la table nouvellement créée aux différents contrôles de données du formulaire lorsque l'utilisateur va cliquer sur le menu **Fichier > Nouveau** de l'**Editeur du Coach C#** ; La liaison des données aux contrôles est ce que l'on appelle le **DataBinding** .



C'est quoi le DataBinding ?

Le DataBinding est le mécanisme par lequel la valeur (donnée - « data ») d'une propriété d'un contrôle est automatiquement maintenue à jour (liée – « binding ») à la valeur d'une propriété d'un objet fournissant des données (« datasource »). L'originalité du DataBinding est que n'importe quel objet peut fournir une ou des données, et il est donc par exemple possible, avec le DataBinding, de lier automatiquement la valeur de la propriété **Text** ou **Tag** d'un bouton avec la propriété **Text** d'une boîte de texte (**TextBox**).



Comment ça marche, le DataBinding ?

Le fonctionnement du **DataBinding** peut être schématisé comme suit :



Un objet intervient comme un gestionnaire de liaison, et il va automatiquement refléter et maintenir à jour les valeurs des propriétés qui sont préalablement définies comme étant en liaison. Dans notre cas, nous allons utiliser un objet **BindingSource** pour accéder au gestionnaire de liaisons et gérer la source de toutes les liaisons (**Bindings**) dont nous allons avoir besoin.



Vous trouverez sur ce lien une vidéo d'introduction au DataBinding :
<http://www.microsoft.com/france/vision/WebcastMsdn.aspx?EID=7972b8bd-5208-461e-ac3e-949acff98d74>



Et sur ce lien une vidéo d'anthologie à ne pas manquer sur le DataBinding :
<http://www.microsoft.com/france/vision/WebcastMsdn.aspx?EID=5c92551b-6cee-49e2-89c3-1604d1511882>



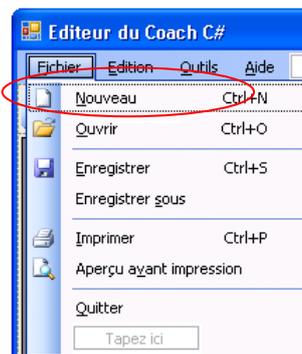
Si vous développez des applications qui manipulent des données, vous devez *absolument* comprendre et utiliser le DataBinding (c'est plus rapide avec beaucoup moins de lignes de code), en lieu et place des mécanismes que nous utilisons tous auparavant, essentiellement basés sur le développement spécifique de code de liaison pour chacun des contrôles.

- Revenez sur le formulaire **Main.cs** en mode **[Design]** (soit en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail, soit en double-cliquant sur le fichier **Main.cs** dans l'**Explorateur de solutions**) ;

- Dans la boîte à outils, ouvrez l'onglet **Données**, s'il est fermé, en cliquant sur  qui précède le nom de l'onglet ;
- Faites un **glisser-déplacer** du contrôle **BindingSource** sur la zone de dépôt de contrôles du formulaire. Un contrôle **bindingSource1** apparaît dans cette zone de dépôt ;
- **Sélectionnez** le contrôle de source de données **bindingSource1** ;
- Dans la fenêtre de **Propriétés**, configurez le nom de ce contrôle en changeant la propriété **(Name)** avec **mainBindingSource** ;



- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement.
- Dans la fenêtre **Main.cs** en mode **[Design]**, faites un **double-clic** sur le menu **Fichier > Nouveau** ;



- Une méthode `nouveauToolStripMenuItem_Click` de réponse au menu **Nouveau** a été ajoutée au code du formulaire **Main.cs** ;
- Dans la méthode `nouveauToolStripMenuItem_Click`, ajoutez le code configurant la source de données (**DataSource**) des liaisons (**mainBindingSource**) avec une nouvelle table ;

Code C#

```
private void nouveauToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Charge une table vide dans la source des bindings
    mainBindingSource.DataSource = CreerTable();
}
```

- Dans la méthode `nouveauToolStripMenuItem_Click`, ajoutez le code indiquant que le barre de navigation (**mainBindingNavigator**) navigue (logique !) dans la source de liaisons tout juste définie ;

Code C#

```
private void nouveauToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Charge une table vide dans la source des bindings
    mainBindingSource.DataSource = CreerTable();
    // Configure la navigation
    mainBindingNavigator.BindingSource = mainBindingSource;
}

```

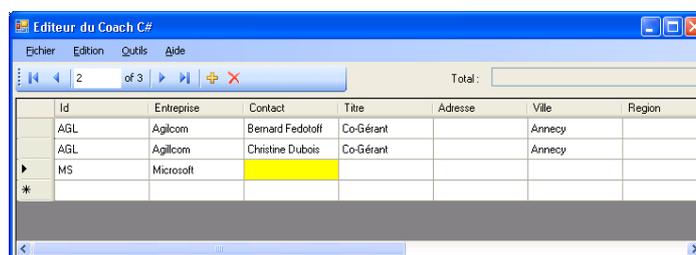
- Enfin, dans la méthode `nouveauToolStripMenuItem_Click`, ajoutez le code liant la grille de données (**mainDataGridView**) avec la source de données de la source des liaisons (**BindingSource**) dans laquelle navigue la barre de navigation (**mainBindingNavigator**) ;

Code C#

```
private void nouveauToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Charge une table vide dans la source des bindings
    mainBindingSource.DataSource = CreerTable();
    // Configure la navigation
    mainBindingNavigator.BindingSource = mainBindingSource;
    // Configure la grille de données sur la même source que la source de
    // navigation
    mainDataGridView.DataSource = mainBindingNavigator.BindingSource;
}

```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement.
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ;
- Dans l'**Editeur du Coach C#**, cliquez le menu **Fichier > Nouveau** ; Une nouvelle ligne de donnée apparaît dans le formulaire ;
- A l'aide des boutons de la barre de navigation, **ajoutez** () et **détruisez** () quelques lignes de données ; Amusez-vous à saisir des informations : la grille fonctionne ! Franchement, vous êtes des As.



Pour tout savoir sur l'objet **BindingSource** :

[http://msdn2.microsoft.com/fr-fr/library/h974h4y2\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/h974h4y2(VS.80).aspx)



Pour tout savoir sur l'objet **BindingNavigator** :

[http://msdn2.microsoft.com/fr-fr/library/b9y7cz6d\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/b9y7cz6d(VS.80).aspx)



Pour tout savoir sur l'objet **DataGridView** :

[http://msdn2.microsoft.com/fr-fr/library/e0ywh3cz\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/e0ywh3cz(VS.80).aspx)

3.2 Alimenter la table en mémoire avec le fichier CSV

L'objectif de ce deuxième exercice est de lire les données de la grille à partir d'un fichier au format CSV (texte délimité), et bien sûr de sauvegarder de nouveau ces données, après modification, dans le même fichier CSV (ou un autre d'ailleurs) ! Il va nous donner l'opportunité d'aborder certains mots clés de test, de boucle et de bloc.

Déroulement de l'exercice :

1. Développez le code de lecture du fichier CSV :



Un fichier CSV est composé de lignes de valeurs, séparées par un séparateur qui dans notre cas est le « ; », et dont l'ordre des valeurs est toujours le même d'une ligne à l'autre. Chaque ligne est finie par un retour-chariot (**CR**).



L'algorithme à développer est composé de deux étapes :

1. Lire le fichier CSV ligne à ligne ;
2. A partir de la ligne de texte lue, générer un enregistrement dans la table de données ; Dans la ligne lue, chacune des valeurs à lire est séparée par un « ; » ; le premier paramètre est l'**ID**, le deuxième l'**Entreprise**, etc. jusqu'au douzième qui est le **Chiffre d'affaire** réalisé ;



Afin de générer un nouvel enregistrement vide, nous allons utiliser la méthode **NewRow()** de notre table de données. Cette méthode est fournie par la classe **DataTable**, qui est le type de notre objet de table de données en mémoire.



Une bonne pratique de développement est de créer une méthode ou fonction par étape d'algorithme, et ce afin d'augmenter la maintenabilité et la lisibilité du code. Autrement dit, nous allons développer deux méthodes : une pour lire le fichier ligne à ligne, et une pour ajouter à notre table de données une nouvelle ligne. C'est par cette dernière que nous allons commencer !

- Affichez le code du formulaire **Main.cs**, en faisant un **clic-droit** sur **Main.cs** dans l'**Explorateur de solutions**, et en sélectionnant le menu **Afficher le code** ;
- Positionner le curseur juste après l'accolade } fermante du constructeur de la classe Main ;
- Ajoutez deux lignes vides en tapant sur la touche **Entrée** ;
- Créez une fonction **AjouterLigneATable** en ajoutant le code suivant :

Code C#

```
private void AjouteLigneATable(string LigneLue, DataTable TableDeDonnees)
```

```
{  
}
```



C'est quoi **void** ?

void indique au compilateur que la méthode est une fonction qui ne retourne aucune valeur.



C'est quoi (**string** `LigneLue`, **DataTable** `TableDeDonnees`) ?

En fait, une fonction peut avoir zéro ou plusieurs paramètres, qui indique(nt) généralement à la fonction le contexte d'exécution de celle-ci. Par exemple, le premier paramètre **string** `LigneLue` est la ligne qui a été lue dans le fichier CSV, alors que le deuxième **DataTable** `TableDeDonnees` est la table de données dans laquelle ajouter un nouvel enregistrement. Le type précède le nom du paramètre.



LigneLue Et **TableDeDonnees** :

- **LigneLue** correspond à ligne de donnée au format texte délimité avec un « ; » qu'il faut couper pour ajouter une nouvelle ligne de données ;
- **TableDeDonnees** est la table de données sur laquelle il faut ajouter une nouvelle ligne ;



Passage par valeur ou **passage par référence** ?

En C#, les paramètres sont passés par défaut par valeur, c'est-à-dire que la valeur est copiée sur la pile et que tous les changements de cette valeur sont détruits lorsque la méthode est quittée. Néanmoins, C# utilise communément deux types : les types par valeur et les types par référence (cf. page 26). Dans le cas de passage de paramètre d'un type par valeur (par exemple **int** ou **double**), le paramètre est donc copié et les changements réalisés sur la valeur de ces paramètres sont détruits lors du retour de fonction. Un paramètre ayant un type par référence (tout objet) est aussi passé par valeur, mais dans ce cas c'est une copie de la référence (qui représente la localisation de l'objet en mémoire) qui est ajoutée à la pile. La méthode utilise donc un alias sur l'objet, qui va être utilisé pour modifier cet objet (c'est un peu comme si, lors de l'appel de la fonction, on ajoutait sur une pièce une deuxième porte d'accès). Quand la fonction va se terminer, la référence utilisée par la fonction va être détruite (la deuxième porte est détruite) mais les modifications apportés à l'objet seront gardés (en gros, les meubles que vous aurez ajouté en passant par la porte juste détruite restent dans la pièce). C'est pour cela que le paramètre **DataTable** `TableDeDonnees` est passé par valeur, mais tous les ajouts d'enregistrements qui seront fait dans la fonction `AjouteLigneATable` seront préservés lors de la fin de cette fonction.



Pour tout savoir sur le passage de paramètres :

[http://msdn2.microsoft.com/fr-fr/library/0f66670z\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/0f66670z(VS.80).aspx)

- Ajoutez à la fonction **AjouterLigneATable** la ligne de création de l'index que nous utiliserons pour indiquer la valeur lue dans la ligne (Id, Entreprise, etc.) ;

Code C#

```
private void AjouterLigneATable(string LigneLue, DataTable TableDeDonnees)
{
    // Definition de l'index de la position de lecture
    int index = 0;
}
```

- Ajoutez à la fonction **AjouterLigneATable** le code pour découper la ligne de texte lue en un tableau de chaînes de caractères, en fonction du séparateur « ; » ; La méthode à utiliser est fournie par la classe **string** qui est le type de la variable **LigneLue** ;

Code C#

```
private void AjouterLigneATable(string LigneLue, DataTable TableDeDonnees)
{
    // Definition de l'index de la position de lecture
    int index = 0;
    // Découper la ligne en fonction du caractère de séparation
    string[] valeursLues = LigneLue.Split(new char[] { ';' });
}
```

- Ajoutez à la fonction **AjouterLigneATable** le code pour générer une nouvelle ligne de données vide ; La méthode à utiliser est fournie par la classe **DataTable** qui est le type de la variable **TableDeDonnees** ;

Code C#

```
private void AjouterLigneATable(string LigneLue, DataTable TableDeDonnees)
{
    // Definition de l'index de la position de lecture
    int index = 0;
    // Découper la ligne en fonction du caractère de séparation
    string[] valeursLues = LigneLue.Split(new char[] { ';' });
    // Crée une nouvelle ligne de données
    DataRow ligneDeDonnees = TableDeDonnees.NewRow();
}
```

- Ajoutez à la fonction **AjouterLigneATable** le code pour balayer l'ensemble des valeurs contenues dans le tableau **valeursLues** (Pensez à utiliser un extrait de code pour le faire) ;

Code C#

```
private void AjouterLigneATable(string LigneLue, DataTable TableDeDonnees)
{
    // Definition de l'index de la position de lecture
    int index = 0;
    // Découper la ligne en fonction du caractère de séparation
    string[] valeursLues = LigneLue.Split(new char[] { ';' });
    // Crée une nouvelle ligne de données
    DataRow ligneDeDonnees = TableDeDonnees.NewRow();
    // Pour toute les valeurs, met à jour le ligne de données
}
```

```
foreach (string valeur in valeursLues)
{
}
}
```



Au fait, comment utiliser les extraits de code ?

Bon, on l'a déjà vu, mais on va se répéter !

1. La première solution est de faire un clic-droit à l'endroit où vous voulez insérer un extrait de code, et sélectionnez le menu **Insérer un extrait...** ;

```
DataRow ligneDeDonnees = TableDeDonnees.NewRow();
}
}
```

Dans notre cas, sélectionnez ensuite l'extrait **Collections and arrays > Iterate through a collection** ;

```
DataRow ligneDeDonnees = TableDeDonnees.NewRow();
Insérer un extrait: collections and arrays >
// Crée une liste d'éléments de type unique.
// Crée un dictionnaire trié.
// Définit un comparateur de type à utiliser dans les routines de tri.
// Indexe dans un dictionnaire fortement typé en utilisant une clé.
// Initialise un tableau unidimensionnel.
// Itère à travers une collection.
// Itère à travers un dictionnaire.
// Itère à travers un dictionnaire trié.
```

Il vous reste ensuite à paramétrer l'extrait et à ajouter le commentaire ;

2. La deuxième solution est de taper **foreach** dans le code ; L'éditeur de Visual C# Express vous propose l'IntelliSense, en présélectionnant **foreach** ;

```
DataRow ligneDeDonnees = TableDeDonnees.NewRow();
foreach
{
}
}
```

Tapez alors deux fois sur la touche **TAB** (tabulation) ;



C'est quoi **foreach** ?

foreach est une instruction à utiliser pour itérer automatiquement tous les éléments d'une collection ou d'un tableau.



Pour tout savoir sur **foreach** :

[http://msdn2.microsoft.com/fr-fr/library/ttw7t8t6\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/ttw7t8t6(VS.80).aspx)

- Ajoutez à la fonction **AjouterLigneATable** le code de mise à jour de la colonne de la ligne à ajouter, en fonction de la valeur lue ;

Code C#

```
private void AjouterLigneATable(string LigneLue, DataTable TableDeDonnees)
{
    ...
    // Pour toute les valeurs, met à jour le ligne de données
    foreach (string valeur in valeursLues)
    {
        switch (index++)
        {
            case 0:
                ligneDeDonnees["Id"] = valeur.Trim();
                break;
            case 1:
                ligneDeDonnees["Entreprise"] = valeur.Trim();
                break;
            case 2:
                ligneDeDonnees["Contact"] = valeur.Trim();
                break;
            case 3:
                ligneDeDonnees["Titre"] = valeur.Trim();
                break;
            case 4:
                ligneDeDonnees["Adresse"] = valeur.Trim();
                break;
            case 5:
                ligneDeDonnees["Ville"] = valeur.Trim();
                break;
            case 6:
                ligneDeDonnees["Region"] = valeur.Trim();
                break;
            case 7:
                ligneDeDonnees["CodePostal"] = valeur.Trim();
                break;
            case 8:
                ligneDeDonnees["Pays"] = valeur.Trim();
                break;
            case 9:
                ligneDeDonnees["Telephone"] = valeur.Trim();
                break;
            case 10:
                ligneDeDonnees["Telecopie"] = valeur.Trim();
                break;
            case 11:
                ligneDeDonnees["CA"] = valeur.Trim();
                break;
            default:
                ligneDeDonnees[string.Format("Colonne {0}", index)]
                    = valeur.Trim();
                break;
        }
    }
}
```



C'est quoi **switch** ?

switch est une instruction de sélection afin d'évaluer une expression et de déterminer le bloc de code à exécuter ; **switch** saute automatiquement à un libellé qui correspond à l'expression, pour exécuter le bloc de code finissant nécessairement par **break** ;



Pour tout savoir sur **switch** :

[http://msdn2.microsoft.com/fr-fr/library/06tc147t\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/06tc147t(VS.80).aspx)

- Enfin, juste avant **}** fermante de la méthode, ajoutez à la fonction **AjouterLigneATable** le code d'ajout de la ligne de données nouvellement créée ;

Code C#

```
private void AjouterLigneATable(string LigneLue, DataTable TableDeDonnees)
{
    ...
    // Pour toute les valeurs, met à jour le ligne de données
    foreach (string valeur in valeursLues)
    {
        switch (index++)
        {
            ...
        }
    }

    // Ajoute la nouvelle ligne de données à la table
    TableDeDonnees.Rows.Add(ligneDeDonnees);
}
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement.



Bon, maintenant que nous savons ajouter une nouvelle ligne de données à partir de la ligne de texte lue, il faut lire le fichier texte ligne à ligne et appeler la fonction **AjouterLigneATable** pour chacune des lignes de texte !

- Affichez le code du formulaire **Main.cs**, en faisant un clic-droit sur **Main.cs** dans l'**Explorateur de solutions**, et en sélectionnant le menu **Afficher le code** ;
- Positionnez le curseur juste après l'accolade **}** fermante du constructeur de la classe **Main** ;
- Ajoutez deux lignes vides en tapant sur la touche **Entrée** ;
- Créez une fonction **OuvrirFichier** en ajoutant le code suivant :

Code C#

```
private DataTable OuvrirFichier(string NomDuFichier)
{
}
}
```



La fonction `ouvrirFichier` possède un seul paramètre, `NomDuFichier` qui est le nom complet du fichier à ouvrir et renvoie une table de données complète de type `DataTable`, contenant tous les enregistrements lus;



Encore une fois, une règle de bonne programmation est de systématiquement ajouter le code de renvoi de fonction avec la création de la fonction. Pour ma part, j'utilise toujours une variable nommée **result** (bon, je radote un peu mais cela évite des bogues). Et comme l'assignation est obligatoire, assignez à **null** vos variables de retour de type objet, si vous n'avez pas d'autres valeurs à donner (logique).

- Ajoutez à la fonction **OuvrirFichier** le code de définition et de renvoi de la valeur de retour ;

Code C#

```
private DataTable OuvrirFichier(string NomDuFichier)
{
    // definition de la variable de retour
    DataTable result = null;

    // renvoi de la valeur
    return result;
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ; La structure de la fonction est correcte.



Nous avons besoin de définir deux variables :

1. **ligneLu**, de type **string**, qui correspond à la ligne de fichier lue et qui est passée en paramètre à la fonction **AjouterLigneATable** ;
2. **estPremièreLigne**, de type **booléen**, qui indiquera si nous lisons la première ligne, car dans ce cas il faudra assigner la DataTable devant recevoir les enregistrements qui est passée en paramètre de la méthode **AjouterLigneATable** ;

- Ajoutez à la fonction **OuvrirFichier** le code de définition des variables **ligneLu** et **estPremièreLigne** ;

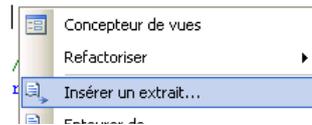
Code C#

```
private DataTable OuvrirFichier(string NomDuFichier)
{
    // definition de la variable de retour
    DataTable result = null;
    // Definition et assignation des variables
    string ligneLu = string.Empty;
    Boolean estPremiereLigne = true;

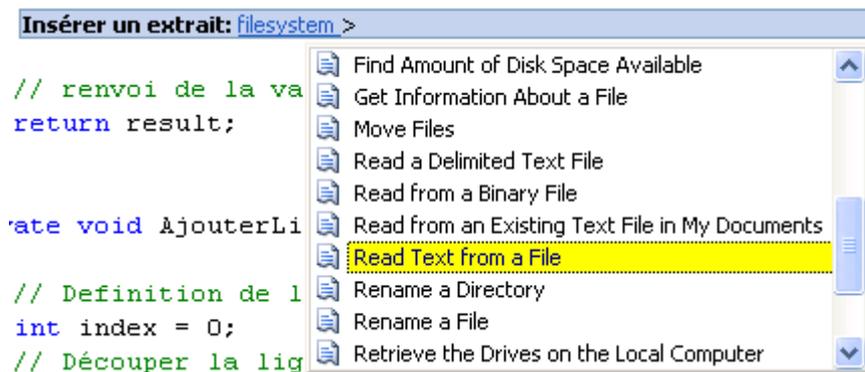
    // renvoi de la valeur
    return result;
}
```

- Sous la dernière ligne insérée, faites un **clic-droit** et sélectionnez le menu **Insérer un extrait...** ;

```
Boolean estPremiereLigne = true;
```



- Sélectionnez l'extrait de code **filesystem > Read Text From a File** ;



Le code suivant est ajouté à la fonction :

Code C#

```
private DataTable OuvrirFichier(string NomDuFichier)
{
    ...
    Boolean estPremiereLigne = true;

    string fileContents;
    using (System.IO.StreamReader sr =
        new System.IO.StreamReader(@"C:\Test.txt"))
    {
        fileContents = sr.ReadToEnd();
    }

    // renvoi de la valeur
    return result;
}
```



Dans l'extrait de code ajouté apparaît des parties en vert ; ce sont les éléments à paramétrer de l'extrait ;

Elément à définir pour l'extrait

```
new System.IO.StreamReader(@"C:\Test.txt")
```

Dans notre cas, il s'agit de définir le nom du fichier à ouvrir ;



C'est quoi @ devant la chaîne de caractère ?

Le caractère @ devant une chaîne de caractère indique qu'il faut traiter les caractères spéciaux (« \ » dans notre cas) comme un caractère normal.



C'est quoi **using** ?

using définit la portée d'une variable en dehors de laquelle elle sera supprimée. Bon, si ce n'est pas très clair, voici l'explication de la portée et de la visibilité des variables. La portée d'une variable définit l'étendue de code durant laquelle le nom d'une variable ne peut pas être réutilisé pour nommer une seconde variable. La visibilité d'une variable définit l'étendue de code durant laquelle le contenu de la variable peut être accédé en utilisant son nom (ou alias). Le petit schéma suivant illustre la différence entre la portée et la visibilité :

```
private DataTable OuvrirFichier(string NomDuFichier)
{
    // definition de la variable de retour
    DataTable result = null;
    // Definition et assignation des variables
    string ligneLu = string.Empty;
    Boolean estPremiereLigne = true;

    string fileContents;
    using (System.IO.StreamReader sr
        = new System.IO.StreamReader(@"C:\Test.txt"))
    {
        fileContents = sr.ReadToEnd();
    }

    // renvoi de la valeur
    return result;
}
```

Diagram illustrating the scope and visibility of variables in the code:

- Portée de :** NomDuFichier, result, ligneLu, estPremiereLigne, fileContents
- Visibilité de :** ligneLu
- Portée et Visibilité de :** sr

Donc, le grand intérêt de **using** est de limiter la portée des variables uniquement à un bloc de code déterminé.



Pour tout savoir sur **using** :

[http://msdn2.microsoft.com/fr-fr/library/yh598w02\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/yh598w02(VS.80).aspx)



C'est quoi un **StreamReader** ?

C'est un objet qui lit des caractères à partir d'un flux d'octets dans un codage particulier. C'est l'objet à utiliser pour lire des lignes d'informations à partir d'un fichier texte standard. Il appartient à l'espace de nommage **System.IO**, qui fournit tous les objets nécessaires pour réaliser tout type d'Entrées/Sorties avec le système. La méthode du **StreamReader** que nous allons utiliser pour lire une ligne de donnée texte est **ReadLine()** ;



Pour tout savoir sur **StreamReader** :

[http://msdn2.microsoft.com/fr-fr/library/6aetdk20\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/6aetdk20(VS.80).aspx)

- Dans la fonction **OuvrirFichier**, retirez le code de l'extrait qui nous est inutile, et indiquez que le nom du fichier à ouvrir est le paramètre **NomDuFichier** ;

Code C#

```
private DataTable OuvrirFichier(string NomDuFichier)
{
    ...
    Boolean estPremiereLigne = true;
    string fileContents;
}
```

```

using (System.IO.StreamReader sr =
    new System.IO.StreamReader(NomDuFichier))
{
    fileContents = sr.ReadToEnd();
}

// renvoi de la valeur
return result;
}

```



Reste maintenant à lire ligne à ligne le fichier texte délimité, en utilisant une instruction de boucle ;



C# propose différentes instructions de boucle dont l'objectif commun est d'exécuter d'une manière itérative un bloc de code :

1. **for**, qui est à utiliser dans le cas où un numéro de compteur est disponible ;
2. **while**, qui est à utiliser dans le cas où une itération est à exécuter tant que (« while » en anglais) une expression est vraie, ce qui veut dire que la boucle peut ne pas être exécutée si l'expression est fautive ;
3. **do**, c'est un peu comme un **while**, mais elle est à utiliser dans le cas où l'expression est à évaluer après que la boucle ait été exécutée au moins une fois ; On est alors sûr de passer au moins une fois dans le bloc de code ;
4. **foreach**, qui est à utiliser pour itérer automatiquement les éléments d'une collection ou d'une liste ;



Pour tout savoir sur les instructions d'itération :

[http://msdn2.microsoft.com/fr-fr/library/32dbftby\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/32dbftby(VS.80).aspx)



Dans notre cas, nous allons utiliser un **do**, car durant la première boucle nous devons assigner la table de données à une nouvelle valeur. Puis nous ajouterons une nouvelle ligne à la table de données, tout ceci si la valeur lue n'est pas nulle bien sûr !

- Ajoutez à la fonction **OuvrirFichier** le code de lecture de chacune des lignes ;

Code C#

```

private DataTable OuvrirFichier(string NomDuFichier)
{
    ...
    using (System.IO.StreamReader sr
        = new System.IO.StreamReader(NomDuFichier))
    {
        do
        {
            // Pour chaque ligne lu
            ligneLu = sr.ReadLine();
        } while (ligneLu != null);
    }
    ...
}

```



If faut maintenant tester la valeur de la ligne lue et vérifier si cette ligne est la première ligne lue. Nous allons utiliser l'instruction de sélection **if-else** ;



C'est quoi **if-else** ?

L'instruction **if** permet de sélectionner une instruction à exécuter en fonction de la valeur d'une expression **Booléenne**.



Pour tout savoir sur **if-else** :

[http://msdn2.microsoft.com/fr-fr/library/5011f09h\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/5011f09h(VS.80).aspx)

- Sous la lecture de la ligne, ajoutez à la fonction **OuvrirFichier** le code de test créant la table de données et ajoutant une nouvelle ligne à celle-ci ;

Code C#

```
private DataTable OuvrirFichier(string NomDuFichier)
{
...
    using (System.IO.StreamReader sr
        = new System.IO.StreamReader(NomDuFichier))
    {
        do
        {
            // Pour chaque ligne lu
            ligneLu = sr.ReadLine();

            // si c'est la première ligne lue,
            // alors créé la table de donnée
            if (estPremiereLigne && ligneLu != null)
            {
                result = CreerTable();
                estPremiereLigne = false;
            }

            // Si la ligne de données n'est pas nulle,
            // alors ajoute la ligne à la table de données
            if (ligneLu != null)
                AjouterLigneATable(ligneLu, result);

        } while (ligneLu != null);
    }
...
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ;



La dernière étape reste la sélection et l'ouverture du fichier sur le disque. Le Framework .Net propose des boîtes de dialogue communes à toutes les applications pour les tâches les plus courantes : l'ouverture et la sauvegarde de fichier, la sélection de couleurs ou de polices(font), ou encore la sélection de répertoires. Ces boîtes de dialogues communes

sont disponibles soit au travers de contrôles d'écran sous la section **Boîte de dialogues** dans la **Boîte à outils**, ou directement à partir de classes pour instancier vos propres objets.



Pour ouvrir un fichier sur le disque, nous allons utiliser un objet de type **OpenFileDialog** ; Les principales propriétés et méthodes que nous allons utiliser sont :

1. **Filter** : Cette propriété sert à filtrer les fichiers en se basant notamment sur l'extension ; nous filtrerons tous les fichiers d'extension ***.coach** ;
 2. **InitialDirectory** : Cette propriété configure le répertoire affiché lors de l'ouverture de la fenêtre ;
 3. **FileName** : Cette propriété obtient ou définit le nom complet du fichier sélectionné dans la liste des fichiers proposés ;
 4. **ShowDialog** : Cette méthode affiche la boîte de dialogue, et retourne une valeur de type **DialogResult** indiquant notamment le bouton que l'utilisateur a cliqué pour sortir de la fenêtre ; Nous testerons ici que l'utilisateur a bien cliqué le bouton **OK** ;
- Dans la fenêtre **Main.cs** en mode **[Design]**, faites un **double-clic** sur le menu **Fichier > Ouvrir** ;
 - Une méthode `ouvrirToolStripMenuItem_Click` de réponse au menu **Ouvrir** a été ajoutée au code du formulaire **Main.cs** ;
 - Dans la méthode `ouvrirToolStripMenuItem_Click`, ajoutez le code de création et d'affichage de la boîte de dialogue d'ouverture de fichier ;

Code C#

```
private void ouvrirToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (OpenFileDialog fileOpen =
        new System.Windows.Forms.OpenFileDialog())
    {
        fileOpen.Filter = "Fichiers coach|*.coach";
        fileOpen.InitialDirectory = @"c:\";

        if (fileOpen.ShowDialog() == DialogResult.OK)
        {

        }
    }
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ; Vous pouvez d'ailleurs aussi tester le bon fonctionnement de la boîte de dialogue en lançant l'application (ça ne coûte rien et c'est visuel ...) ;
- Dans la méthode `ouvrirToolStripMenuItem_Click`, ajoutez le code d'ouverture du fichier et de configuration des sources de données ; C'est exactement le même code que celui écrit dans la méthode `nouveauToolStripMenuItem_Click` (cf. page 58), à l'exception que la

table de données est la valeur de retour de l'appel de la fonction **OuvrirFichier** ;

Code C#

```
private void ouvrirToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (OpenFileDialog fileOpen =
        new System.Windows.Forms.OpenFileDialog())
    {
        fileOpen.Filter = "Fichiers coach|*.coach";
        fileOpen.InitialDirectory = @"c:\";

        if (fileOpen.ShowDialog() == DialogResult.OK)
        {
            // Charge le fichier de données dans la source des bindings
            mainBindingSource.DataSource =
                OuvrirFichier(fileOpen.FileName);
            // Configure la navigation
            mainBindingNavigator.BindingSource = mainBindingSource;
            // Configure la grille de données sur la même source
            // que la source de navigation
            mainDataGridView.DataSource
                = mainBindingNavigator.BindingSource;
        }
    }
}
```



Il faut maintenant mémoriser le nom du fichier sélectionné pour l'utiliser lors de la sauvegarde. Pour ce faire, vous allez ajouter une variable **nomFichier**, de type **string**, et dont la portée va être la totalité de la classe **Main** (pour la voir de partout dans la classe);

- Positionnez le curseur juste avant la définition du constructeur de la classe **Main** ;
- Ajoutez deux lignes vides en tapant sur la touche **Entrée** ;
- Ajoutez le code suivant pour définir un membre privé (variable) de portée sur la totalité de la classe pour stocker le nom du fichier ;

Code C#

```
public partial class Main : Form
{
    // Membres privés
    private string nomFichier = string.Empty;

    public Main()
    {
        ...
    }
}
```

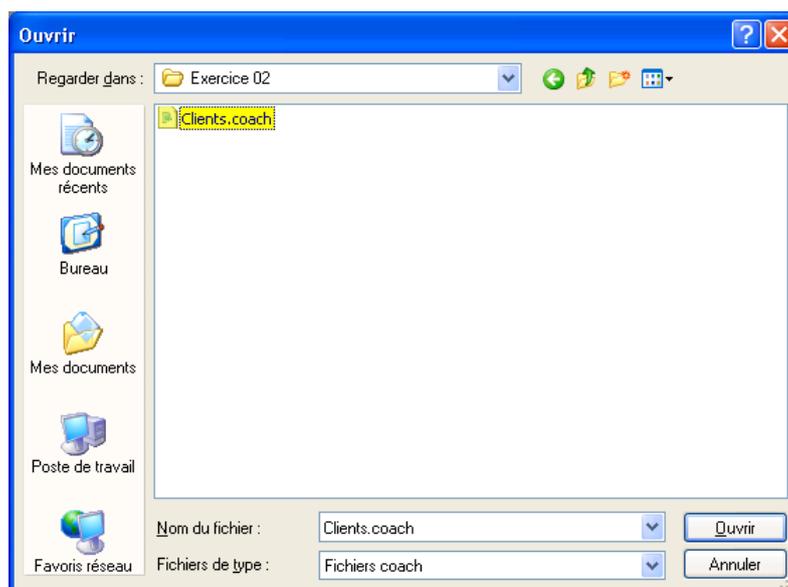
- Revenez maintenant dans le code de la méthode d'ouverture de fichier `ouvrirToolStripMenuItem_Click` ;
- Dans la méthode `ouvrirToolStripMenuItem_Click`, ajoutez le code de sauvegarde du nom du fichier ;

Code C#

```
private void ouvrirToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (OpenFileDialog fileOpen =
        new System.Windows.Forms.OpenFileDialog())
    {
        fileOpen.Filter = "Fichiers coach|*.coach";
        fileOpen.InitialDirectory = @"c:\\";

        if (fileOpen.ShowDialog() == DialogResult.OK)
        {
            // Charge le fichier de données dans la source des bindings
            mainBindingSource.DataSource =
                OuvrirFichier(fileOpen.FileName);
            // Configure la navigation
            mainBindingNavigator.BindingSource = mainBindingSource;
            // Configure la grille de données sur la même source
            // que la source de navigation
            mainDataGridView.DataSource
                = mainBindingNavigator.BindingSource.DataSource;
            // Mémorise le nom du fichier
            nomFichier = fileOpen.FileName;
        }
    }
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ;
- Dans l'**Editeur du Coach C#**, cliquez le menu **Fichier > Ouvrir** ; La boîte de dialogue **Ouvrir** d'ouverture de fichier apparaît ;



- Sélectionnez le fichier d'exemple **Clients.coach** situé dans les fichiers utiles de l'exercice 02 (par exemple **C:\Coach C#\Atelier 02\Fichiers Utiles\Exercice 02\Clients.coach**) ;
- Dans la boîte de dialogue **Ouvrir**, cliquez le bouton **OK** ; l'éditeur du coach s'affiche maintenant avec l'ensemble des lignes lues ;

	Id	Entreprise	Contact	Titre	Adresse	Ville	Region
▶	ALFKI	Alfreds Futterkiste	Sales Represent...	Obere Str. 57	Berlin		12209
	ANATR	Ana Trujillo Empa...	Owner	Avda. de la Cons...	México D.F.		05021
	ANTON	Antonio Moreno ...	Owner	Mataderos2312	México D.F.		05023
	AROUT	Around the Horn	Sales Represent...	120 Hanover Sq.	London		WA1 1DP
	BERGS	Berglunds snabb...	Order Administrator	Berguvsvägen8	Luleå		S-958 22
	BLAUS	Blauer See Delik...	Sales Represent...	Forsterstr. 57	Mannheim		68306
	BLONP	Blondesddsl père...	Marketing Manager	24, place Kléber	Strasbourg		67000
	BOLID	Bóldo Comidas p...	Owner	C/ Araquil, 67	Madrid		28023
	BONAP	Bon app'	Owner	12, rue des Bouc...	Marseille		13008
	BOTTM	Bottom-Dollar Ma...	Accounting Man...	23 Tsawassen Bl...	Tsawassen	BC	T2F 8M4
	BSBEV	B's Beverages	Sales Represent...	Fauntleroy Circus	London		EC2 5NT
	CACTU	Cactus Comidas ...	Sales Agent	Cerrito 333	Buenos Aires		1010
	CENTC	Centro comercial ...	Marketing Manager	Sierras de Grana...	México D.F.		05022

Bon, il n'y a pas à dire, bien que la programmation soit « à l'ancienne », ça le fait tout de même !

2. Développez le code d'écriture du fichier CSV :



Ici, on va programmer l'inverse de l'exercice précédent, c'est-à-dire que nous allons programmer la sauvegarde des informations de la grille au format CSV sur le disque.



Nous allons tout d'abord faire une fonction de sauvegarde des informations de la grille dans un fichier dont le nom est fourni en paramètre ;

- Affichez le code du formulaire **Main.cs**, en faisant un clic-droit sur **Main.cs** dans l'**Explorateur de solutions**, et en sélectionnant le menu **Afficher le code** ;
- Positionner le curseur juste après l'accolade **}** fermante du constructeur de la classe **Main** ;
- Ajoutez deux lignes vides en tapant sur la touche **Entrée** ;
- Créez une fonction **sauverFichier** en ajoutant le code suivant :

Code C#

```
private void SauverFichier(string NomDuFichier)
{
}
}
```



La fonction `sauverFichier` ne possède qu'un seul paramètre, `NomDuFichier` qui est le nom complet du fichier à sauvegarder, et ne renvoie aucune valeur de retour (d'où `void`)



Pour lire un fichier, nous avons utilisé un **StreamReader**. Pour l'écrire, nous allons utiliser un ?? **StreamWriter**, bien sûr !



C'est quoi un **StreamWriter** ?

C'est un objet qui écrit des caractères à partir d'un flux d'octets dans un codage particulier. C'est l'objet à utiliser pour écrire dans un fichier texte des lignes de texte. Tout comme le **StreamReader**, il appartient à l'espace de nommage **System.IO**, qui fournit tous les objets nécessaires pour réaliser tout type d'Entrées/Sorties avec le système. La méthode du **StreamWriter** que nous allons utiliser pour écrire une ligne de données texte est **WriteLine()** (bien sûr);



Pour tout savoir sur **StreamWriter** :

[http://msdn2.microsoft.com/fr-fr/library/3ssew6tk\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/3ssew6tk(VS.80).aspx)



Il va aussi falloir itérer dans la collection de lignes de la table de données. Bon, si l'instruction **foreach** serait plus indiquée dans le cas présent (**foreach** est toujours à préférer à **for** dans une liste ou une collection), nous allons utiliser un **for**, histoire de l'avoir fait au moins une fois !

- Ajoutez à la méthode **SauverFichier** le code d'utilisation du **StreamWriter** pour écrire le fichier;

Code C#

```
private void SauverFichier(string NomDuFichier)
{
    // Utilisation d'un écrivain public !
    using (System.IO.StreamWriter streamWriter
        = new System.IO.StreamWriter(NomDuFichier, false))
    {
    }
}
```



Au fait, elle est où la collection de lignes de la table de données ? ça c'est une bonne question ! Elle est dans la propriété **DataSource** de notre objet de gestion de la source des liaisons **mainBindingSource**, bien sur ! En effet, aussi bien lors de la création d'un nouveau fichier que de l'ouverture d'un fichier existant, nous avons respectivement les lignes :

```
// Charge une table vide dans la source des bindings
mainBindingSource.DataSource = CreerTable();
OU
// Charge le fichier de données dans la source des bindings
mainBindingSource.DataSource =
    OuvrirFichier(fileOpen.FileName);
```

DataSource contient donc un objet de type **DataTable**, qui a une propriété **Rows** pointant sur la collection de lignes de la table de

données.

Au passage, si dans l'une des deux fonctions vous positionnez le curseur sur la propriété **DataSource**, une aide rapide apparaît vous indiquant entre autre le type de la propriété **DataSource** :

Type de DataSource

```
// Charge le fichier de données dans la source des bindings
mainBindingSource.DataSource = OuvrirFichier(nomFichier);
object BindingSource.DataSource
Obtient ou définit la source de données à laquelle le connecteur effectue une liaison. type;
```

DataSource est de type objet, ce qui est normal car cette propriété est destinée à configurer ou obtenir non seulement des tables de données (**DataTable**), mais aussi tout objet pouvant fournir des données (**DataSet**, etc.), donc le Framework ne connaît pas à priori le type à stocker.

- A l'intérieur du bloc **using**, ajoutez à la méthode **SauverFichier** le code pour itérer l'ensemble des lignes de la table de données, en utilisant la boucle **for** ;

Code C#

```
private void SauverFichier(string NomDuFichier)
{
    // Utilisation d'un écrivain public !
    using (System.IO.StreamWriter streamWriter
        = new System.IO.StreamWriter(NomDuFichier, false))
    {
        // Balayage de l'ensemble des lignes de la table
        for (int i = 0;
            i < mainBindingSource.DataSource.Rows.Count;
            i++)
        {
        }
    }
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Vous devez obtenir une erreur (bon, au moins une, quoi !)

Liste d'erreurs					
1 erreur 0 avertissements 0 messages					
	Description	Fichier	Ligne	Colonne	Projet
1	'object' ne contient pas de définition pour 'Rows'	Main.cs	41	58	Coach.Editeur



C'est normal, le compilateur ne peut pas savoir que **DataSource** contient un objet de type **DataTable**. Et comme un objet de base n'a pas de propriété **Rows**, une erreur est générée !



Mais comment indiquer qu'un objet a un type précis ?

C# propose des instructions et des méthodes de conversion (**cast**) d'un type en un autre. Si la conversion est possible, alors la valeur retournée est du type attendu ; Lorsque le compilateur peut automatiquement convertir une valeur dans un autre type, alors on parle de conversion **implicite** ; Si le compilateur nécessite une information pour pouvoir convertir une valeur, alors on parle de conversion **explicite** ;



Comment indiquer une conversion (**cast**) explicite ?

C# propose deux formalismes pour préciser des conversions explicites :

1. La valeur à convertir est précédée du type de conversion entre parenthèse :

```
((DataTable)mainBindingSource.DataSource).Rows
```

Dans le cas où la conversion n'est pas possible, ce formalisme génère une exception à l'exécution ;

2. La valeur à convertir est suivie de l'instruction **as** et du type de conversion :

```
(mainBindingSource.DataSource as DataTable).Rows
```

Dans le cas où la conversion n'est pas possible, ce formalisme génère une valeur nulle (**null**) à l'exécution : c'est pour cette raison que je préfère **as** !

Dans les deux cas, un jeu de parenthèses () isole la valeur à convertir du reste de l'expression.



Pour tout savoir sur la conversion (**cast**) :

[http://msdn2.microsoft.com/fr-fr/library/ms173105\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/ms173105(VS.80).aspx)



Pour tout savoir sur l'instruction **as** :

[http://msdn2.microsoft.com/fr-fr/library/cscsdfbt\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/cscsdfbt(VS.80).aspx)

- Modifiez la méthode **SauverFichier** afin de faire une conversion de **DataSource** en utilisant **as** ;

Code C#

```
private void SauverFichier(string NomDuFichier)
{
    // Utilisation d'un écrivain public !
    using (System.IO.StreamWriter streamWriter
        = new System.IO.StreamWriter(NomDuFichier, false))
    {
        // Balayage de l'ensemble des lignes de la table
        for (int i = 0;
            i < (mainBindingSource.DataSource as DataTable).Rows.Count;
            i++)
        {
        }
    }
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
 - Tapez la touche **F6** pour lancer la compilation du code ; Tout compile !
- Reste maintenant à concaténer la valeur de chacun des champs de chaque ligne de la table de données.



Comment concaténer des chaînes de caractères ?

Pour concaténer les chaînes, le plus simple est d'utiliser la méthode **Concat** de la classe **string**. Cette méthode concatène *n* chaînes de caractères précisées en paramètre.



Pour tout savoir sur l'utilisation des chaînes de caractères :

[http://msdn2.microsoft.com/fr-fr/library/ms228362\(vs.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/ms228362(vs.80).aspx)



Pour tout savoir sur la classe **string** :

[http://msdn2.microsoft.com/fr-fr/library/s1wwdcbf\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/s1wwdcbf(VS.80).aspx)



Comment accéder à la valeur d'un champ d'une ligne de données ?

La propriété **Rows** du type **DataTable** est une collection de lignes de données (**DataRow**). Comme toutes les collections, nous pouvons accéder à une ligne déterminée en donnant son numéro d'ordre entre crochets (en commençant à 0 pour la première). Par exemple, la ligne de code (`mainBindingSource.DataSource as DataTable`).Rows[2] renvoie la troisième ligne du tableau de données.

Pour accéder à la valeur d'un champ, il faut ensuite indiquer, toujours entre crochets, le nom du champ de données voulu. Par exemple, la ligne (`mainBindingSource.DataSource as DataTable`).Rows[2] ["Id"] renvoie la valeur de l'ID de la troisième ligne du tableau de données.

- Fort de tout cela, ajoutez à la méthode **SauverFichier** à ligne de code de concaténation d'une ligne au format texte (l'indentation de départ des lignes a ici été retirée pour plus de lisibilité) ;

Code C#

```
private void SauverFichier(string NomDuFichier)
{
    // Utilisation d'un écrivain public !
    using (System.IO.StreamWriter streamWriter
        = new System.IO.StreamWriter(NomDuFichier, false))
    {
        // Balayage de l'ensemble des lignes de la table
        for (int i = 0;
            i < (mainBindingSource.DataSource as DataTable).Rows.Count;
            i++)
        {
            // Concaténation de la ligne de texte
            streamWriter.WriteLine(
                string.Concat(
                    (mainBindingSource.DataSource as DataTable).Rows[i] ["Id"], ";"
                    , (mainBindingSource.DataSource as DataTable).Rows[i] ["Entreprise"], ";"
                    , (mainBindingSource.DataSource as DataTable).Rows[i] ["Contact"], ";"
                    , (mainBindingSource.DataSource as DataTable).Rows[i] ["Titre"], ";"
                )
            );
        }
    }
}
```

```

, (mainBindingSource.DataSource as DataTable).Rows[i] ["Adresse"], ";"
, (mainBindingSource.DataSource as DataTable).Rows[i] ["Ville"], ";"
, (mainBindingSource.DataSource as DataTable).Rows[i] ["Region"], ";"
, (mainBindingSource.DataSource as DataTable).Rows[i] ["CodePostal"], ";"
, (mainBindingSource.DataSource as DataTable).Rows[i] ["Pays"], ";"
, (mainBindingSource.DataSource as DataTable).Rows[i] ["Telephone"], ";"
, (mainBindingSource.DataSource as DataTable).Rows[i] ["Telecopie"], ";"
, (mainBindingSource.DataSource as DataTable).Rows[i] ["CA"]
  == System.DBNull.Value ?
  "0" :
  (mainBindingSource.DataSource as DataTable).Rows[i] ["CA"].ToString()
));

    }
}
}

```

Si, si, c'est bien une seule ligne de code (compter bien les « ; » et vous verrez, il n'y en a qu'un seul qui appartient à la ligne !). Bon, pour écrire ce genre de code, utilisez l'alignement des lignes et le copier-coller. Vous verrez, c'est très rapide.



Tiens, c'est quoi l'expression ? : ?

L'opérateur conditionnel (?:) retourne l'une de deux valeurs selon la valeur d'une expression booléenne. C'est très utile pour exprimer de manière concise et élégante des calculs qui pourraient nécessiter une construction **if-else** ;



Dans notre cas, si la valeur du **CA** est nulle dans la grille de données (**System.DBNull.Value**), alors la valeur 0 est ajoutée à la chaîne, sinon on écrit le **CA** indiqué ;



Pourquoi utilise-t-on **System.DBNull.Value** ?

System.DBNull.Value est la valeur nulle d'une donnée, indépendamment de son type. En effet, dans notre cas, nous avons un champ de type **int**, qui ne supporte donc pas la valeur **null** ; Pour indiquer qu'aucune valeur de donnée n'a été saisie, la valeur **System.DBNull.Value** est utilisée ;

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout compile normalement encore correctement.



La dernière chose à faire maintenant est d'appeler cette méthode **SauverFichier** à tous les endroits où l'on peut déclencher la sauvegarde d'un fichier, c'est-à-dire lorsque l'utilisateur clique dans l'éditeur du coach sur le menu **Fichier > Enregistrer** ou **Fichier > Enregistrer sous** ;



Pour sauvegarder un fichier sur le disque, nous allons utiliser un objet de type **SaveFileDialog** ; Les principales propriétés et méthodes que nous

allons utiliser sont :

1. **Filter** : Cette propriété sert à filtrer les fichiers en se basant notamment sur l'extension ; nous filtrerons tous les fichiers d'extension ***.coach** ;
 2. **InitialDirectory** : Cette propriété configure le répertoire affiché lors de l'ouverture de la fenêtre ;
 3. **FileName** : Cette propriété obtient ou définit le nom complet du fichier à sauvegarder dans la liste des fichiers proposés ;
 4. **ShowDialog** : Cette méthode affiche la boîte de dialogue, et retourne une valeur de type **DialogResult** indiquant notamment le bouton que l'utilisateur a cliqué pour sortir de la fenêtre ; Nous testerons ici que l'utilisateur a bien cliqué le bouton **OK** avant de lancer la sauvegarde ;
- Dans la fenêtre **Main.cs** en mode **[Design]**, faites un double-clic sur le menu **Fichier > Enregistrer sous** ;
 - Une méthode `enregistrersousToolStripMenuItem_Click` de réponse au menu **Enregistrer sous** a été ajoutée au code du formulaire **Main.cs** ;
 - Dans la méthode `enregistrersousToolStripMenuItem_Click`, ajoutez le code de création et d'affichage de la boîte de dialogue de sauvegarde du fichier ;

Code C#

```
private void enregistrersousToolStripMenuItem_Click(object sender,
EventArgs e)
{
    using (SaveFileDialog fileSave = new SaveFileDialog())
    {
        fileSave.Filter = "Fichiers coach|*.coach";
        fileSave.InitialDirectory = @"c:\\";

        if (fileSave.ShowDialog() == DialogResult.OK)
        {

        }

    }
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ; Vous pouvez d'ailleurs aussi tester le bon fonctionnement de la boîte de dialogue en lançant l'application (ça ne coûte rien et c'est toujours visuel ...) ;
- Dans la méthode `enregistrersousToolStripMenuItem_Click`, ajoutez le code de sauvegarde par appel de la méthode **SauverFichier** ;

Code C#

```
private void enregistrersousToolStripMenuItem_Click(object sender,
EventArgs e)
{
    using (SaveFileDialog fileSave = new SaveFileDialog())
```

```

{
    fileSave.Filter = "Fichiers coach|*.coach";
    fileSave.InitialDirectory = @"c:\";

    if (fileSave.ShowDialog() == DialogResult.OK)
    {
        // Mémorise le nom du fichier
        nomFichier = fileSave.FileName;
        // Sauvegarde les informations
        SauverFichier(nomFichier);
    }
}

```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ;
- Dans la fenêtre **Main.cs** en mode **[Design]**, faites un double-clic sur le menu **Fichier > Enregistrer** ;
- Une méthode `enregistrerToolStripMenuItem_Click` de réponse au menu **Enregistrer sous** a été ajoutée au code du formulaire **Main.cs** ;



Cette méthode de réponse au menu **Enregistrer** va tester si le fichier a déjà été sauvegardé (c'est-à-dire si le nom du fichier n'est pas vide) pour utiliser le bon nom de fichier ; Sinon, elle va router le code sur la méthode correspondant au menu **Enregistrer sous** ;

- Dans la méthode `enregistrerToolStripMenuItem_Click`, ajoutez le code de sauvegarde du fichier, si le nom de fichier existe ;

Code C#

```

private void enregistrerToolStripMenuItem_Click(object sender, EventArgs
e)
{
    // Si le nom du fichier est n'existe pas (i.e. est vide)
    if (nomFichier == string.Empty)
    {
        // route l'appel vers la sauvegarde avec selection du nom
        this.enregistrersousToolStripMenuItem_Click(sender, e);
    }
    else
    {
        // Sauvegarde les informations
        SauverFichier(nomFichier);
    }
}

```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ;

- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ;
- Testez maintenant votre application en créant un nouveau fichier (menu **Nouveau**), ajoutez quelques enregistrement, sauvegardez le tout, ré ouvrez le fichier, etc. C'est magnifique ! C'est presque pro (hum ??).

3.3 Finaliser le comportement de la fenêtre



Pour faire vraiment professionnel, on va ajouter à la fenêtre des comportements bien utiles, comme :

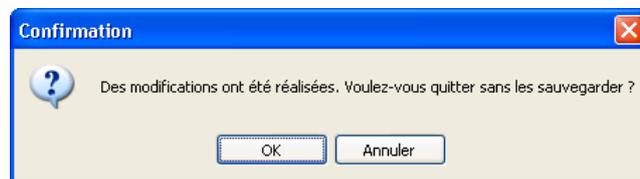
- Protéger la fenêtre contre les fermetures intempestives lorsque des modifications non encore sauvegardées ont été apportées à la liste des données ;
- Indiquer le nom du fichier dans la barre de titre de la fenêtre ;
- Modifier l'icône de la fenêtre ;
- Ou encore, trapper les erreurs qui pourraient survenir sur la grille de données ;

Déroulement de l'exercice :

1. Protégez contre les fermetures intempestives de fenêtre :



Il s'agit ici d'ajouter une boîte de dialogue de confirmation de sortie lorsque l'utilisateur quitte l'application sans sauvegarder les données modifiées, et ce quelque soit le moyen de sortie ;



Pour ce faire, nous allons ajouter au formulaire une fonction de prise en charge d'un événement : **FormClosing** ;



C'est quoi **FormClosing** ?

L'événement **FormClosing** se produit avant la fermeture du formulaire. Lorsqu'un formulaire est fermé, il est supprimé et libère de ce fait toutes les ressources associées au formulaire. Si vous annulez cet événement, le formulaire reste ouvert. Pour annuler la fermeture d'un formulaire, affectez **true** à la propriété **Cancel** du **FormClosingEventArgs** passé au gestionnaire d'événements, en paramètre de la méthode de prise en charge de l'événement.

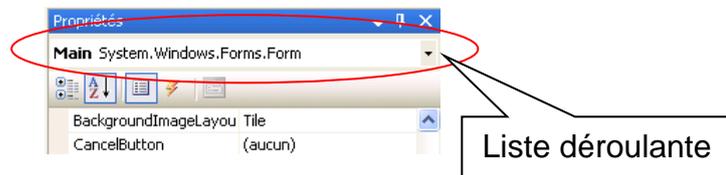


Pour tout savoir sur **FormClosing** :

[http://msdn2.microsoft.com/fr-fr/library/ddy1d5sa\(VS.80\).aspx](http://msdn2.microsoft.com/fr-fr/library/ddy1d5sa(VS.80).aspx)

- Revenez sur le formulaire **Main.cs** en mode **[Design]** en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail ;

- Faites un **clic-droit** à l'intérieur de la fenêtre (en dehors de tout autre contrôle, sur la zone grise de la fenêtre par exemple) et sélectionnez le menu **Propriétés** ;
- Dans la barre d'outils de la fenêtre de propriétés, cliquez le bouton  pour afficher la liste des événements disponibles sur l'objet **Main** ; Pensez à vérifier que vous êtes bien sur le bon objet, dont le nom et le type s'affiche dans la liste déroulante en haut ;



- Dans la liste des événements disponibles, localisez l'événement **FormClosing** ;
- Faites un double-clic sur l'événement **FormClosing** dans la fenêtre de **Propriétés** ; Vous venez d'ajouter la méthode de prise en charge de l'événement **Main_FormClosing** au code de l'application ;



Maintenant, dans le cas où des modifications ont été réalisées, il faut afficher la boîte de dialogue ; Nous allons donc ajouter et tester une variable de type booléen dont la portée sera la totalité de la classe **Main** ; Dès qu'une modification va être apportée à une ligne de données, nous assignerons cette variable à vrai (**true**). Quand l'utilisateur fera une sauvegarde, nous l'assignerons de nouveau à faux (**false**). Et le tour est joué !

- Si ce n'est plus le cas, affichez le code du formulaire **Main.cs**, soit en faisant un clic-droit sur **Main.cs** dans l'**Explorateur de solutions** et en sélectionnant le menu **Afficher le code**, soit en cliquant sur l'onglet **Main.cs** de la zone de travail ;
- Localisez le code où nous avons défini préalablement le membre privé (variable) **nomFichier** ; il est normalement juste avant la définition du constructeur de la classe **Main** ;
- Ajoutez le code suivant pour définir un membre privé (variable) de portée sur la totalité de la classe pour indiquer si des modifications ont été apportées à la liste de données ;

Code C#

```
public partial class Main : Form
{
    // Membres privés
    private string nomFichier = string.Empty;
    private bool desModificationsOntEteRealisees = false;

    public Main()
    {
        ...
    }
}
```

- Revenez maintenant à la méthode **Main_FormClosing** et ajoutez le code suivant pour afficher une boîte de dialogue dans le cas où des modifications ont été apportées à la liste ;

Code C#

```
private void Main_FormClosing(object sender, FormClosingEventArgs e)
{
    // Test si des modification ont été apportées
    if (desModificationsOntEteRealisees)
    {
        if (MessageBox.Show("Des modifications ont été réalisées. Voulez-vous quitter sans les sauvegarder ?", "Confirmation",
            MessageBoxButtons.OKCancel, MessageBoxIcon.Question)
            == DialogResult.Cancel)
            // Annulation de la sortie
            e.Cancel = true;
    }
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ;



Ensuite, il faut déterminer quand une donnée a changé : cette dernière aura changé quand une ligne de la table de données a changé (tiens, on utilise quand – c'est donc un événement ! cf. page 30). Dans la méthode **CreerTable**, nous allons donc connecter une méthode de réponse à l'événement **RowChanged** sur notre objet **result** qui représente la table de données;



Comment connecter en code une méthode de réponse à un événement ? C'est vrai, jusqu'à maintenant nous sommes passés par la fenêtre de propriétés sur un contrôle pour connecter des méthodes de réponses à un événement. Mais il est possible de le faire par le code en utilisant l'opérateur d'assignation pour les événements **+=**, aussi appelé dans la littérature *opérateur de concaténation de délégué* (on verra cela plus en détail dans le prochain atelier).

- Localisez la méthode de création d'une nouvelle table de données **CreerTable** ;
- Positionnez vous juste avant le retour de la valeur et ajoutez la ligne de commentaire ;

Code C#

```
private DataTable CreerTable()
{
    ...
    result.Columns.Add(new DataColumn("CA", typeof(int)));

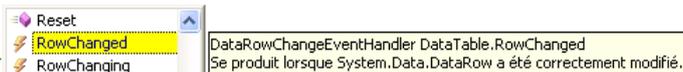
    // Définition de la méthode de réponse en cas de changement

    // retour de la valeur
}
```

```
return tableDeDonnees;
}
```

- Commencez à taper la ligne `result.r` et aidez vous de l'IntelliSense pour localiser l'événement **RowChanged** ;

```
// Définition de la méthode de réponse en cas de changement
result.r
```



- Tapez maintenant l'opérateur `+=` ; L'IntelliSense vous propose d'appuyer sur **TAB** pour insérer la bonne écriture de connexion de l'événement (super, non ?)

```
// Définition de la méthode de réponse en cas de changement
result.RowChanged+=
new DataRowChangeEventHandler(result_RowChanged); (Appuyez sur TABULATION pour insérer)
```

- Appuyez donc une première fois sur la touche **TAB** pour écrire la ligne ; L'IntelliSense vous propose d'appuyer une deuxième fois sur **TAB** pour générer la méthode de prise en charge de l'événement (là, c'est carrément génial !)

```
// Définition de la méthode de réponse en cas de changement
result.RowChanged+=new DataRowChangeEventHandler(result_RowChanged);
Appuyez sur TABULATION pour générer le gestionnaire 'result_RowChanged' dans cette classe
```

- Appuyez donc une deuxième fois sur la touche **TAB** pour générer la méthode de prise en charge de l'événement ; La méthode est automatiquement générée ;

Code C#

```
void result_RowChanged(object sender, DataRowChangeEventArgs e)
{
    throw new Exception("The method or operation is not implemented.");
}
```

- Détruisez la seule ligne de cette méthode (qui indique qu'elle n'a pas encore été implémentée) et ajoutez le code pour indiquer que des modifications ont été apportées à la liste de données ;

Code C#

```
void result_RowChanged(object sender, DataRowChangeEventArgs e)
{
    throw new Exception("The method or operation is not implemented.");
    // Indique que des modifications ont été apportées
    desModificationsOntEteRealisees = true;
}
```



Reste à repasser la variable `desModificationsOntEteRealisees` à **faux (false)** quand la sauvegarde est réalisée, cette dernière opération étant réalisée dans la méthode **SauverFichier** ; Il faudra aussi le faire lors de

l'ouverture d'un fichier CSV (méthode **OuvrirFichier**) car lors de la lecture du fichier CSV, on ajoute les enregistrements et l'événement **RowChanged** est déclenché !

- Localisez la méthode de sauvegarde des données **SauverFichier** ;
- Positionnez en tout début de méthode et ajoutez le code pour assigner **false** à la variable `desModificationsOntEteRealisees` ;

Code C#

```
private void SauverFichier(string NomDuFichier)
{
    // Indique que tout est sauvegardé (s'il n'y a pas un bogue ...)
    desModificationsOntEteRealisees = false;

    // Utilisation d'un écrivain public !
    using (System.IO.StreamWriter streamWriter
...

```

- Localisez la méthode d'ouverture d'un fichier de données **OuvrirFichier** ;
- Positionnez vous en fin de méthode, avant le retour de valeur, et ajoutez le même code que ci-dessus ;

Code C#

```
private DataTable OuvrirFichier(string NomDuFichier)
{
    ...

    // Indique que tout est sauvegardé
    desModificationsOntEteRealisees = false;

    // renvoi de la valeur
    return result;
}

```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ;
- Testez maintenant votre application en modifiant une ou deux lignes et en sortant : tout marche bien !

2. Indiquez l'icône et le nom du fichier :



Bon, là il n'y a pas vraiment de nouveautés, mais cela me chagrinait que l'icône soit faux et que la barre de titre n'indique pas le nom du fichier (un peu perfectionniste, certainement). En challenge, vous pouvez tenter de corriger ces points vous-même en réfléchissant au meilleur moyen d'y arriver 😊.

- Affichez le code du formulaire **Main.cs**, soit en faisant un clic-droit sur **Main.cs** dans l'**Explorateur de solutions** et en sélectionnant le menu **Afficher le code**, soit en cliquant sur l'onglet **Main.cs** de la zone de travail ;
- Localisez la méthode `enregistrersousToolStripMenuItem_Click` ;
- Indiquez que la propriété **Text** du formulaire (**this**) est la concaténation de son titre et du nom du fichier ;

Code C#

```
private void enregistrersousToolStripMenuItem_Click(object sender,
EventArgs e)
{
    using (SaveFileDialog fileSave = new SaveFileDialog())
    {
        ...

        if (fileSave.ShowDialog() == DialogResult.OK)
        {
            ...
            // Sauvegarde les informations
            SauverFichier(nomFichier);
            // Indique le nom du fichier dans le titre
            this.Text
                = string.Concat("Editeur du Coach C#", " - ", nomFichier);
        }
    }
}
```

- Répétez la même opération pour la méthode d'ouverture de fichier `ouvrirToolStripMenuItem_Click` ;

Code C#

```
private void ouvrirToolStripMenuItem_Click(object sender, EventArgs e)
{
    using (OpenFileDialog fileOpen = new OpenFileDialog())
    {
        ...

        if (fileOpen.ShowDialog() == DialogResult.OK)
        {
            ...
            // Mémoire le nom du fichier
            nomFichier = fileOpen.FileName;
            // Indique le nom du fichier dans le titre
            this.Text
                = string.Concat("Editeur du Coach C#", " - ", nomFichier);
        }
    }
}
```

- Localisez le constructeur de la classe **Main** ;
- Juste en dessous de l'appel à **InitializeComponent()**, indiquez que la propriété **Icon** du formulaire (**this**) pointe sur l'icône sauvegardé en ressources ;

Code C#

```
public Main()
{
    ...
    InitializeComponent();

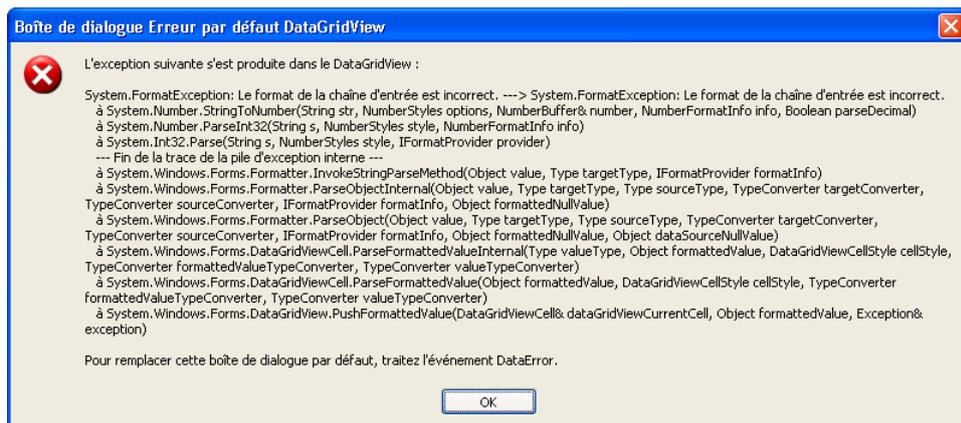
    // Paramétrage de l'icône de la fenêtre
    this.Icon = Properties.Resources.lan_connected;
    ...
}
```

- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ; On obtient bien l'icône et le titre de la fenêtre avec le nom du fichier.

3. Trapez les erreurs de la grille :



Lorsqu'une erreur survient sur la grille, la boîte suivante s'affiche :



Nous allons la remplacer par une boîte plus propre comme celle-ci :



Et aussi (pour le fun) ajouter une entrée dans le journal d'événement de la machine, pour améliorer la maintenabilité de l'application.

- Revenez sur le formulaire **Main.cs** en mode **[Design]** en cliquant sur l'onglet **Main.cs [Design]** de la zone de travail ;
- Faites un **clic-droit** sur la grille de données et sélectionnez le menu **Propriétés** ;
- Dans la barre d'outils de la fenêtre de propriétés, cliquez le bouton  pour afficher la liste des événements disponibles sur l'objet

mainDataFridView ; Pensez à vérifier que vous êtes bien sur le bon objet, dont le nom et le type s'affiche dans la liste déroulante en haut ;

- Dans la liste des événements disponibles, localisez l'événement **DataError** ;
- Faites un **double-clic** sur l'événement **DataError** dans la fenêtre de **Propriétés** ; Vous venez d'ajouter la méthode de prise en charge de l'événement **mainDataGridView_DataError** au code de l'application ;
- Ajoutez à cette méthode le code pour afficher le message d'erreur ;

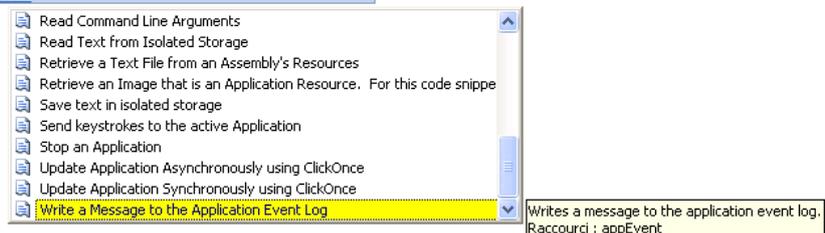
Code C#

```
private void mainDataGridView_DataError(object sender,
DataGridViewDataErrorEventArgs e)
{
    // Affiche le message d'erreur
    MessageBox.Show(e.Exception.Message);
}
```

- Sous la dernière ligne insérée, faites un **clic-droit** et sélectionnez le menu **Insérer un extrait...** ;
- Sélectionnez l'extrait de code **application > Write a message to the Application Event Log** ;

```
// Affiche le message d'erreur
MessageBox.Show(e.Exception.Message);
```

Insérer un extrait: application >



- Remplacez "ApplicationName" par "Coach C#" ;
- Remplacez "Action complete." par e.Exception.Message;

Code C#

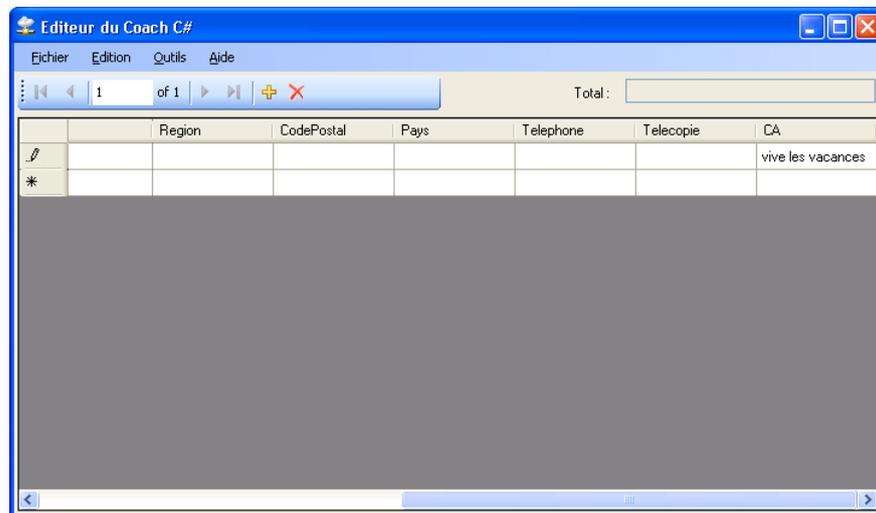
```
private void mainDataGridView_DataError(object sender,
DataGridViewDataErrorEventArgs e)
{
    // Affiche le message d'erreur
    MessageBox.Show(e.Exception.Message);

    // Create the source, if it does not already exist.
    if (!System.Diagnostics.EventLog.SourceExists("Coach C#"))
    {
        System.Diagnostics.EventLog.CreateEventSource("Coach C#",
"Application");
    }

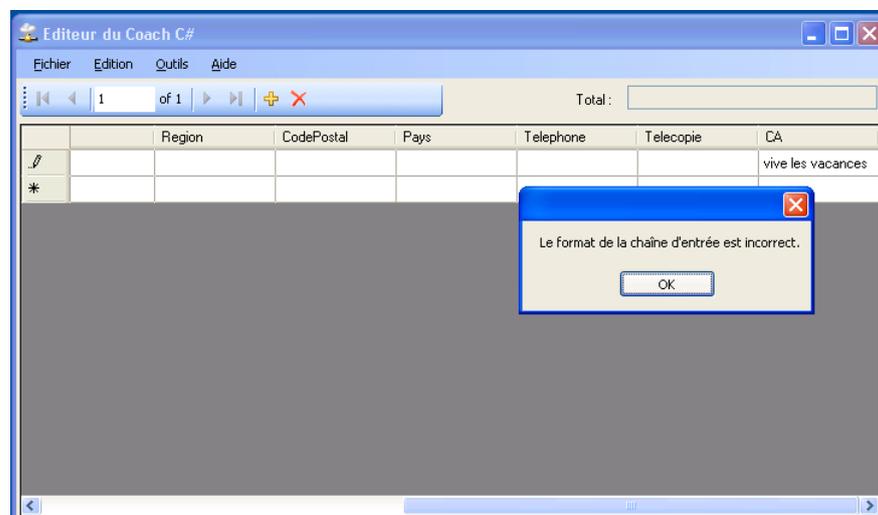
    // Create an EventLog instance and assign its source.
    System.Diagnostics.EventLog myLog =
        new System.Diagnostics.EventLog();
    myLog.Source = "Coach C#";
}
```

```
// Write an informational entry to the event log.  
myLog.WriteEntry(e.Exception.Message);  
}
```

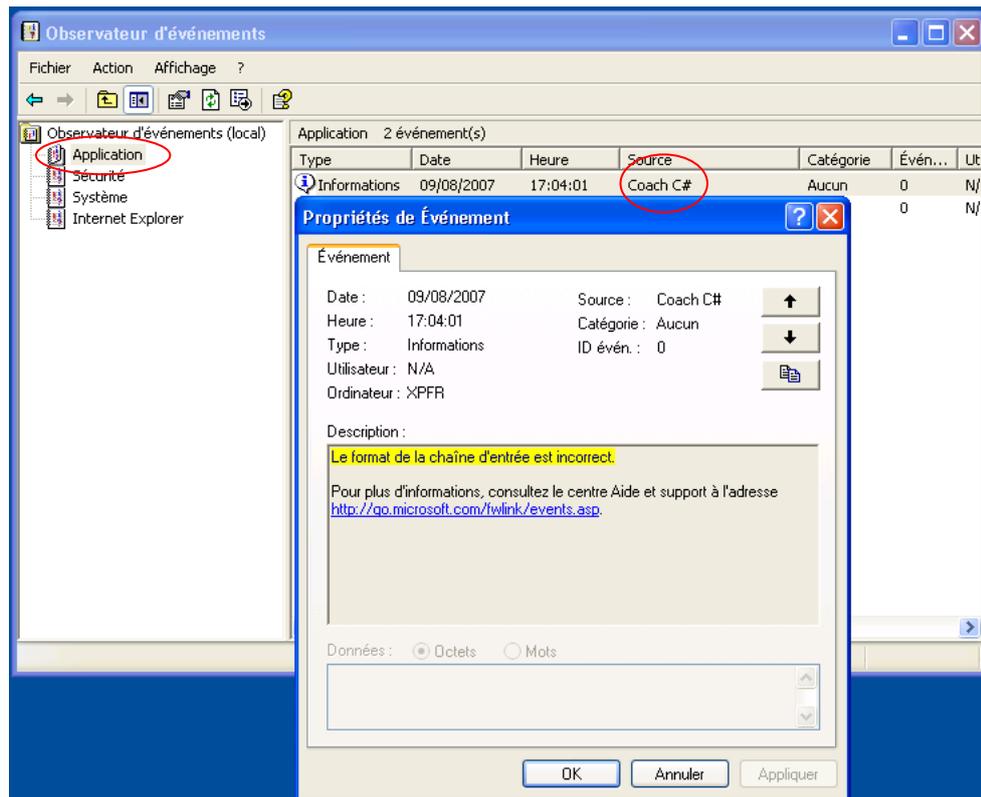
- Dans la barre d'outils de Visual C# Express, cliquez le bouton  pour sauvegarder les modifications réalisées ;
- Tapez la touche **F6** pour lancer la compilation du code ; Tout doit se compiler correctement ;
- Dans la barre d'outil de Visual C# Express, cliquez le bouton  pour démarrer votre application en mode de débogage ;
- Dans l'**Editeur du Coach C#**, sélectionnez le menu **Fichier > Nouveau** ;
- Déplacez le curseur sur le champ **CA** de la première ligne, et saisissez une chaîne de caractères quelconque, sans signification dans notre contexte ;



- Cliquez sur la deuxième ligne de la grille pour valider la saisie ; Le message d'erreur apparaît ;



- Cliquez le bouton « **Ok** », puis la touche **Echappement (ESC)** pour sortir de la saisie ; Fermez la fenêtre de l'éditeur ;
- Dans Windows XP, sélectionnez le menu **Démarrer > Panneau de configuration > Outils d'administration > Observateur d'événements** ;
- Sélectionnez les événements de type **Application** et **double-cliquez** sur celui dont la source est **Coach C#** ; Votre événement apparaît ;



- Fermez maintenant l'observateur d'événement ; Et voilà, vous avez ce que l'on peut appeler une première application en C#.



Plutôt que d'écrire le message d'erreur uniquement, vous pouvez en profiter pour concaténer différentes informations comme la méthode qui plante, le nom de l'utilisateur, etc. enfin tout ce qui sera utile pour vous aider dans votre travail de maintenance.



Bon, et bien voilà, nous sommes arrivés à la fin de ce deuxième atelier. Vous avez été formidable (si, si, vous avez tenu jusqu'à cette ligne !).

Et l'air de rien, nous avons vu les types et les structures de base du C#, tout en développant notre interface utilisateur. Mais ne vous arrêtez pas là, continuez le coach, car maintenant nous allons voir comment modéliser les objets. En gros, nous allons maintenant changer le mode de fonctionnement interne de l'application, sans impact sur l'interface

utilisateur, et vous allez comprendre la puissance du C# !

4 Pour aller plus loin...

4.1 Les liens utiles

Pour avoir plus d'information concernant les bonnes règles à suivre pour le développement, voici un lien intéressant :

- <http://msdn2.microsoft.com/fr-fr/library/bb278146.aspx> , qui explique comment réviser un code ;

4.2 Idée/ressource

Si vous souhaitez apprendre la programmation d'application Web avec C#, il existe le Coach ASP.Net :

<http://www.microsoft.com/france/msdn/aspnet/coach/coach.aspx>

Et pour bien gérer vos projets de développement, suivez le coach Visual Studio Team System :

<http://www.microsoft.com/france/msdn/vstudio/teamsystem/evaluez/CoachVSTS.aspx>