PROCEDURE D'INSTALLATION D'UN ENVIRONNEMENT POUR DEVELOPPER AVEC VISUAL BASIC

SOMMAIRE

SE PREPARER A L'INSTALLATION DE L'ENVIRONNEMENT 3

RÉCAPITULER LES COMPOSANTS À INSTALLER 3

RASSEMBLER TOUS LES SUPPORTS D'INSTALLATION 6

Précautions avant le téléchargement 7

Télécharger les produits 8

INSTALLER L'ENVIRONNEMENT 9

INSTALLER MICROSOFT VISUAL BASIC EXPRESS EDITION (AVEC SQL EXPRESS) 9

INSCRIRE MICROSOFT VISUAL BASIC EXPRESS EDITION 15

CONFIGURER LES SERVICES DE SQL SERVER EXPRESS EDITION 19

INSTALLER SQL SERVER MANAGEMENT STUDIO EXPRESS 22

SE PREPARER A L'INSTALLATION DE L'ENVIRONNEMENT

Dans cet exercice, vous allez vous préparer à dérouler le processus d'installation d'un environnement complet pour développer avec Visual Basic.NET. Il vous faut :

- Récapituler ce qu'il faut installer
- Récupérer tous les supports d'installation
- Valider les pré-requis

Objectif

A la fin de ce premier module, vous serez prêt d'un point de vue matériel et ressources utiles à vous lancer dans le processus d'installation des logiciels nécessaires.

RÉCAPITULER LES COMPOSANTS À INSTALLER

De quoi avez-vous besoin pour développer une application avec VB.NET ?

Avant de vous lancez à développer, il faut faire le point sur ce dont vous avez besoin ! Visual Basic.NET est un ***langage de développement*** et non un produit, même si nous allons le voir, Microsoft a nommé l'un de ses produits de la gamme Express Edition Microsoft Visual Basic !

Qu'est-ce que cela veut dire ?

Cela veut dire qu'avec le langage Visual Basic.NET, vous pouvez donc construire tout type d'applications : Windows, Web, des services, etc.

Dans le cadre des exercices du coach VB.NET, nous allons créer une application Windows, mais uniquement en support d'illustration des caractéristiques du langage.

Récapitulons ensemble ce dont vous avez besoin pour développer avec VB.NET :

1. un système d'exploitation :

Microsoft Windows XP avec le Service Pack 2 ou **Microsoft Windows Vista avec le service Pack 1** sont les systèmes d'exploitation incontournables pour installer un poste de développement. L'environnement configuré pour le coach utilise Microsoft Vista.

Notez que vous pouvez conserver votre poste de travail intact et tout simplement créer une machine virtuelle en utilisant **Microsoft Virtual PC 2007** totalement gratuit. C'est d'ailleurs la configuration utilisée par le coach VB.NET !

Pour tout savoir au sujet de Virtual PC :

http://msdn2.microsoft.com/fr-fr/library/wxh6fsc7(VS.80).aspx

2. un environnement de développement :

Microsoft Visual Studio est la plate-forme de développement idéale pour développer des applications codées en Visual Basic.NET. Il est disponible en plusieurs éditions dont **Microsoft Visual Basic 2008 Express Edition** qui ne nécessite aucune licence et que nous vous proposons d'utiliser dans le cadre de ces ateliers.

Mais qu'est-ce qu'on entend par ***environnement de développement*** ?

En réalité, rien ne vous empêche de développer votre application Visual Basic .NET dans le Bloc-notes de Windows ! Mais s'il est possible de développer en utilisant un simple éditeur de texte, c'est incomparable avec un environnement de développement tel que Visual Studio qui propose tout un ensemble d'outils d'aide au développement. C'est tout votre travail de codage qui s'en trouve simplifié !

Par contre il est important de comprendre que l'environnement de développement n'enrichit ni le langage, ni ce que peut faire la plate-forme .NET. Ce qui caractérise les différentes versions de Visual Studio réside uniquement dans les outils, modèles et options disponibles en fonction des versions. Par exemple, selon vous, est-ce qu'on peut développer en Visual Basic.NET avec l'édition Microsoft Visual Web Developer Express Edition ? Oui, bien sûr ! VWD est simplement une édition gratuite de Visual Studio dédié au développement web, mais vous être libre de développer dans le langage .NET de votre choix !!

Pour voir quel type d'environnement propose Visual Basic Express :

http://msdn2.microsoft.com/fr-fr/express/aa718406.aspx

Pour voir quel type d'environnement propose Visual Web Developer Express :

http://msdn2.microsoft.com/fr-fr/express/aa700797.aspx

3. un gestionnaire de base de données :

Ce n'est évidemment pas une nécessité mais si vous prévoyez de gérer des données dans votre application, c'est tout simplement indispensable ! **Microsoft SQL Server 2008** est le gestionnaire de base de données de Microsoft. Tout comme Visual Studio, ce produit existe dans une édition totalement gratuite appelée **Microsoft SQL Server 2008 Express Edition**. En revanche dans cette édition, il n'y a pas d'outil de gestion par défaut (c'est un peu comme si on vous donnait le moteur d'une voiture mais sans la carrosserie ☺). Il faut donc prévoir d'installer en plus **Microsoft SQL Server Management Studio Express**.

Si vous craignez d'installer SQL Server, inutile de vous inquiéter, ça se fait tout seul ! Toutes les éditions de Visual Studio 2008 intègre en standard l'édition de SQL Server 2008 correspondante. Dans ce tutorial, avec l'édition Express des outils, vous verrez qu'il n'y a pas plus simple pour découvrir en douceur comment manipuler des données dans une application, tout en utilisant la puissance d'un vrai gestionnaire de bases de données.

Mais au fait, c'est quoi ces éditions Express ?

Les éditions Express proposent des outils plus légers que les éditions complètes, très faciles à utiliser pour découvrir les technologies de développement Microsoft et surtout totalement gratuits ! En d'autres termes, pour démarrer c'est l'idéal. Mais préférez les éditions complètes des outils pour un développement professionnel de solutions d'entreprise.

Pour comparer les différentes éditions de Visual Studio :

http://msdn2.microsoft.com/fr-fr/vstudio/aa700921.aspx

RASSEMBLER TOUS LES SUPPORTS D'INSTALLATION

Dans cette procédure, nous vous proposons de travailler sur la base des éditions Express des produits en version française. La procédure a été testée sur une machine virtuelle VPC 2007 installée avec Windows Vista Professionnel en français.

Voici un récapitulatif des supports d'installation dont vous avez besoin, hors système d'exploitation, sachant que le second inclut le premier (et l'installe donc automatiquement).

- ☐ **Microsoft SQL Server 2008 Express Edition**
- ☐ **Microsoft Visual Basic 2008 Express Edition**
- ☐ **Microsoft SQL Server Management Studio Express**

Ces produits en édition Express sont téléchargeables directement depuis le site web Microsoft.

Même si cette procédure s'appuie sur les éditions Express des produits, qui sont faciles d'accès, sachez que si vous disposez d'une licence pour les autres éditions, vous pouvez bien évidemment vous configurer un environnement basé sur celles-ci, ou utiliser un environnement existant.

Pour récupérer des versions d'évaluation des produits complets :

- Pour Visual Studio 2008 :

http://msdn2.microsoft.com/fr-fr/vstudio/cc138245.aspx

- Pour SQL Server 2008 :

http://www.microsoft.com/sqlserver/2008/en/us/trial-software.aspx

- pour SQL Server Management Studio Express :

http://msdn2.microsoft.com/fr-fr/express/bb410791.aspx

PRECAUTIONS AVANT LE TELECHARGEMENT

Quelques petites remarques avant de procéder au téléchargement :

Pour pouvoir lancer le téléchargement des éditions Express des produits, il n'est plus nécessaire de s'inscrire auprès de Microsoft avec un compte Windows Live ID (anciennement Passeport). En revanche, cet enregistrement est nécessaire pour utiliser le produit une fois installé. Vous verrez que cela ne prend que quelques minutes, même si vous n'avez pas encore de Windows Live ID.

Comme beaucoup de produits téléchargeables sur Internet, le premier fichier téléchargé ne contient pas l'ensemble du produit à installer. Il faut prévoir une seconde phase de téléchargement ***pendant*** le processus d'installation. Donc prévoyez d'avoir une connexion Internet pendant toute la première phase d'installation.

Enfin, pensez à protéger votre système Windows en installant les dernières mises-à-jour de Windows Update :

http://www.update.microsoft.com/microsoftupdate

TELECHARGER LES PRODUITS

Puisque Visual Basic 2008 Express Edition installe également SQL Server 2008 Express Edition, vous n'avez donc qu'à procéder au téléchargement de l'édition Express de Visual Studio pour Visual Basic :

1. Télécharger Microsoft Visual Basic 2008 Express Edition en français :
   - Téléchargez le fichier **vbsetup.exe** à partir de l'adresse suivante :

     http://msdn2.microsoft.com/fr-fr/express/aa975050.aspx

2. Vous pouvez également télécharger dès maintenant Microsoft SQL Management Studio Express :
   - Le téléchargement est disponible à l'adresse suivante :

     http://msdn2.microsoft.com/fr-fr/express/bb410791.aspx

   - Enregistrez-vous sur le site avec un votre compte Windows Live ID si vous le souhaitez.

- Localisez sur la page la rubrique **SQL Server Management Studio Express** puis cliquez le bouton **Téléchargement ** (43,1 Mo)** pour récupérer le fichier **SQLServer2005_SSMSEE.msi**.

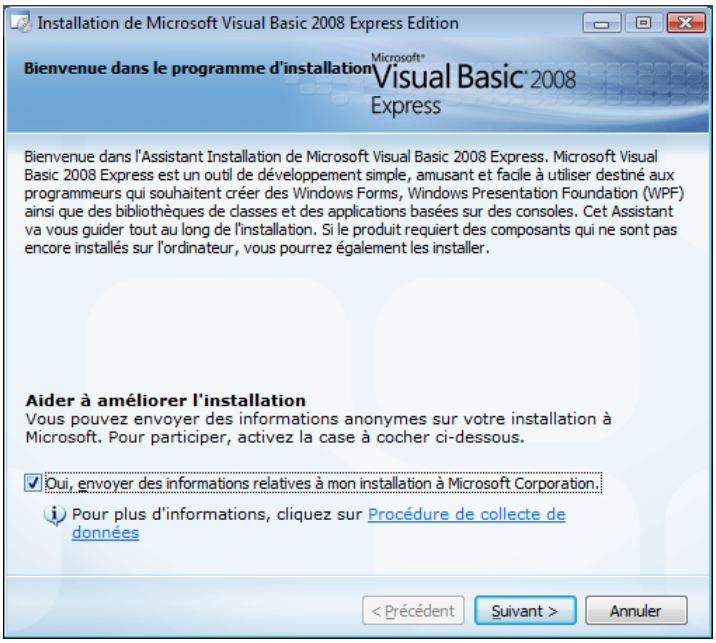
INSTALLER L'ENVIRONNEMENT

INSTALLER MICROSOFT VISUAL BASIC EXPRESS EDITION (AVEC SQL EXPRESS)

L'objectif de cette étape est d'installer Visual Basic 2008 Express Edition.

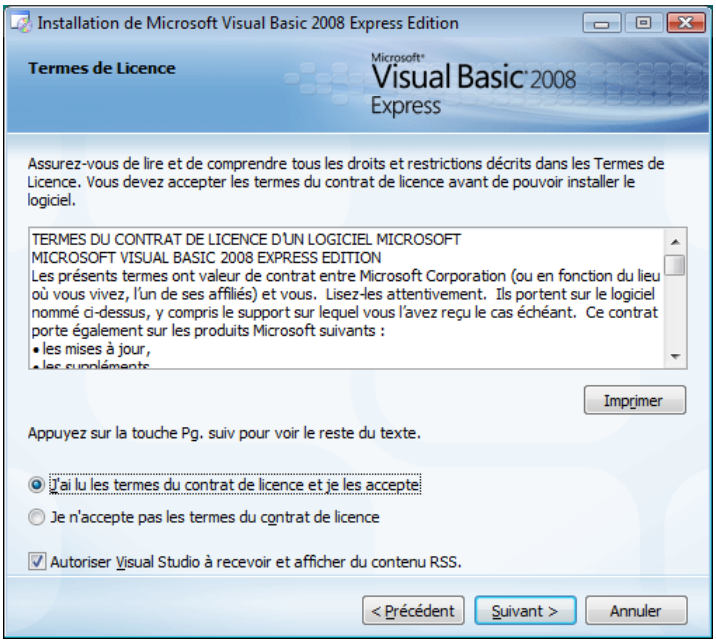
Déroulement de l'étape :

1. Lancez le programme d'installation :
   - Double cliquez sur **vbsetup.exe**.
2. Dans l'écran de bienvenue :
   - Cliquez sur la case à cocher **Oui, envoyer des informations relatives à mon installation à Microsoft Corporation** *si vous souhaitez envoyer un rapport d'installation à Microsoft.*



- Cliquez sur **Suivant**.

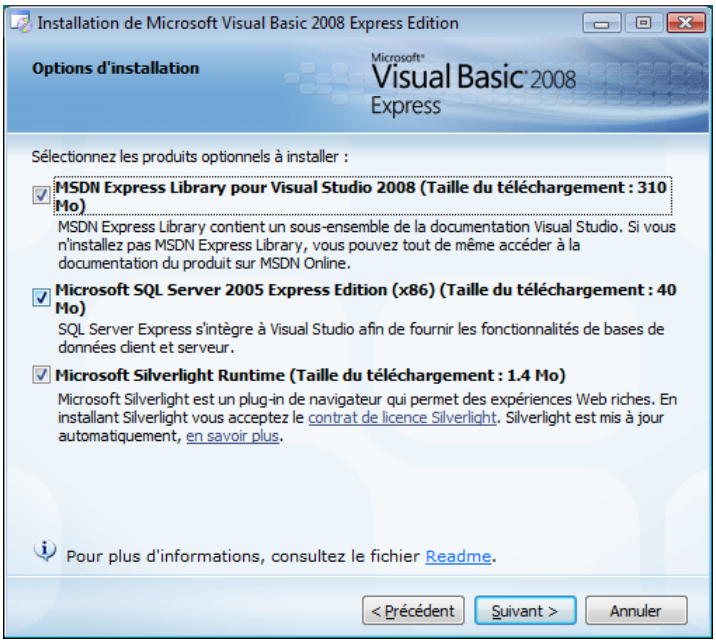
3. Dans l'écran **Termes de Licence** :
    - Cliquez sur la case à cocher **J'ai lu les termes du contrat et je les accepte**.



- Cliquez sur **Suivant**.

4. Dans l'écran **Options d'installation** :

   - Cliquez sur les trois cases à cocher pour installer la librairie MSDN contenant toute la documentation du produit, SQL Server Express Edition et le runtime de Microsoft Silverlight.



A l'heure où nous éditons cette procédure, SQL Server 2008 n'étant pas encore disponible (bien que sur le point de l'être), les éditions Express de Visual Studio proposent encore la version 2005 de SQL Server.

Notez que parmi les options d'installation vous est proposé également le plug-in Silverlight. Même s'il ne vous servira pas pour le développement de l'application Windows que nous vous proposons dans le coach VB.NET, installez-le ! Il s'agit d'un simple plug-in à votre navigateur internet qui vous permettra d'accéder à une nouvelle génération d'applications web riches et interactives (type RIA).

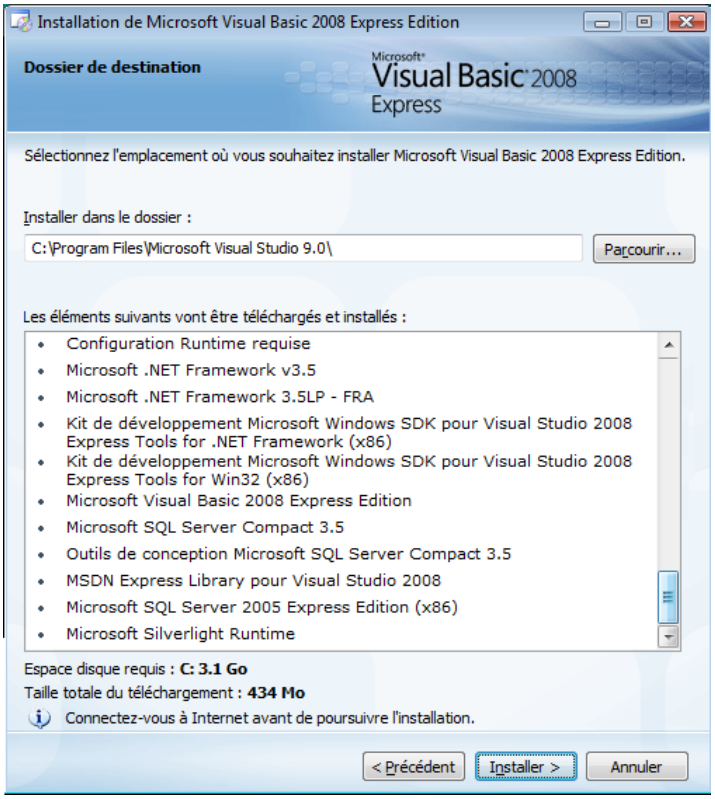
Pour en savoir plus sur Silverlight, sachez qu'il existe d'ors et déjà un centre de développement MSDN dédié au produit :

http://msdn2.microsoft.com/fr-fr/asp.net/bb187358.aspx

- Cliquez sur **Suivant**.

5. Dans l'écran **Dossier de destination** :

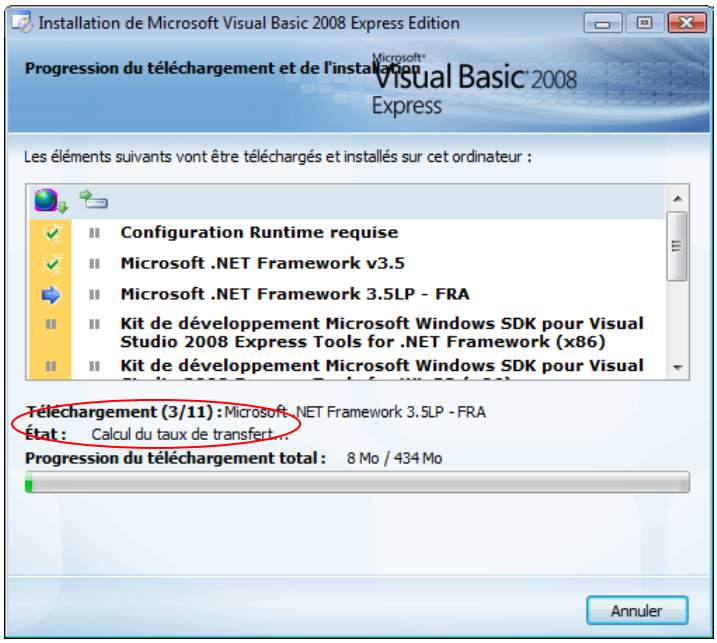
   - Changer le chemin vers le dossier d'installation par défaut *si vous le souhaitez* en cliquant sur **Parcourir…**



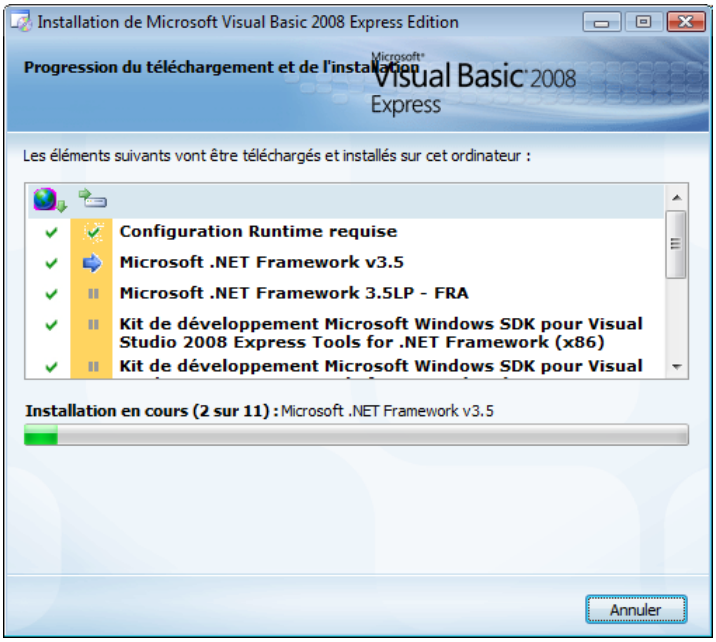
- Cliquez sur **Installer**.

6. Dans l'écran **Progression du téléchargement et de l'installation** :

   - Patientez.

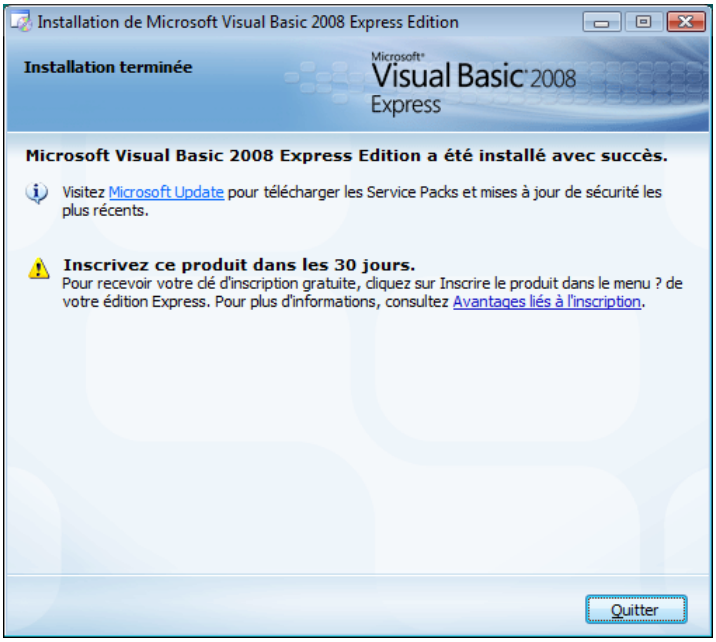


C'est dans cette étape que se produit la suite et fin du téléchargement des produits. Au sortir de cette étape, vous pourrez vous déconnecter d'Internet.

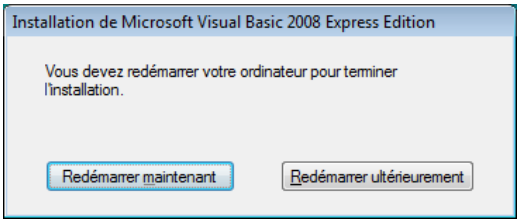


7. Dans l'écran **Installation terminée** :

   - Contrôlez que l'installation s'est terminée avec succès.



   - Cliquez sur **Quitter**.
   - Redémarrer Windows si cela vous est demandé :



INSCRIRE MICROSOFT VISUAL BASIC EXPRESS EDITION

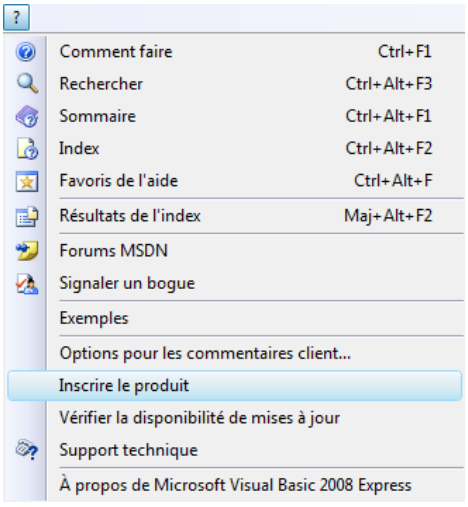
L'objectif de cette étape est d'enregistrer et d'activer Visual Basic Express Edition auprès de Microsoft.

Il faut savoir que chaque installation d'une version de Visual Studio Express Edition requiert une inscription qui conditionne la réception d'une clé d'activation unique pour déverrouiller le produit Express que vous avez installé de façon à l'utiliser sans limite. D'autre part, l'inscription du produit peut vous faire bénéficier de nombreux avantages gratuits.

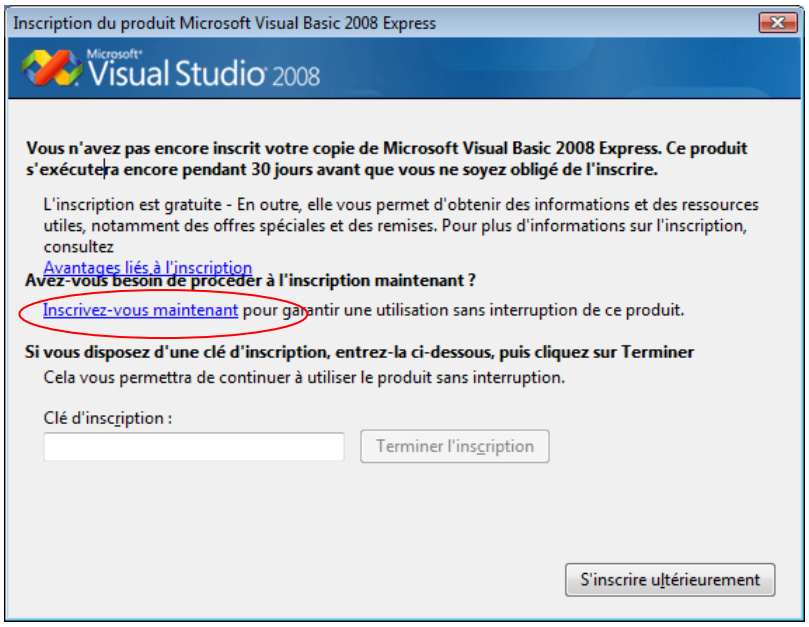
Pour en savoir plus :
http://www.microsoft.com/france/msdn/vstudio/express/register.mspx

Déroulement de l'étape :

1. Enregistrez et activez Visual Basic Express Edition :
    - Cliquez sur **Démarrer** > **Tous les programmes** > **Microsoft Visual Basic 2008 Express Edition**.
    - Dans le menu **Aide** de Visual Basic, cliquez sur **Inscrire le produit…**



- Dans la fenêtre **Inscription du produit…** cliquez sur le lien **Inscrivez-vous maintenant**.

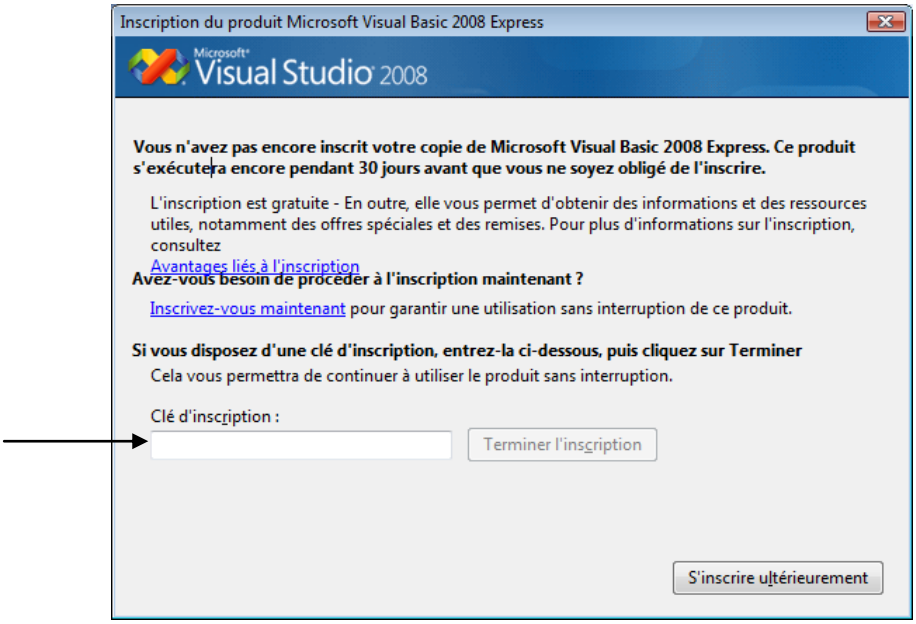


- Connectez-vous avec votre compte Windows Live ID. Si vous n'en possédez pas encore un, créez un compte en cliquant sur **Inscription** dans la rubrique **Vous ne possédez pas de compte Windows Live ID**.

- Si vous avez un compte Passeport, l'enregistrement vous demande également de répondre à une page d'information sur votre compte et de vérifier votre adresse email. C'est aussi l'occasion de vous abonner aux mises à jour sur le produit. Cliquez sur **Continuer**.

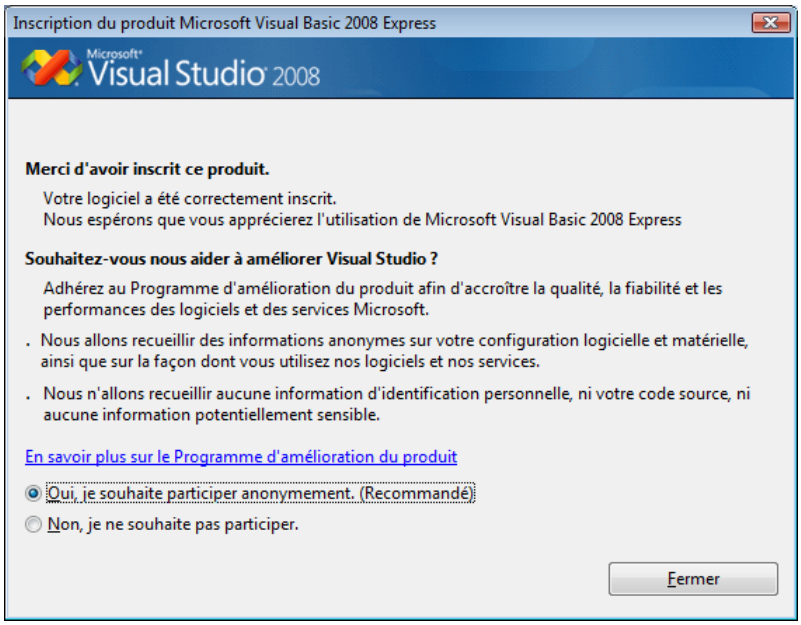
- Une fois que vous êtes connecté, notez **la clé d'enregistrement** à 14 caractères :

- Revenez sur la fenêtre **Inscription du produit…** précédente et saisissez la clé d'inscription notée précédemment :



- Le bouton **Terminer l'inscription** devient actif. Cliquez-le pour terminer l'inscription.

- Dans l'écran suivant, cochez **Oui** si vous voulez contribuer à l'amélioration du produit.



- Cliquez sur **Fermer**.

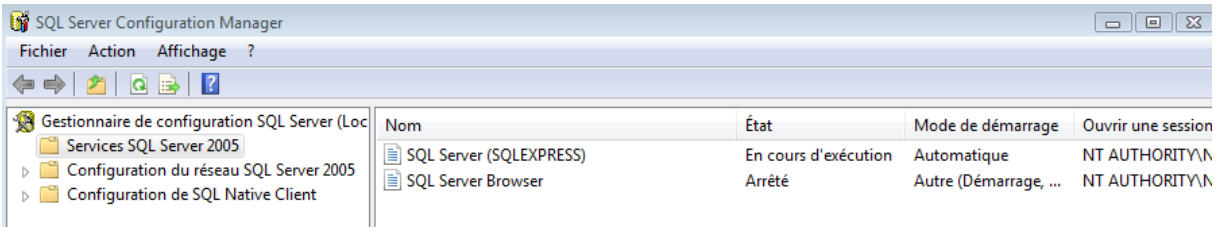
CONFIGURER LES SERVICES DE SQL SERVER EXPRESS EDITION

L'objectif de cette étape est d'apprendre à configurer les services de Microsoft SQL Server Express Edition.

Déroulement de l'étape :

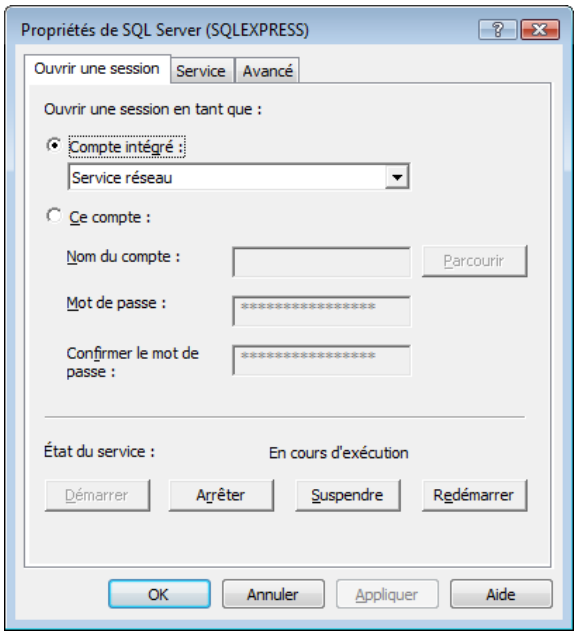
1. Lancez l'outil de configuration de SQL Server Express :
   - **Démarrer** > **Tous les programmes** > **Microsoft SQL Server 2005** > **Outils de configuration** > **Gestionnaire de configuration SQL Server**.
2. Configurez les services SQL Server :

- Cliquez **Gestionnaire de configuration SQL Server (Local)** > **Services SQL Server 2005**.

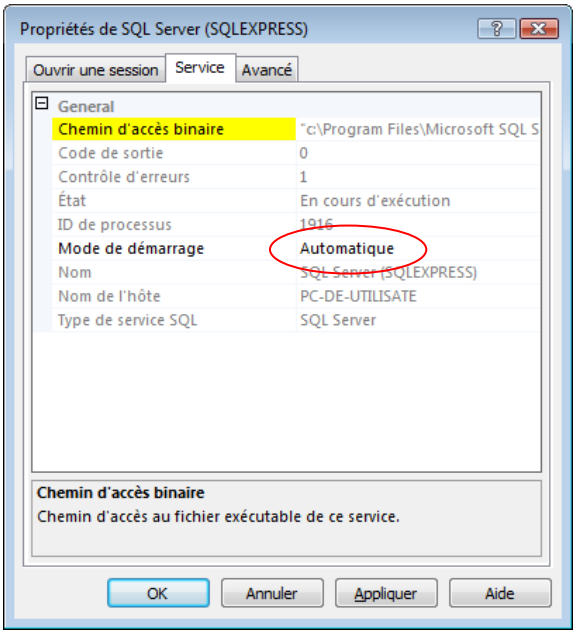


Pour des raisons de sécurité, le service **SQL Server Browser** n'est pas démarré par défaut. Il s'agit d'un nouveau service dans SQL Server 2005 qui est utilisé pour identifier les ports sur lesquels écoutent les instances nommées. Ce qui signifie que ce service doit être démarré dans le cas où vous souhaiteriez utiliser votre serveur SQL à distance.

- Cliquez avec le bouton droit sur **SQL Server (SQLEXPRESS)** > **Propriétés** pour faire apparaître les propriétés du service.
- Dans l'onglet **Ouvrir une session**, observez que le compte d'exécution du service est **Service Réseau**.
- Si le service n'est pas démarré, cliquez sur **Démarrer**.



- Dans l'onglet **Service**, validez que le mode de démarrage est en automatique pour éviter d'avoir à redémarrer manuellement le service à chaque redémarrage de votre système.



- Fermez la fenêtre en cliquant sur **OK**.

INSTALLER SQL SERVER MANAGEMENT STUDIO EXPRESS

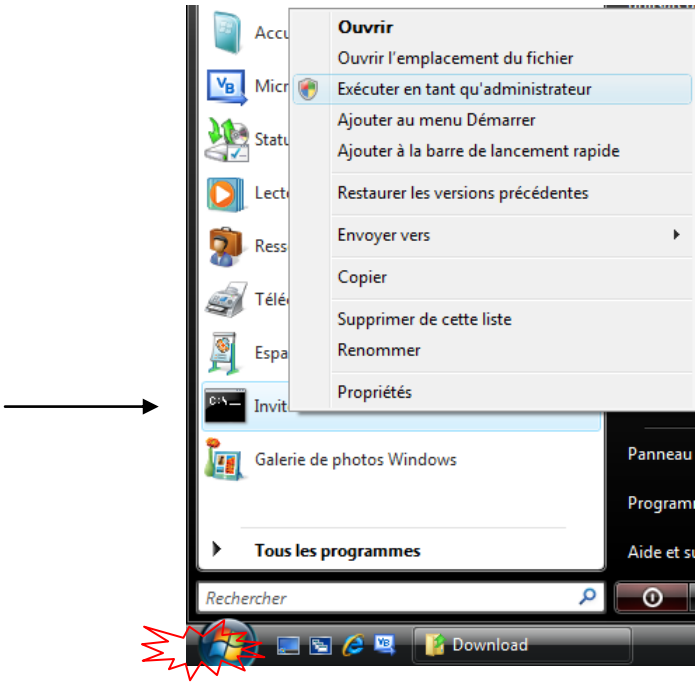
L'objectif de cette étape est d'installer l'outil d'administration de SQL Server 2005.

Déroulement de l'étape :

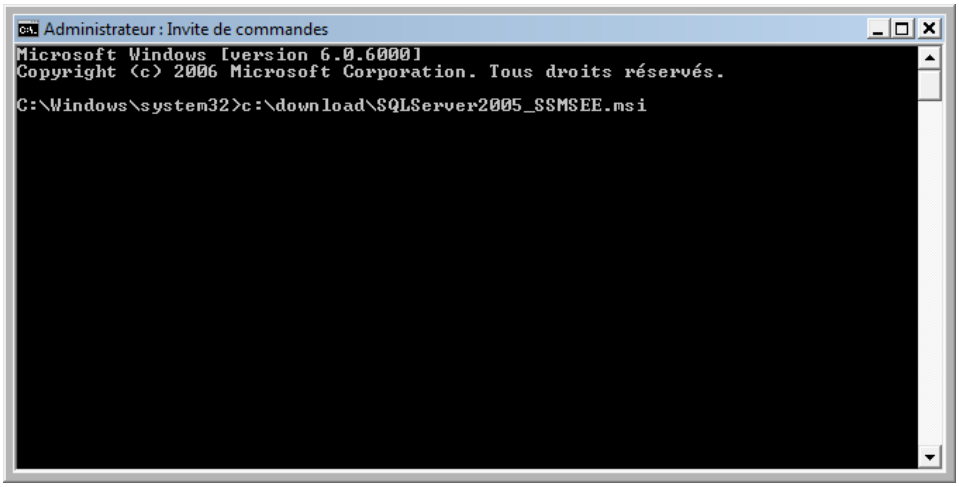
1. Lancez l'installation :

Attention, sous Windows Vista, il faut absolument lancer le fichier d'installation (.msi) en tant qu'administrateur pour avoir les privilèges adéquats sinon vous récupérez une erreur à l'installation.

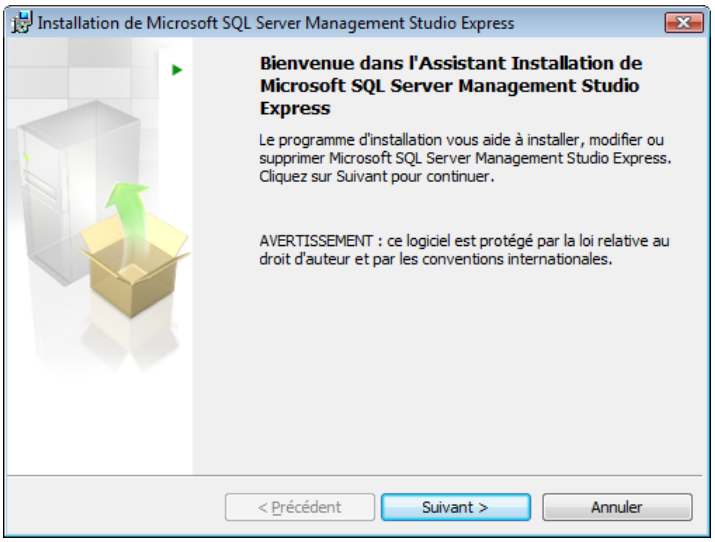
- Ouvrez l'Invite de commandes en tant qu'administrateur en faisant un clic droit sur le menu **Invite de commandes** de Windows Vista > **Exécuter en tant qu'administrateur**.



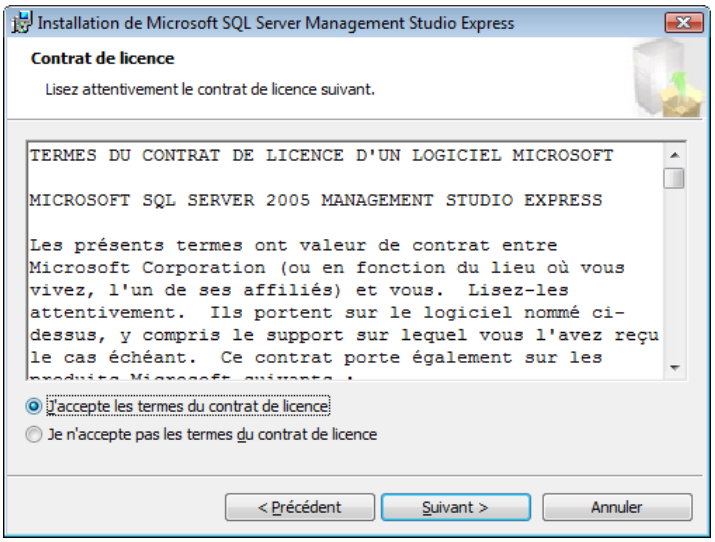
- Validez la demande d'autorisation de Windows Vista.
- Exécutez le fichier **SQLServer2005_SSMSEE.msi** à partir de l'invite de commandes.



- Dans l'écran de bienvenue, cliquez sur **Suivant** :

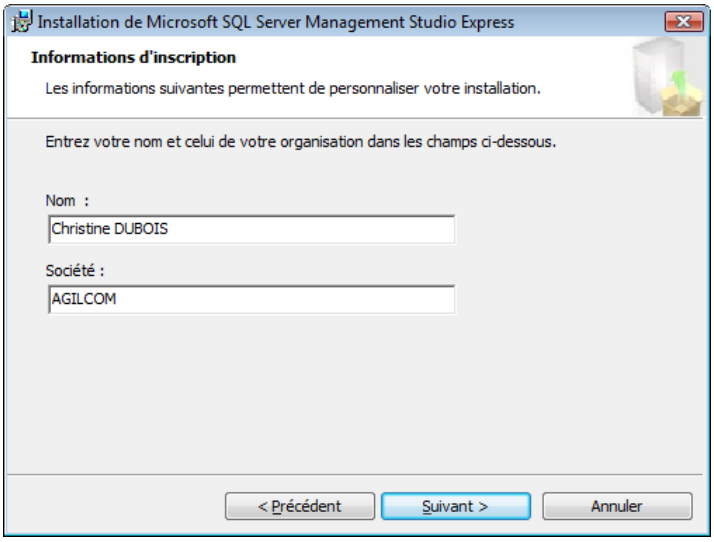


- Dans l'écran **Contrat de licence,** cochez **J'accepte les termes du contrat de licence.**



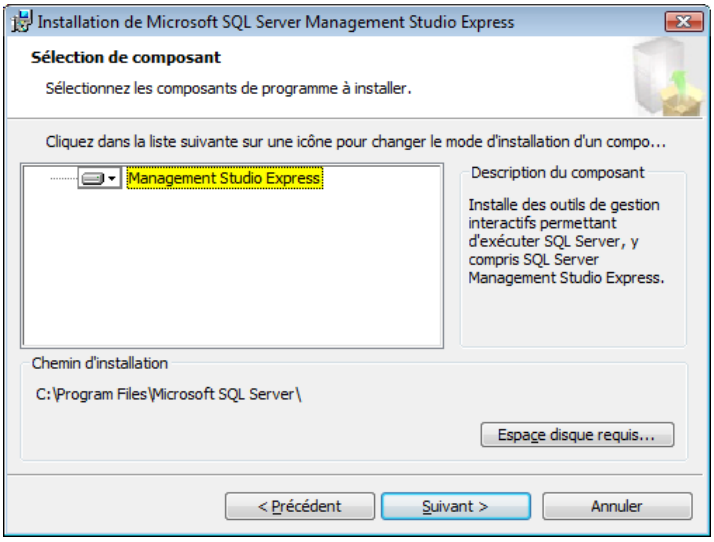
- Cliquez **Suivant**.

- Dans l'écran **Informations d'inscription**, entrez votre nom et le nom de votre entreprise :



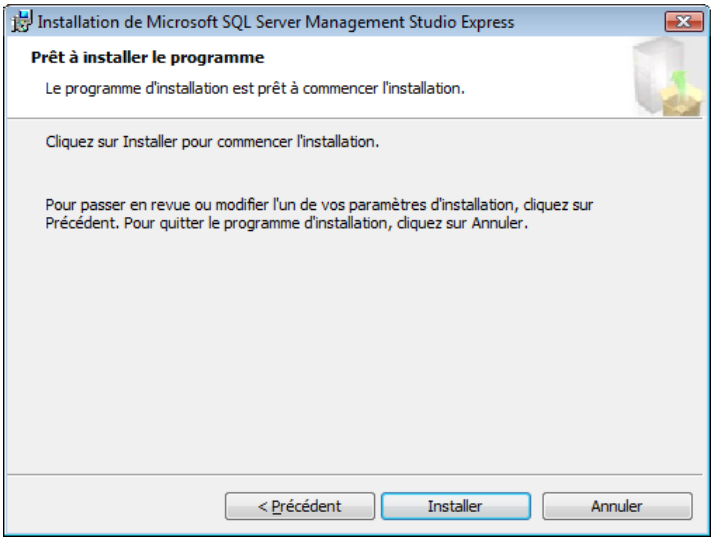
- Cliquez **Suivant**.

- Dans l'écran **Sélection de composant**, modifiez si besoin le chemin d'installation proposé par défaut.



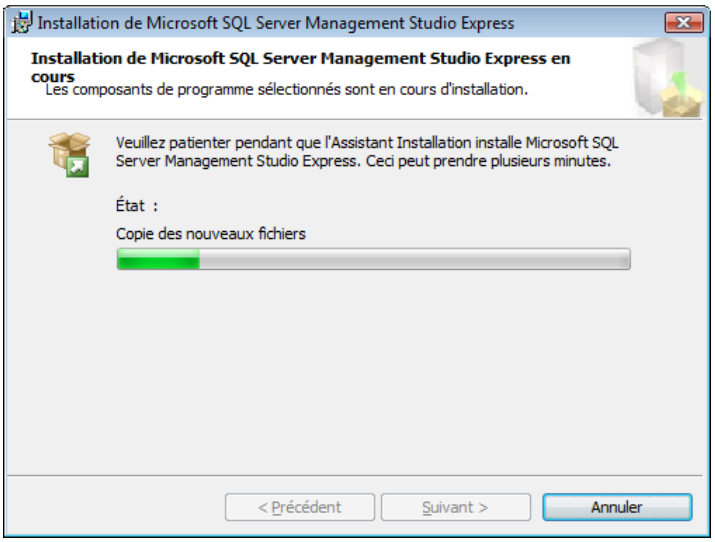
- Cliquez **Suivant**.

- Dans l'écran **Prêt à installer le programme**, cliquez **Installer** :

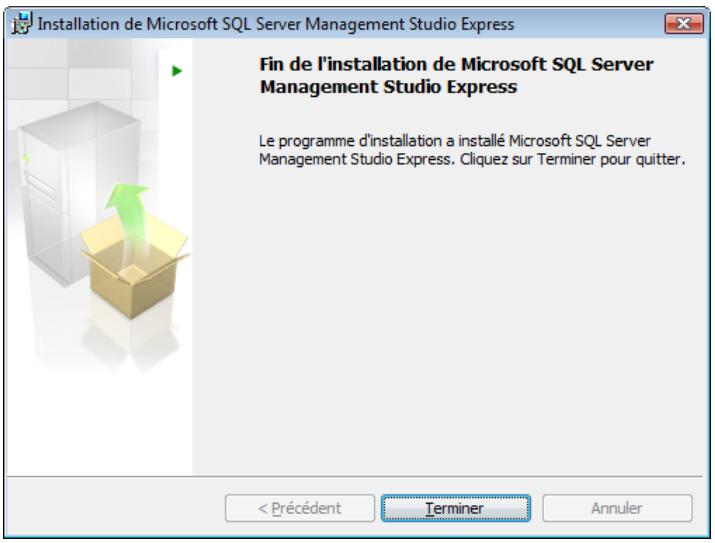


- Validez la demande d'autorisation de Windows Vista pour lancer l'installation.

L'installation ne prend que quelques minutes.

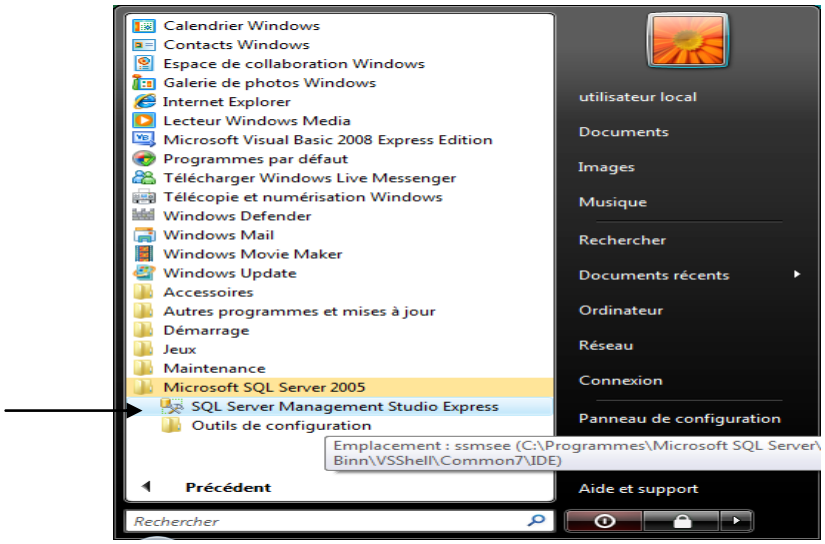


- Dans l'écran final, cliquez sur **Terminer**.

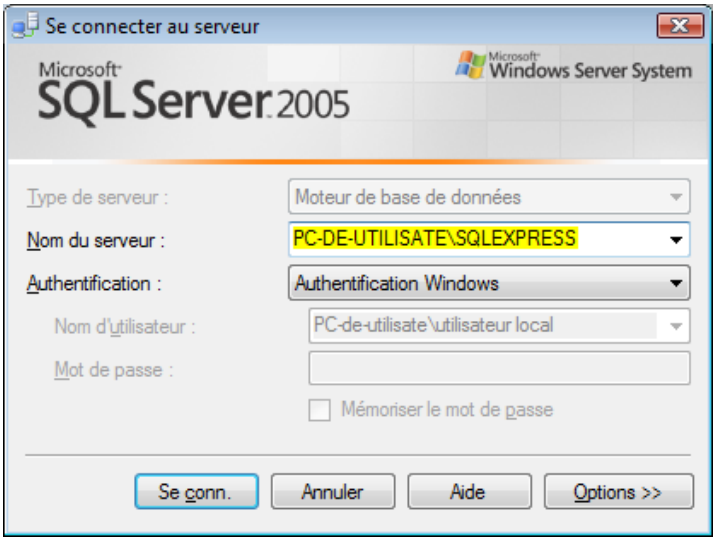


2. Vérifiez maintenant l'installation :

   - **Démarrer** > **Tous les programmes** > **Microsoft SQL Server 2005** > **SQL Server Management Studio Express.**



- Dans la boîte de dialogue **Se connecter au serveur**, entrez le nom de l'instance de votre serveur :



Il faut savoir que SQL Server Express s'installe par défaut comme une instance nommée, intitulée **SQLEXPRESS**.

La notion d'instance nommée vient de la capacité de SQL Server à s'installer plus d'une fois sur une même machine. On appelle **instance nommée** toute nouvelle instance de

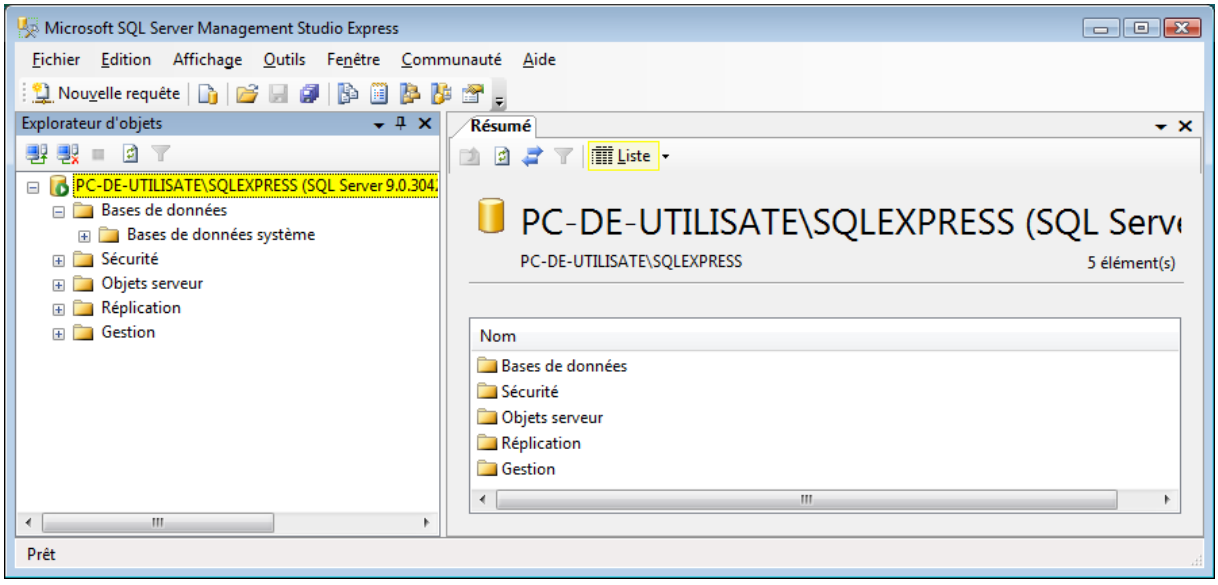
SQL Server installée sur une machine en plus de l'instance par défaut. Elle reçoit un nom pour la caractériser.

Pour vous connecter à Microsoft SQL Server 2005, il vous faut donc référencer cette instance nommée. Le nom complet est **<le nom de votre machine>\SQLExpress** (et non le nom de votre machine comme c'est le cas de l'instance par défaut avec Microsoft SQL Server).

Le format est : **<nom de la machine>\<nom de l'instance>**. Vous pourrez constater que les appellations dérivées suivantes fonctionnent toutes (la casse des noms importe peu) :

- **.\SQLEXPRESS**
- **(local)\SQLEXPRESS**
- **localhost\SQLEXPRESS**
- **<nom de la machine>\SQLEXPRESS**

- Cliquez sur **Se conn.** pour vous connecter.



Vous constatez que SQL Express s'installe sans **base de données d'exemple**. Mais cela ne veut pas dire qu'il n'en existe pas ☺. Voici un lien vers une base d'exemple hébergée sur le site **CodePlex** (site communautaire de projets OpenSource à l'initiative de Microsoft) :

http://www.codeplex.com/MSFTDBProdSamples/

Vous trouverez un didacticiel sur SQL Server Management Studio ici :

http://msdn2.microsoft.com/fr-fr/library/ms167593.aspx

Pour connaître les différences entre cette édition Express de l'outil et la version complète, rendez-vous sur :

http://msdn2.microsoft.com/fr-fr/library/ms365247.aspx

Explorer l’environnement de développement

Sommaire

SOMMAIRE

1 INTRODUCTION 3

1.1 CONTEXTE FONCTIONNEL 3

1.2 CONTEXTE TECHNIQUE 4

2 ET SI ON SE PASSAIT DE L'IDE… 4

3 ET SI ON ABUSAIT DE L'IDE POUR… 23

3.1 GÉRER SES PROJETS 24

3.2 EDITER LE CODE 55

3.3 COMPILER LE CODE 84

3.4 DÉBOGUER LE CODE 110

3.5 ET PLUS ENCORE 119

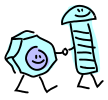
INTRODUCTION

Cet atelier s'inscrit dans le cadre du tutorial du coach Visual Basic dont l'objectif est la découverte et l'utilisation du langage Visual Basic (VB), actuellement en version 9.0 avec Visual Studio 2008, pour la construction d'applications avec une approche orientée objet.

VB 9.0 est une évolution du langage Visual Basic (que vous connaissez peut-être ou peut-être pas, ce n'est pas un pré requis du tout pour vous lancer dans ce tutorial) qui permet de créer des applications basées sur le .NET Framework.

Avec le langage VB vous pouvez construire tout type d'applications : Windows, Web, des services, etc. Dans le cadre de ce tutorial, nous allons créer une application Windows, mais uniquement en support d'illustration des points abordés.

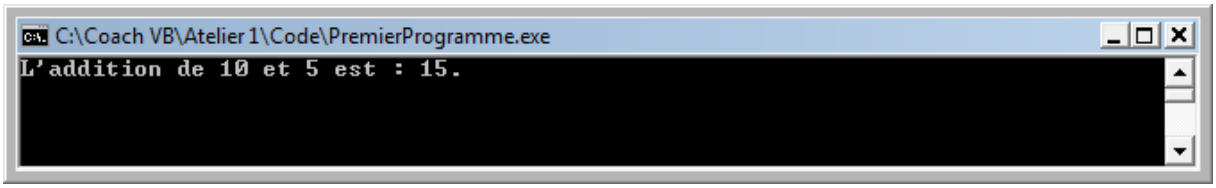
Une des caractéristiques les plus importantes du langage VB est que c'est un langage qui permet de développer vite (et bien) c'est-à-dire sans trop de contrainte donc avec un maximum de productivité. Dans ce tutorial, chaque fois qu'une fonctionnalité de VB ou de Visual Studio permettant de gagner du temps est illustrée, vous verrez le petit logo en marge.



Une des avancées les plus importantes du langage Visual Basic avec l'arrivée de cette nouvelle génération est que le langage est maintenant conçu pour générer des applications orientées objet. Si vous appréhendez le passage à cette autre approche de programmation, mettez carrément de côté tout apriori sur la question et laissez vous guider par ce tutorial ☺. Chaque fois que le sujet sera abordé, vous verrez le petit logo en marge.

CONTEXTE FONCTIONNEL

Le programme que nous vous proposons de développer dans ce premier atelier est un calculateur qui s'exécute en mode console et dont l'objectif est de calculer le résultat de la somme de deux nombres entiers (si vous vous souvenez de vos tables d'addition, ça devrait le faire donc ☺).



CONTEXTE TECHNIQUE

Pour bien appréhender le langage, il faut être familiarisé avec l'environnement de développement. Dans le contexte présent, il s'agit bien sûr de Visual Studio.

Peut-être l'avez-vous déjà ouvert et vous vous êtes fait une frayeur en pensant que ce n'était pas pour vous. L'objectif de cet atelier est évidemment de vous persuader du contraire ☺. Personnellement (cela n'engage que moi), chaque fois que je me retrouve à développer sans Visual Studio, je me sens comme un fermier qui doit retourner son champ avec pour seul outil une binette plutôt qu'une charrue.

A la fin de cet atelier, vous saurez comment :

- Gérer vos projets dans une solution,
- Naviguer dans les différentes fenêtres de Visual Basic Express,
- Utiliser les fonctionnalités de l'éditeur de code pour développer vite,
- Compiler et exécuter vos projets,
- Déboguer pas à pas le code de vos projets.

La solution de cet atelier est disponible dans le répertoire **..\Atelier 1\Solution**. La première partie est dans le sous-répertoire **sans IDE** et la seconde dans le sous-répertoire **avec IDE**.

Les fichiers utiles, auxquels font référence les exercices sont disponibles dans le répertoire **..Atelier 1\Fichiers utiles**.

ET SI ON SE PASSAIT DE L'IDE…

Et oui après tout ! Est-ce qu'on peut se passer de Visual Studio pour développer en VB ? Contrairement aux aprioris, la réponse à cette question est oui. Vous pourriez développer vos projets .NET avec un éditeur comme le Bloc-notes ! Mais ne vous méprenez pas, l'idée est de vous montrer à quel point ce serait une hérésie…

Dans cet exercice, vous allez apprendre à :

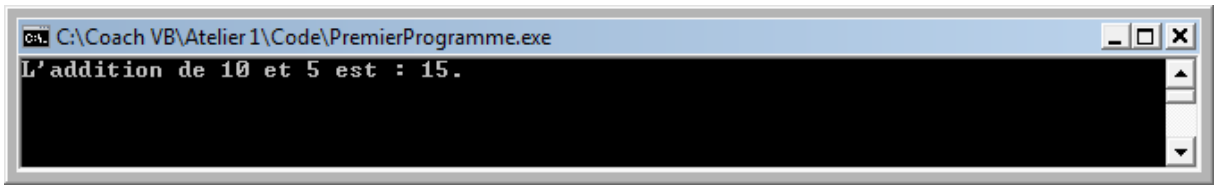
- Développer un premier programme écrit en VB
- Compiler et exécuter un programme à l'aide du .NET Framework

Objectif

Avant de se lancer à la découverte de Visual Studio, l'objectif de ce premier exercice est de positionner les éléments de base qui sont impliqués dans le développement d'une application avec le langage VB.

Contexte fonctionnel

Nous allons dès cette première partie nous lancer dans le développement du calculateur qui s'exécute en mode console :



Déroulement de l'exercice :

De quoi avez-vous besoin pour développer un programme écrit en VB ?

S'il on va à l'essentiel, il vous faut :

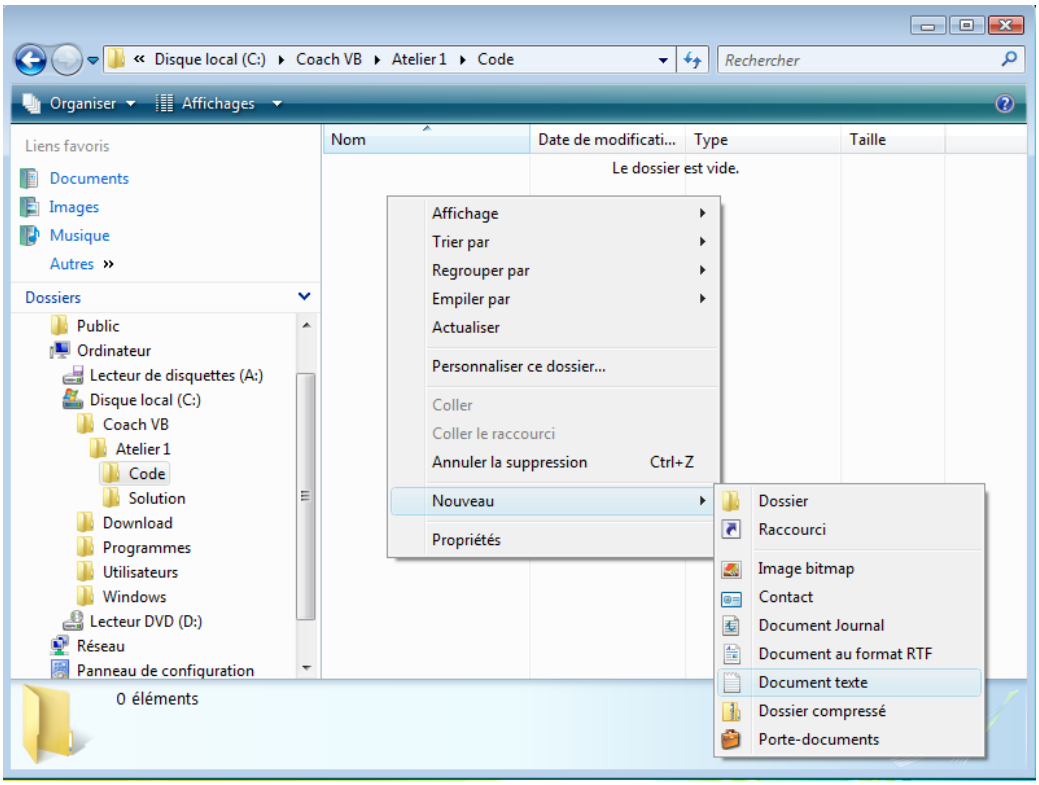
- Un éditeur pour coder
- Un compilateur pour convertir votre code en code exécutable
- Un environnement d'exécution (runtime)

Contrairement à ce qu'on pourrait croire, tous ces ingrédients sont fournis non pas par Visual Studio mais par le Framework .NET ! Sans lui, rien ne marche ! C'est d'ailleurs pour cela qu'il est installé automatiquement au moment de l'installation de Visual Studio (cf. procédure d'installation de l'environnement fournie avec ce tutorial). Vous auriez d'ailleurs tout aussi bien pu l'installer seul, sans l'IDE.

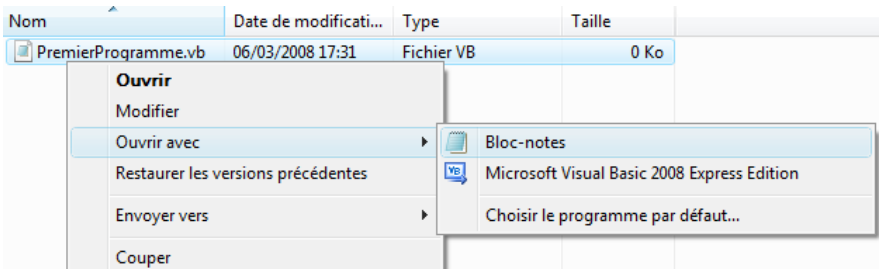
Pour vous procurez le Microsoft .NET Framework 3.5 seul :

http://www.microsoft.com/downloads/details.aspx?displaylang=fr& FamilyID=333325fd-ae52-4e35-b531-508d977d32a6

1. Créez un premier programme :
   - Ouvrez **l'Explorateur Windows**.
   - Sélectionnez un répertoire de travail (par exemple **C:\Coach VB\Atelier 1\Code**).
   - Faites un clic droit **Nouveau** > **Document texte**.



- Renommez le fichier en **PremierProgramme.vb**.
- Faites un clic droit sur **PremierProgramme.vb**, et sélectionnez l'option **Ouvrir avec** > **Bloc-notes** :



- Ajoutez le code suivant :

Code VB.NET

```
Module Programme

     Sub Main()

     End Sub

End Module
```

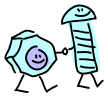
Comme dans tout programme, vous devez indiquer au runtime le point d'entrée de l'application. C'est l'objectif de la procédure **Main** qui contient les premières lignes de code à exécuter au lancement de l'application.

Pour en savoir plus sur la procédure **Main** :

http://msdn2.microsoft.com/fr-fr/library/ms235406.aspx

Les mots clés **Sub** et **End Sub** délimitent le contenu de la procédure **Main**. VB utilise des mots clés qui fonctionnent par paire, du type **X / End X**, pour structurer le programme en blocs de code. Nous verrons dans ce tutorial que les blocs de code sont fondamentaux pour délimiter la portée des éléments du programme.

Ce type d'écriture par paire de mots clés peut vous paraître lourd mais nous verrons dans la suite de cet atelier que Visual Studio s'occupera bien volontiers pour nous de fermer un bloc par le mot clé adapté précédé de **End**.



Un bon réflexe lorsqu'on code avec un langage orienté objet, est de toujours penser à placer son code dans un containeur quelconque. Dites vous bien qu'une procédure qui se balade toute seule dans un fichier, ça n'a pas de sens ! Ici pour démarrer en douceur, avant de se lancer dans la programmation basée sur les classes, nous allons nous autoriser à utiliser l'instruction **Module** de VB. Mais profitez en bien, car c'est la première

et dernière fois ☺. Avec un module, il n'est donc pas question encore d'objet et c'est un peu comme si pour l'instant on continuait à programmer à l'ancienne mais au moins, on a un containeur pour exécuter le programme.

Si vous voulez en savoir plus sur les différences entre les **Modules** et les **Classes** :

http://msdn2.microsoft.com/fr-fr/library/7825002w.aspx

- Ajoutez les lignes suivantes au programme :

Code VB.NET

```
Module Programme

    Sub Main()

        System.Console.WriteLine("Bonjour à tous")

        System.Console.ReadKey()

    End Sub

End Module
```

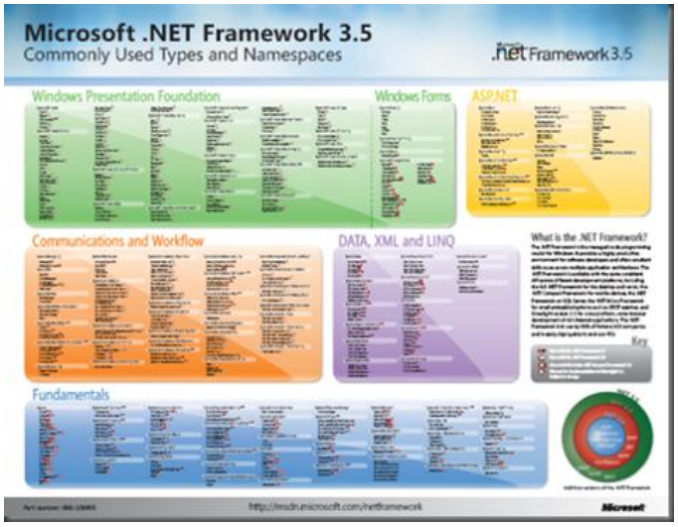
Quel est l'objectif fonctionnel de ces deux lignes ?

Affichez le message « Bonjour à tous » dans une fenêtre de commande (console) puis bloquez la console le temps de lire le message. Nous ferons notre calculateur plus tard dans cet atelier.

Rien de bien sorcier donc… sauf que vous savez écrire sur une fenêtre de commande vous ? Moi, pas ! Heureusement, nous avons à disposition toute une palette de classes fournies par le .NET Framework dont l'objectif est de nous épargner l'écriture du code pour toutes les actions élémentaires de ce type, de façon à ce que vous puissiez concentrer toute votre attention sur l'écriture du code *métier* de votre application.

Ce qu'il faut comprendre, c'est que la méthode **WriteLine** de la classe **System.Console** du .NET Framework, est exactement la même quelque soit le langage que vous utilisez. Elle serait par exemple utilisable telle quelle en C#. Le langage apporte par contre des différences de syntaxe et de structure du code.

Cliquez sur l'image pour télécharger le poster des types et espaces de noms du Framework 3.5 les plus couramment utilisés (donc ils sont loin d'y être tous ☺) :



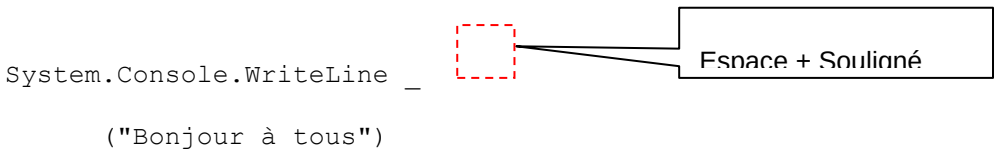
Revenons aux spécificités de VB…

En français, les phrases commencent par une majuscule et se terminent par un point. En VB, une instruction est délimitée tout simplement par la ligne. Facile !

L'instruction suivante, écrite sur deux lignes, génère donc une erreur de compilation :

```
System.Console.WriteLine
      ("Bonjour à tous")
```

Pour éclaircir le code, vous pouvez bien sûr découper une instruction sur plusieurs lignes. Dans ce cas, pensez à ajouter un espace suivi du signe _ (souligné) à chaque fin de ligne (excepté pour la dernière qui marque la fin de l'instruction) pour indiquer au compilateur que votre ligne de code se poursuit sur la ligne suivante.



Autre bonne nouvelle pour développer vite, VB ne tient pas compte de la casse des noms. Aussi les deux lignes suivantes sont absolument identiques pour le compilateur VB.NET :

```
system.console.writeline("Bonjour à tous")

System.Console.WriteLine("Bonjour à tous")
```

En revanche, si on est honnête, il faut reconnaître que si vous utilisez la notation **Pascal** qui consiste à commencer chaque mot par une majuscule, vous obtenez un code beaucoup plus clair et lisible. Là encore, on pourra s'appuyer sur Visual Studio pour nous aider à ne pas perdre de temps.

Pour en savoir plus sur les conventions de nommage des variables du .NET Framework : cliquez ici

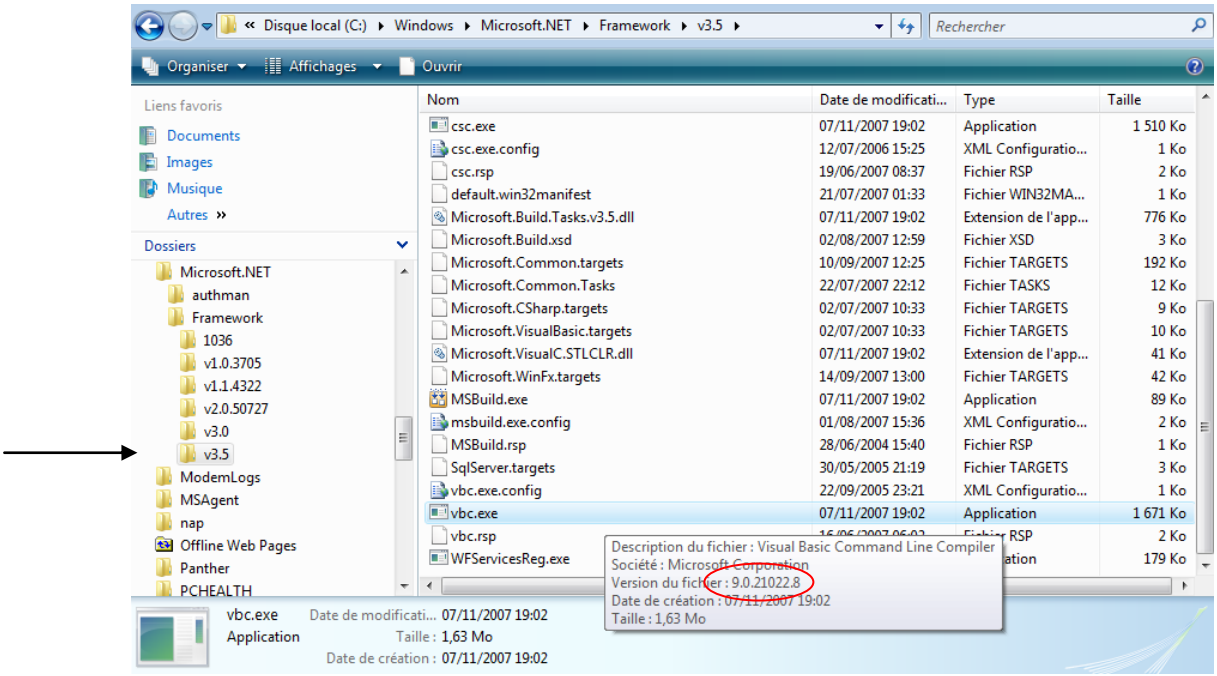
- Sauvegardez votre programme par les menus du Bloc-notes.

2. Compilez le programme :

Où trouvez le compilateur VB ?

Pour rappel, il est fourni non pas par Visual Studio mais par le .NET Framework (qui s'est installé en même temps que vous avez installé Visual Basic 2008 Express Edition (ou autre édition 2008 de Visual Studio)).

- Pour voir les fichiers installés par le .NET Framework, ouvrez Windows Explorer et rendez-vous sous le dossier **c:\WINDOWS\Microsoft .NET**.



Attention dans ce même répertoire, vous constatez qu'il y a en réalité plusieurs versions du .NET Framework installées.

Les éditions 2008 de Visual Studio installent la version 3.5 qui correspond au sous répertoire **v3.5**. Mais cette version du Framework ne fonctionne pas toute seule. Elle s'appuie sur les précédentes versions du Framework, notamment du point de vue du runtime d'exécution, en lui apportant uniquement des extensions nécessaires pour prendre en compte les nouveautés des langages, Linq et diverses autres améliorations.

Néanmoins, si cette version du Framework n'est pas autonome, vous y trouverez

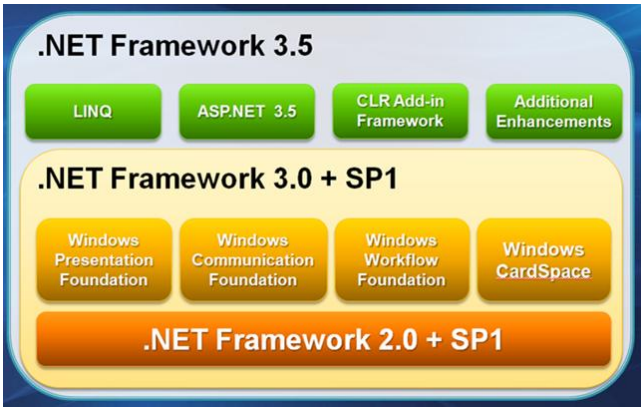
cependant un compilateur VB qui lui est propre, le programme **vbc.exe**, qui prend en compte les nouvelles structures de la toute dernière version du langage (jetez un œil sur l'info bulle du fichier et vous verrez que VB est maintenant en version 9.0). C'est ce programme qu'il nous faut pour compiler notre projet !

A quoi servent les autres versions du Framework ?

- La version 2.0, installée dans le sous répertoire **v2.0.50727**, constitue le noyau de base du Framework sur lequel s'appuient toutes les autres versions du Framework. Vous y trouverez donc également une version du compilateur vbc.exe pour compiler vos projets dans la version précédente de VB.

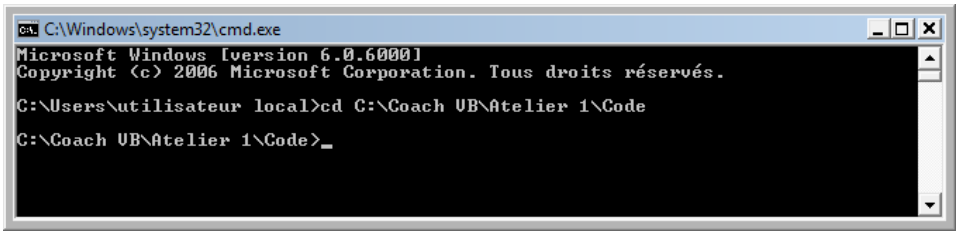
- La version 3.0, qui est installée par exemple en standard avec Windows Vista, fonctionne comme la version 3.5 *au-dessus* de la version 2.0. C'est tellement vrai que si vous regardez le contenu du dossier **v3.0** correspondant, vous constaterez qu'il ne contient que trois dossiers associés aux trois briques supplémentaires fournies par la version 3.0 : **Windows Presentation Foundation**, **Windows Communication Foundation**, et **Windows Workflow Foundation**. Il n'y a même pas de compilateur, cette version du Framework n'apportant aucune nouvelle structure dans les langages, donc s'appuyant sur le compilateur de la version 2.0.

Voici un petit schéma qui récapitule l'imbrication des Frameworks entre eux :



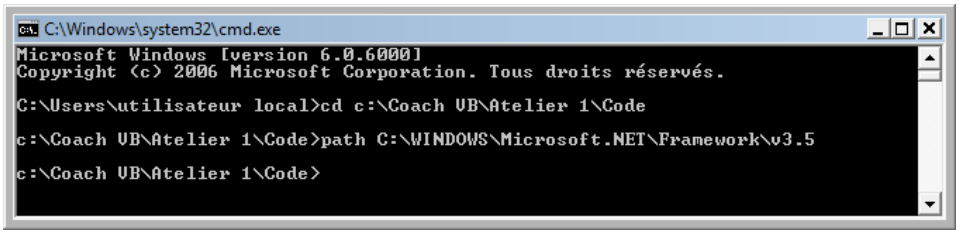
- Lancez l'**Invite de commandes** depuis la liste des programmes de Windows (ou tapez **cmd** dans la zone de recherche de démarrage de Vista).

- A l'intérieur de la nouvelle invite de commande, positionnez-vous dans le sous répertoire contenant votre programme – par exemple, tapez l'ordre **cd C:\Coach VB\Atelier 1\Code**.

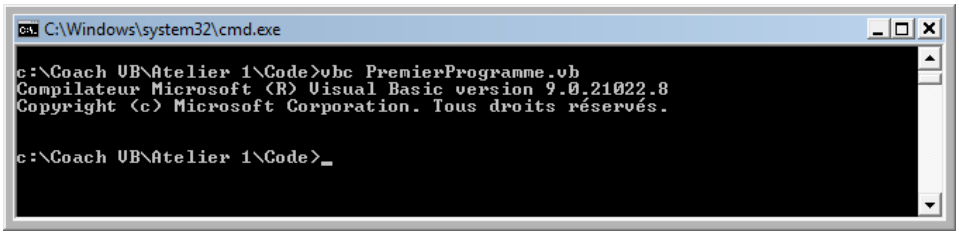


- Indiquez à Windows Vista que vous allez utiliser des commandes d'exécution situées sous le répertoire contenant le compilateur VB, en tapant l'ordre suivant :

  path C:\WINDOWS\Microsoft.NET\Framework\v3.5

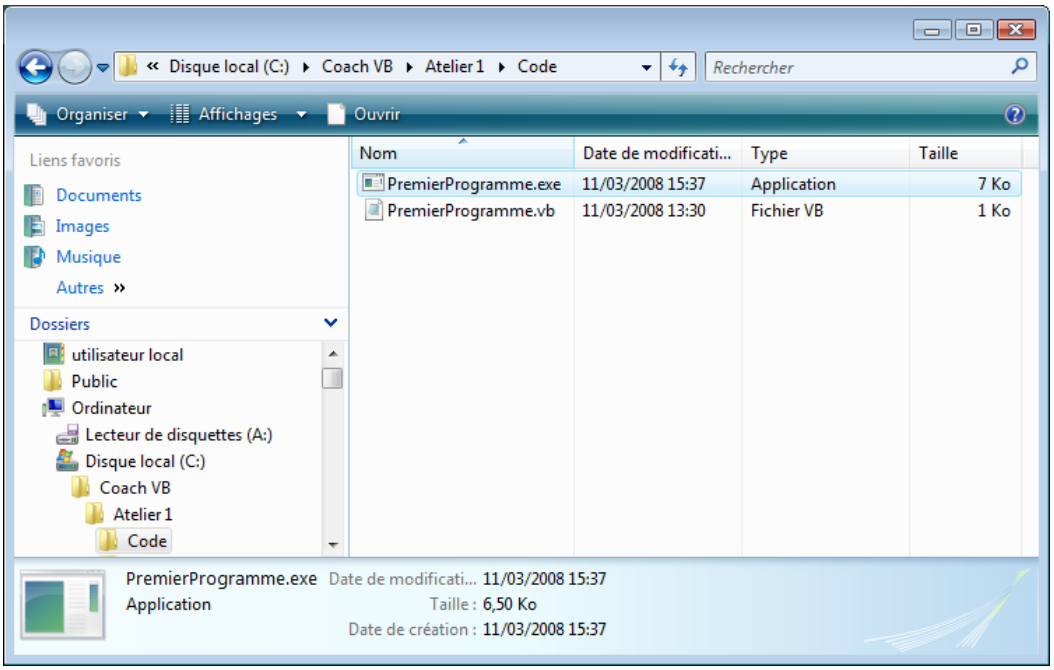


- Compilez votre programme avec l'ordre **vbc premierprogramme.vb** ;



Si vous avez fait une erreur de codage, le compilateur vous l'indique. Il vous reste à corriger la ligne ☺.

- Conservez l'Invite de commandes ouverte.
- Avec l'explorateur Windows, naviguez jusqu'à votre sous répertoire de travail. Un nouveau fichier **PremierProgramme.exe** a été généré par le compilateur.

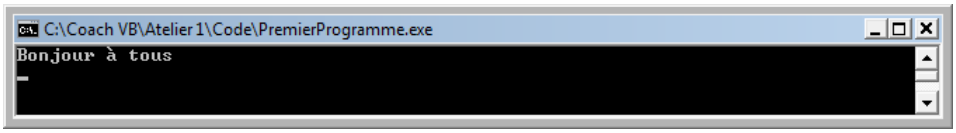


Plutôt que de parler d'*exécutable*, on dit que le compilateur a *assemblé* le code dans un fichier d'extension .exe (ou .dll s'il s'agit d'une librairie). C'est pour cette raison que nous appelons les programmes compilés avec .NET des ***assemblies*** (assembly est un mot qui vient de la littérature anglaise).

Pour en savoir plus sur ce qu'est un assembly :

http://msdn2.microsoft.com/fr-fr/netframework/k3677y81.aspx

- Double-cliquez le programme **PremierProgramme.exe**, et une nouvelle console s'affiche, en vous souhaitant : Bonjour à tous !



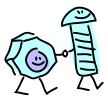
Qui a pris en charge l'exécution de votre programme ?

Encore et toujours lui…le .NET Framework ! Plus précisément, il s'agit de l'environnement d'exécution fourni par le .NET Framework qu'on appelle le CLR pour **C**ommon **L**anguage **R**untime.

Comme l'indique son nom (Common Language), il est le même quelque soit le langage .NET que vous avez utilisé pour coder le programme. En effet, lors de la compilation, le compilateur du langage convertit le code source en langage MSIL (**M**icrosoft **I**ntermediate **L**anguage) c'est-à-dire en un langage intermédiaire indépendant du processeur, qui est ensuite converti en langage natif par le CLR. On dit que le programme que vous avez développé en VB est écrit en *code managé* pour signifier que le code est géré (*managé*) par le CLR du .NET Framework.

3. Codez le calculateur dans une librairie :

Pour rappel, l'objectif de cette première application est d'effectuer le calcul de la somme de deux nombres entiers. Le plus simple est donc de coder une fonction qui attend en paramètres deux nombres entiers et renvoie le résultat de la somme de ces deux nombres en valeur de retour.



On peut supposer que cette fonction d'ajout pourrait être utile dans plusieurs autres applications, aussi un bon réflexe est d'externaliser cette fonction dans un projet séparé sous la forme d'une **librairie** de sortes qu'elle puisse être partagée par tous les programmes qui en ont besoin.

Nous aurons l'occasion de revenir sur cette bonne pratique qui est extrêmement intéressante lorsqu'on développe une application avec une approche orientée objet. En effet, une solution complexe nécessite souvent plusieurs fichiers différents pour exprimer des besoins différents ou pour partager des éléments d'un projet à l'autre.

- Dans votre répertoire de travail (par exemple **C:\Coach VB\Atelier 01\Code**), créez donc un nouveau programme nommé **Calculateur.vb**.
- En utilisant le Bloc-notes, ajoutez le code suivant, qui ajoute un calculateur capable de faire une addition ;

Code VB

```
Public Class Calculateur

   Shared Function Ajouter(ByVal valeur1 As Integer, _
                           ByVal valeur2 As Integer) As Integer

      Return valeur1 + valeur02

   End Function

End Class
```

Termine la fonction avec la valeur de retour spécifiée

Détermine le type de la valeur de retour

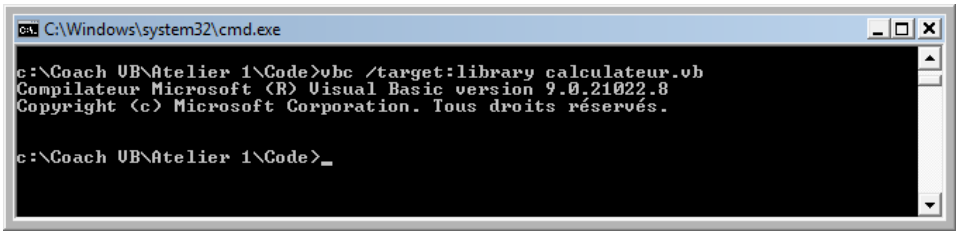
Notez que ce programme ne contient pas de procédure **Main**. En effet, c'est une librairie qui va être appelée par un exécutable extérieur, et donc elle n'a pas besoin d'avoir de point d'entrée. Les librairies possèdent une extension **.dll**.

Le calculateur est codé sous la forme d'une **classe** plutôt que d'un **module** comme précédemment. Mais le mot clé **Shared** devant la définition de la fonction va nous permettre de travailler d'une manière quasi identique au module dont tous les membres sont implicitement **Shared**. Nous reviendrons plus en détails sur ce sujet dans ce tutorial.

Contrairement au module, vous devez préciser le mot clé **Public** pour indiquer que la classe sera visible de l'extérieur par tout programme qui utilise la librairie. Nous reviendrons plus en détail sur la notion de visibilité lors de ce tutorial.

- Sauvegardez ce nouveau programme.

- Avec l'invite de commandes précédemment ouverte, compilez la librairie en indiquant que vous souhaitez obtenir un fichier avec une extension **.dll**, en utilisant l'ordre suivant : **vbc /target:library calculateur.vb**.

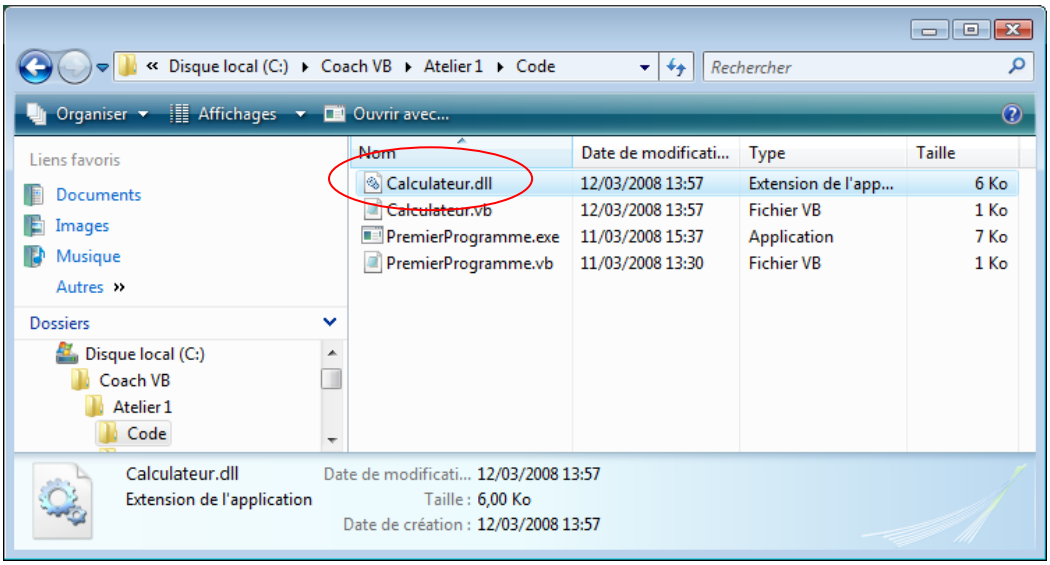


S'il existe une erreur de codage, le compilateur vous l'indique. Corrigez le programme en conséquence.

Lors de la compilation du programme initial, nous avions utilisé le compilateur avec ses options par défaut. C'est pour cela qu'un fichier d'extension **.exe** avait été généré. Ici, nous utilisons l'option **/target** pour générer un fichier d'extension **.dll**.

Pour voir l'ensemble des options disponibles avec le compilateur, tapez l'ordre **vbc /?**.

- Laissez l'Invite de commandes ouverte.
- Avec l'explorateur Windows, naviguez jusqu'à votre sous répertoire de travail. Un nouveau fichier **Calculateur.dll** a été généré par le compilateur.



4. Utilisez la librairie de calcul à partir du programme initial en mode console pour effectuer un calcul entre deux nombres entiers :
   - En utilisant le Bloc-notes, rouvrez le fichier **PremierProgramme.vb**.
   - Modifiez le code initial de façon à utiliser le calculateur :

Code VB

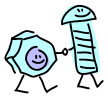
```
Module Programme


   Sub Main()


      System.Console.Writeline("Bonjour à tous")
      System.Console.WriteLine("L'addition de 10 et 5 est : {0}.",  _
                                    Calculateur.Ajouter(10,5))
      System.Console.readKey()
   End Sub


End Module
```

L'espace puis _ indique que l'instruction se poursuit sur la ligne suivante.

La commande est ici écrite sur plusieurs lignes afin d'améliorer la lisibilité dans le cade de ce document. Vous pouvez bien sûr l'écrire en une seule ligne (sans le caractère souligné de fin).

La commande utilise les possibilités de formatage des chaînes de caractères en passant les paramètres entre accolades. Plus d'informations sur le formatage des chaînes est disponible sur :

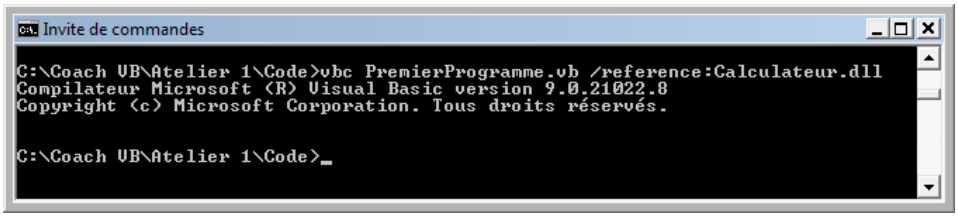
http://msdn2.microsoft.com/fr-fr/library/b1csw23d(VS.80).aspx



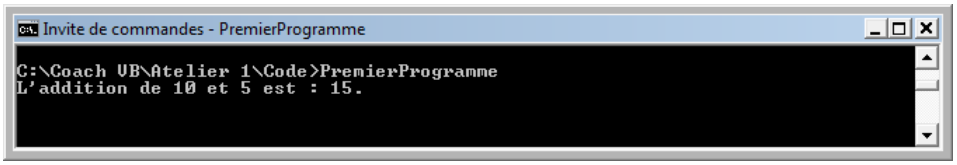
Pour invoquer la fonction **Ajouter**, ici rien de plus simple puisque pour rappel, nous avons utilisé le mot clé **Shared**. Il suffit d'invoquer la fonction en la précédent du nom de la classe. Encore une fois, ne vous inquiétez pas, nous reviendrons plus longuement sur ces principes objet dans ce tutorial.

- Sauvegardez le programme.
- A partir de l'Invite de commandes, compilez de nouveau **PremierProgramme.vb**, mais en indiquant maintenant que vous voulez référencer la librairie qui contient le calculateur. L'ordre à taper est le suivant :

  vbc PremierProgramme.vb /reference:Calculateur.dll



- A partir de l'invite de commande, tapez **PremierProgramme**. le résultat du calcul s'affiche.



Félicitations ! Vous avez écrit votre premier programme en VB !

5. Déployez le projet dans un autre répertoire :
   - Utilisez l'Explorateur de Windows pour copier les deux fichiers **PremierProgramme.exe** et **Calculateur.dll** puis collez-les dans un autre répertoire du disque.
   - Exécutez le programme **PremierProgramme** pour vérifier qu'il fonctionne toujours correctement.

En déplaçant le projet sur un autre emplacement du disque, vous venez de le *déployer*.

Pour faire tourner ce même programme sur la machine de quelqu'un d'autre, il vous suffirait de transmettre les deux fichiers (par exemple par email) au destinataire qui, à condition bien sûr d'avoir la version correspondante du Framework d'installée en local (sans environnement d'exécution, pas question d'exécuter quoique ce soit), pourrait exécuter votre programme directement.

Comment un simple copier/coller permet-il de déployer son code ?

En effet, bien qu'une partie du projet soit une dll, vous remarquez qu'il n'est pas nécessaire d'enregistrer celle-ci dans le registre de la machine pour l'utiliser.

Le principe est que lorsque le compilateur génère le code en langage intermédiaire MSIL, il génère en même temps ce qu'on appelle des *métadonnées*, c'est-à-dire des données qui décrivent tout ce que le runtime doit savoir pour exécuter votre programme (par exemple les références externes dont votre code a besoin). Du coup, La présence de métadonnées dans le fichier en même temps que le jeu d'instructions MSIL permet à votre code de se décrire lui-même, sans nécessité de faire intervenir une autre entité comme le Registre de Windows.

- Pour terminer, déplacez également les codes sources des deux projets dans le nouveau répertoire sur lequel vous avez déployé les exécutables, afin de libérer le répertoire **Atelier 1** pour la suite de l'exercice.

En résumé, dans cette première partie, nous avons vu qu'il suffit d'installer la dernière version du **.NET Framework** sur votre environnement de travail pour développer en VB. Il vous fournit :

- le **compilateur**
- puis s'occupe **d'exécuter** le programme,
- et en prime vous apporte toute une **batterie de classes** pour coder les instructions de base (telles qu'écrire une information sur la console de l'Invite de commandes Windows).

Oui, mais là, nous n'avons tapé qu'une petite dizaine de lignes de code...Imaginons maintenant un projet constitué de plusieurs dizaines de librairies, elles-mêmes fournissant une multitude de classes avec des milliers de lignes de code ☹. Franchement, sans IDE, ce n'est même pas la peine d'y penser !

ET SI ON ABUSAIT DE L'IDE POUR…

Que signifie un **IDE** ?

IDE (**I**ntegrated **D**evelopment **E**nvironmment) signifie Environnement de Développement Intégré. Visual Studio est un IDE c'est-à-dire qu'il vous fournit un **environnement de développement** complet qui regroupe tout ce dont vous avez besoin pour développer vos projets.

Attention ! On parle bien d'un *environnement* et pas seulement d'un éditeur de code. Il s'agit de développer sur la base d'un seul outil (et ce quelque soit votre plateforme) qui vous aide à développer de manière productive du code de qualité.

Vous avez vu précédemment que le compilateur et l'environnement d'exécution sont fournis par le .NET Framework donc Visual Studio s'appuie clairement sur ce dernier pour proposer des fonctions de compilation et d'exécution pour tester vos programmes. Du coup, son rôle consiste plutôt à vous *simplifier la vie*, et ce à tous les niveaux, que vous développiez seul ou à plusieurs, que vous codiez, testiez ou déployiez l'application, que vous développiez une solution simple ou complexe intégrant par exemple des accès au moteur de base de données SQL Server.

Dans cet exercice, vous allez apprendre à :

- Créer puis gérer une solution de projets,
- Repérer et utiliser les différentes fenêtres de Visual Studio,
- Travailler avec l'éditeur de code,
- Générer un projet,
- Déboguer et tester un projet,

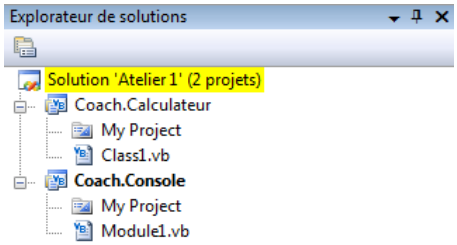
Objectif

L'objectif de cet exercice est de prendre en main les fonctionnalités de base de Visual Studio pour développer une application en VB.

Contexte fonctionnel

L'objectif fonctionnel est rigoureusement le même que pour l'exercice précédent à savoir développer un calculateur qui s'exécute en mode console et qui effectue la somme de deux nombres entiers.

GERER SES PROJETS



Dans l'exercice précédent, vous avez créé deux programmes séparés : un programme d'extension .exe et une librairie d'extension .dll.

L'objectif de ce premier exercice est de voir de quelle manière Visual Studio peut vous aider à créer puis gérer ces projets de manière optimisée.

A la fin de cet exercice, vous saurez :

- Identifier les différentes parties qui constituent la surface de travail de Visual Studio,
- Utiliser l'Explorateur de solutions pour organiser vos projets dans une solution.

Déroulement de l'exercice :

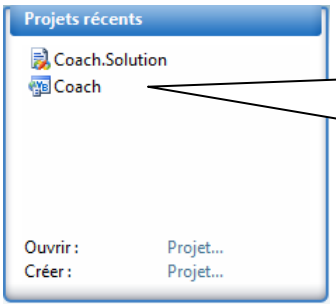
1. La première étape consiste à redévelopper le projet PremierProgramme en utilisant cette fois-ci Visual Studio :
    - Lancez Visual Studio à partir du menu **Démarrer** > **Tous les programmes** > **Microsoft Visual Basic 2008 Express Edition**.

Pour éviter la confusion entre le langage Visual Basic et l'IDE Visual Basic Express Edition, je vous propose d'adopter la terminologie suivante :

- Chaque fois qu'on parle du langage, on utilisera le terme *VB*.
- Chaque fois qu'on parle de l'IDE, on parlera de *Visual Studio* (VB Express Edition n'étant qu'une édition parmi celles proposées dans la gamme Visual Studio. Evidemment, elle est idéale pour ce tutorial puisqu'elle est tout particulièrement dédiée à la découverte du développement d'application en langage VB).

Lorsque vous lancez Visual Studio, vous tombez sur la **Page de démarrage** qui, outre le fait qu'elle donne des points d'entrée vers la communauté de développeurs, présente une fenêtre **Projets récents** pour créer rapidement un nouveau projet ou ouvrir en un seul clic vos projets récemment utilisés.

Pour en savoir plus sur cette page, cliquez ici.

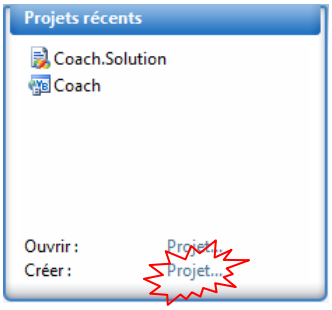


Votre liste de projets doit encore être vide mais dès le prochain démarrage, elle affichera un raccourci vers le projet sur lequel nous allons travailler maintenant (cf. ci-contre). Vous pourrez ainsi le rouvrir en un seul clic

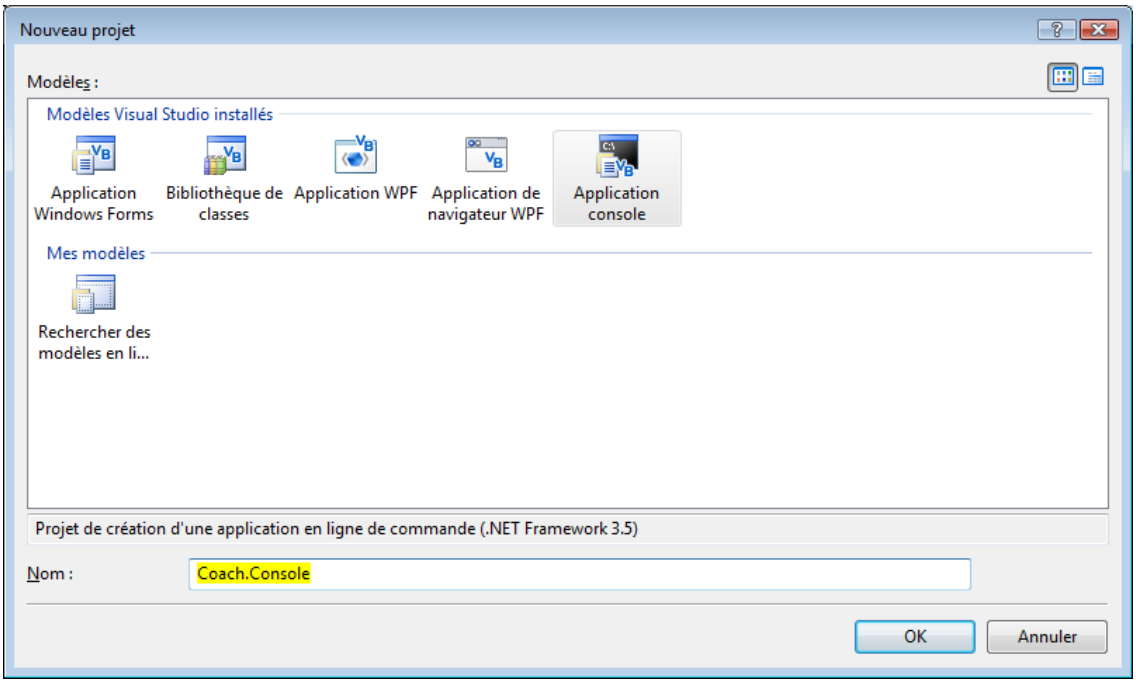
Juste un petit tuyau au passage : si vous voulez réinitialiser cette liste, il faut éditer le registre (à condition que vous ayez les privilèges adéquats) et détruire tous les éléments de la liste **ProjectMRUList** sous :

HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\<version>

- Créez un nouveau projet depuis l'option **Créer :** > **Projet…** de la page de démarrage ou à partir du menu **Fichier** > **Nouveau projet...**



- Dans la fenêtre **Nouveau projet**, sélectionnez le modèle de projet **Application Console** et indiquez **Coach.Console** comme nom de projet.



Dans cette fenêtre, plutôt que de démarrer de zéro, Visual Studio vous propose de charger un modèle de projet qui contient déjà tous les éléments de base pour développer un projet spécifique. Ici nous voulons développer une application en ligne de commandes, donc un projet de type **Application Console**, puis nous créerons par la suite un projet sur la base du modèle **Bibliothèque de classes** pour créer notre librairie de calcul.

Ne soyez pas surpris s'il n'y a pas de modèle de projet web alors qu'il est tout à fait possible de développer une application web en langage VB. Ce qui limite les modèles présents dans cette fenêtre, c'est uniquement l'édition de Visual Studio avec laquelle vous travaillez. Typiquement avec Visual Basic 2008 Express Edition, vous n'avez pas la possibilité d'éditer un projet web. Dans la même gamme d'éditions découverte, installez Visual Web Developer 2008 Express Edition pour cela.

- Validez par **OK**.

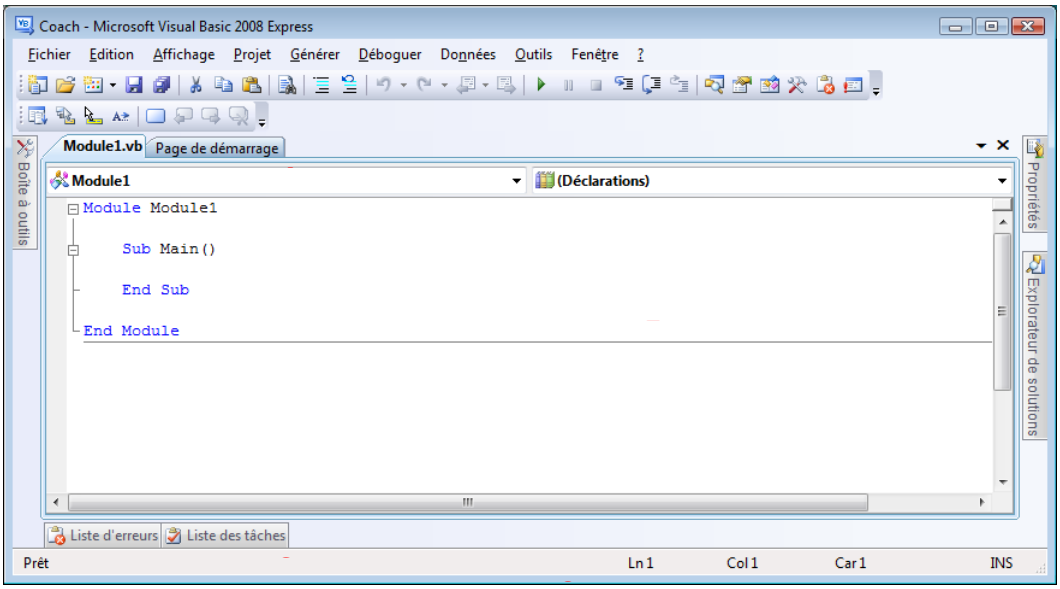
La liste des modèles disponibles en fonction des versions de Visual Studio est sur le lien :

http://msdn2.microsoft.com/fr-fr/library/b4z62wxz.aspx

Sachez également que vous pouvez développer vos propres modèles de projet en regroupant ce que vous utilisez systématiquement dans vos développements :

http://msdn2.microsoft.com/fr-fr/library/s365byhx.aspx

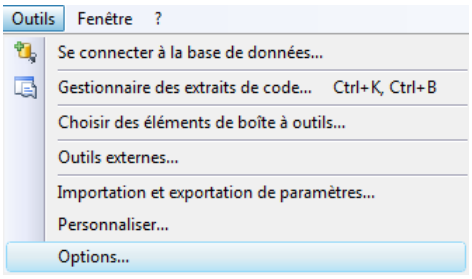
Visual Studio ouvre une surface de travail composée de plusieurs fenêtres et onglets. Notre fichier **Module1.vb** a été créé automatiquement sur la base du modèle de projet choisi, avec l'indispensable procédure **Main** qui constituera le point d'entrée du programme. Pour un peu, vous n'aviez plus rien à faire ☺.



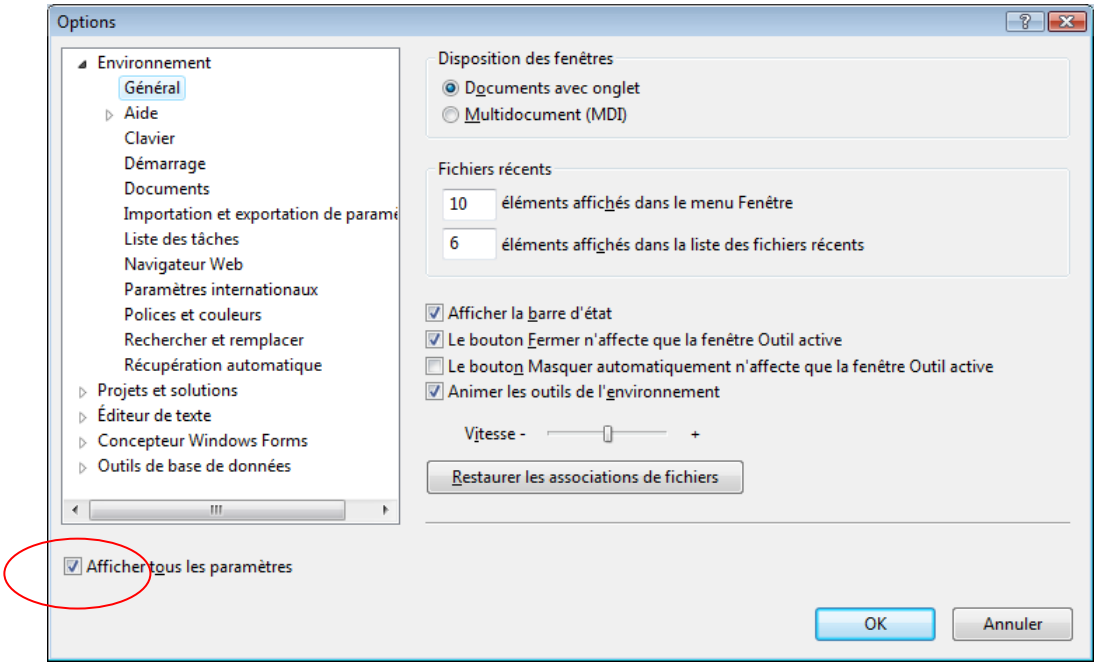
Ne vous laissez pas impressionner par cette multitude d'outils ! Si vous y réfléchissez bien, ce doit être guère plus le foutoire que sur votre bureau (avec la tasse de café en moins ☺). En quelque sorte, c'est votre *bureau de développement*.

- **1** : il s'agit de la barre de menu de Visual Studio. Vous reconnaissez les traditionnels menus **Fichier**/**Edition**/**Affichage** des outils Microsoft, et vous vous doutez bien que par exemple, avec le menu **Déboguer** vous trouverez tout ce qu'il faut pour déboguer le projet…

- Cliquez par exemple le menu **Outils > Options…** pour ouvrir la boîte de dialogue (classique des outils Microsoft) des options de configuration du produit :



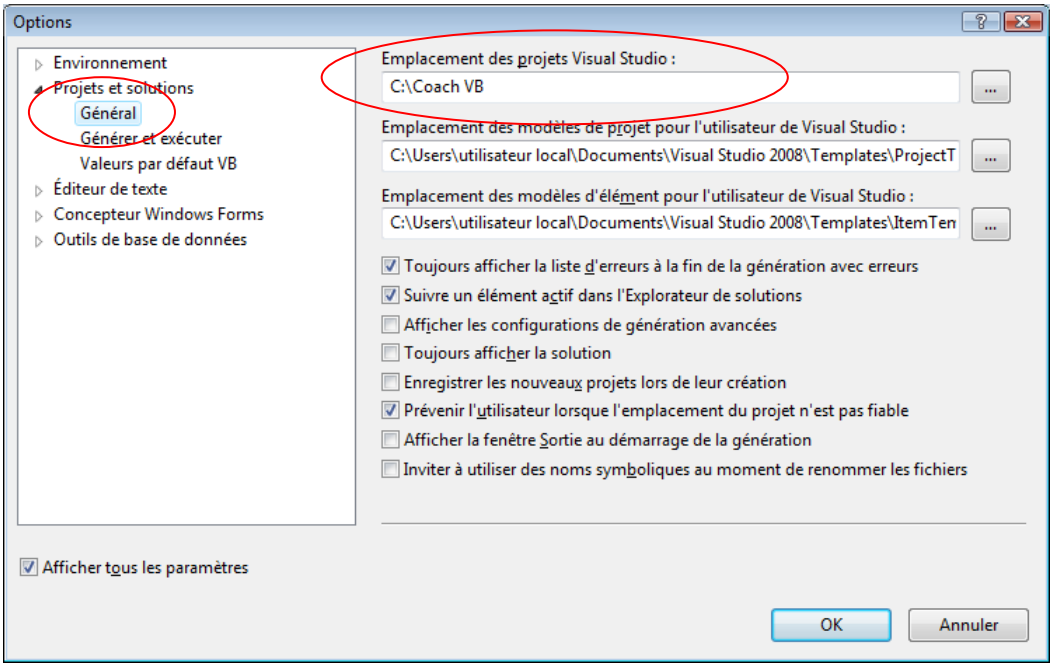
- Dans la boîte de dialogue **Options**, cochez la case **Afficher tous les paramètres** :



- Dans l'arborescence des options, cliquez sur **Projets et solutions** > **Général**.

Visual Studio vous propose une option pour indiquer le chemin de sauvegarde par défaut de tous vos projets Visual Studio. Ainsi chaque fois que vous enregistrerez un nouvel élément de projet, c'est ce chemin qui vous sera automatiquement proposé !

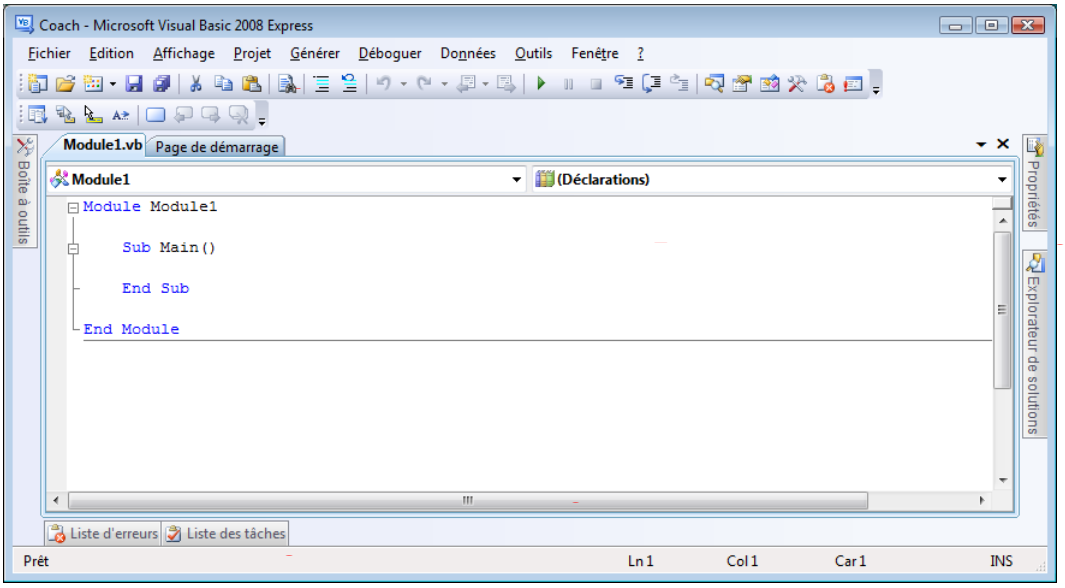
- Dans la zone **Emplacement des projets Visual Studio**, remplacez le chemin par celui où se trouvent vos codes (par exemple **C:\Coach VB**), en utilisant le bouton :



- Cliquez le bouton **OK** ;

Notez que cette boîte de dialogue des options présente plusieurs dizaines de paramètres que vous pouvez changer afin de modifier le fonctionnement de Visual Studio.

Poursuivons notre exploration du bureau :



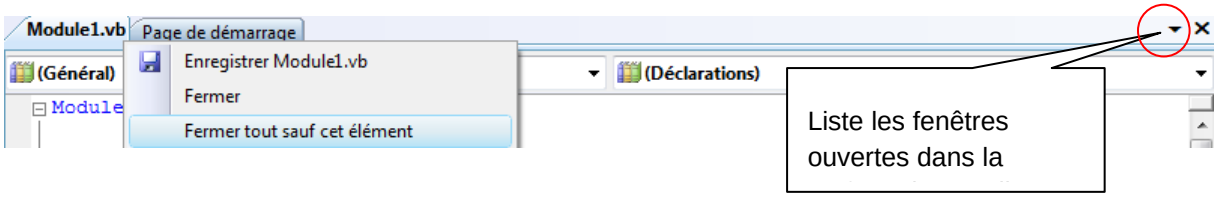
- **2** : il s'agit de la barre d'outils standard de Visual Studio avec des outils que vous connaissez comme les actions **Couper, Copier, Coller** et d'autres, plus spécifiques à Visual Studio comme qui nous servira à démarrer l'exécution du programme pour le tester.

- **3** : vous trouvez ici toute autre barre d'outils comme celle-ci, susceptible de s'afficher en fonction de ce que vous serez en train de faire apparaître en dessous de la barre standard. Pour l'instant, au centre de la surface, vous éditez un fichier de code, donc c'est la **barre d'outils de l'Editeur de texte** qui est actuellement affichée.

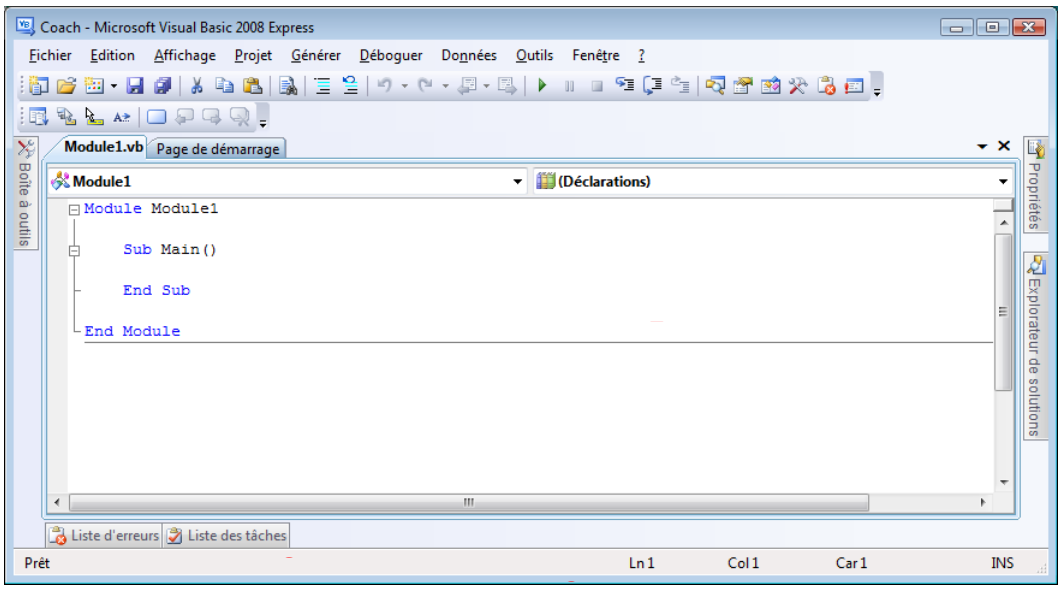
- **4** : Comme dans n'importe quel outil MDI, vous avez la possibilité d'éditer plusieurs fichiers en même temps sur la surface de travail, chacun apparaissant dans un nouvel **onglet**. Pour l'instant, vous avez donc deux « pages », celle du module VB et celle, toujours ouverte de la Page de démarrage de Visual Studio.

Comme dans tous les outils Microsoft, un bon réflexe consiste à faire un **clic droit** sur l'objet qui vous intéresse pour faire apparaître un **menu contextuel** avec des tas d'options utiles. Par exemple, lorsque vous commencez à accumuler les fichiers ouverts, un clic droit sur l'un des onglets offre la possibilité de fermer celui-ci (**Fermer**) ou de les

fermer tous d'un seul clic (**Fermer tout sauf cet élément**) en ne laissant ouvert que l'onglet que vous avez sélectionné.



Poursuivons notre exploration du bureau :

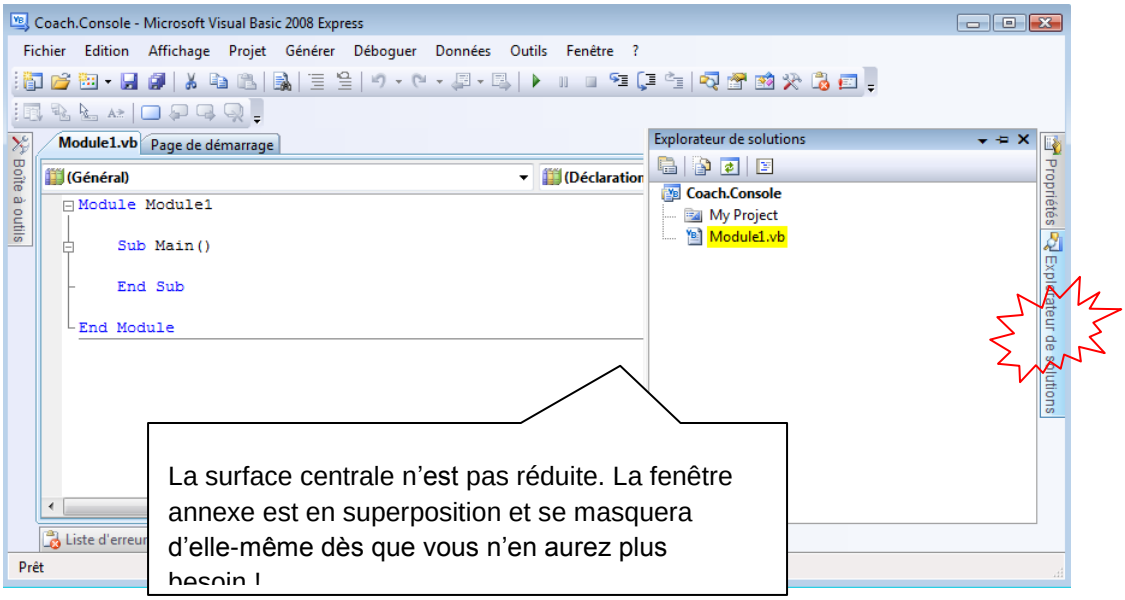


- **5** : La surface centrale est réservée à l'affichage de la fenêtre sur laquelle se concentre votre activité principale. Dans notre cas, nous allons coder le programme donc Visual Studio nous affiche par défaut le fichier contenant la fonction **Main** du projet.

- **6** et **7** : Pour optimiser l'espace, Visual Studio range sur le côté toutes les autres fenêtres proposant des fonctionnalités annexes à votre actuel travail dans la fenêtre centrale. C'est exactement comme lorsque vous poussez sur les côtés tout ce que vous avez sur votre bureau pour garder un maximum de place au centre pour le

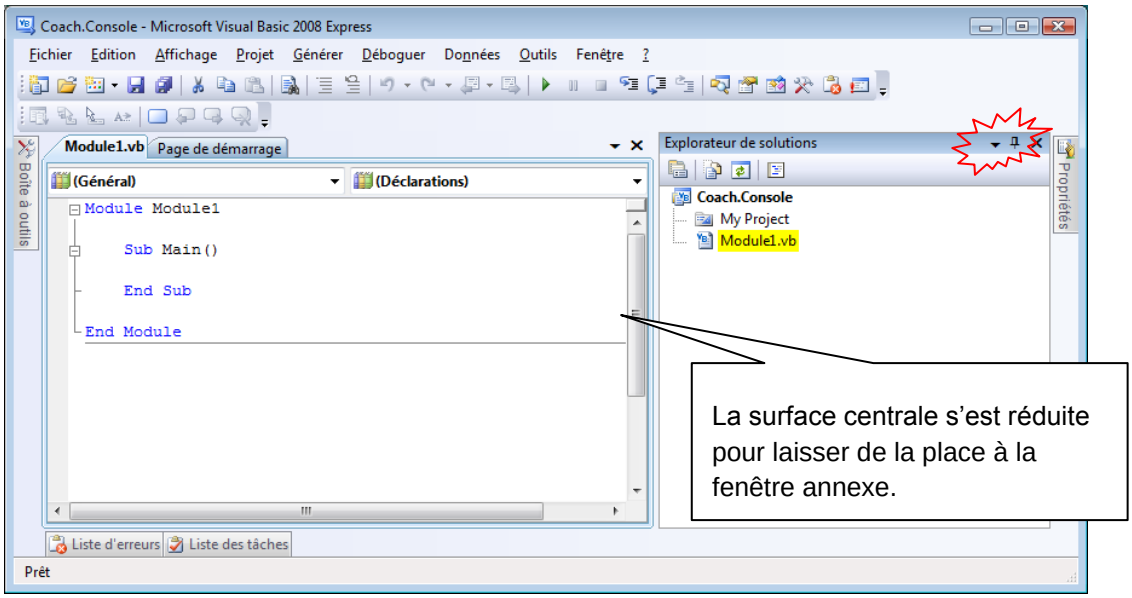
travail que vous réalisez.

Pour activer une fenêtre située sur les côtés, passez avec la souris (sans besoin de cliquer) sur la « poignée » de la fenêtre. La fenêtre qui était à l'état masqué jusque là, apparaît et glisse tout doucement par-dessus la surface centrale.



Déplacez la souris n'importe où à l'extérieur de la fenêtre qui s'est déployée pour la masquer à nouveau automatiquement. De cette manière, Visual Studio vous donne un accès rapide aux fonctionnalités de la fenêtre sans pour autant tronquer la surface de travail qui revient toujours à ses dimensions maximum.

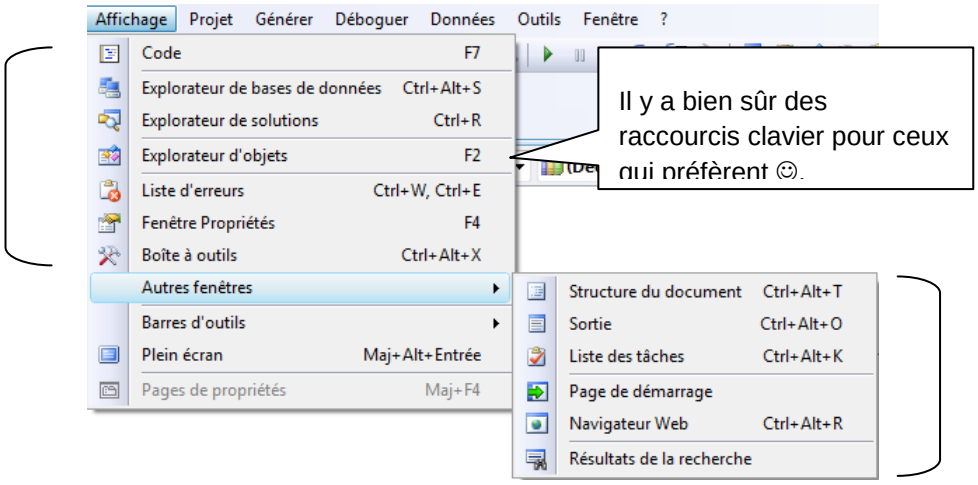
Si au contraire, vous souhaitez préserver l'affichage des deux fenêtres en simultanée, cliquez sur la punaise située dans la barre de titre de la fenêtre concernée (après l'avoir fait glisser sur la surface de travail). De , elle passe ensuite à pour indiquer que la fenêtre est en quelque sorte « punaisée » donc figée sur le bureau.



Cliquez pour la détacher à nouveau et la faire disparaître sur le côté.

Quelles sont ces fenêtres qui peuvent s'afficher sur les côtés ?

Toutes celles proposées dans le menu **Affichage** de Visual Studio et dans **Affichage** > **Autres fenêtres**. Nous aurons l'occasion de les manipuler tout au long de ce tutorial. Nous travaillerons notamment avec la **Boîte à outils** de Visual Studio dans le prochain atelier, pour dessiner une application de type Windows.



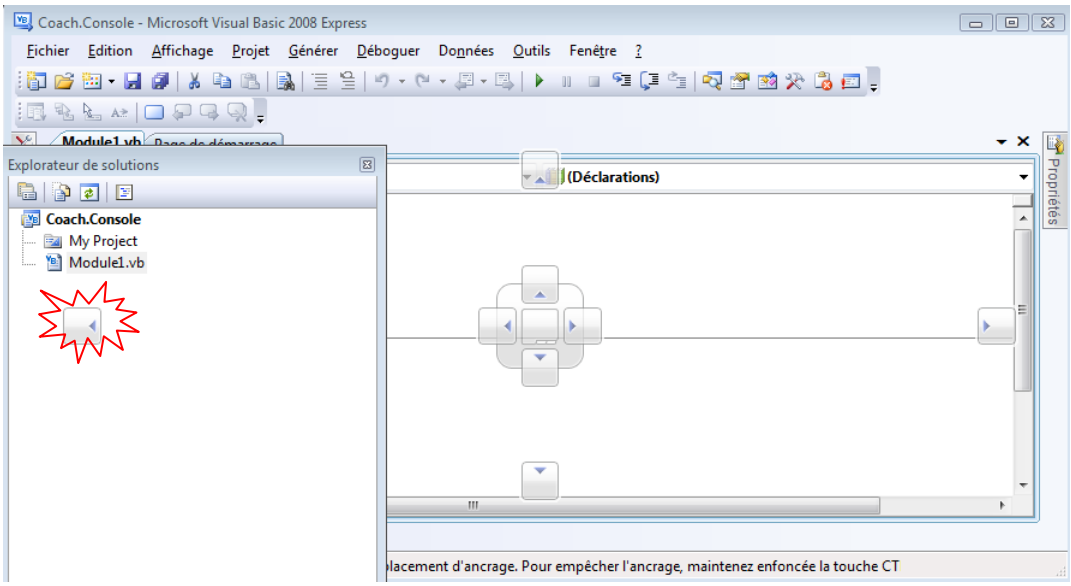
Et pourquoi certaines fenêtres ont leur « poignée » **à droite**, comme l'Explorateur de solutions, d'autres **à gauche**, comme la Boîte à outils, voire même **en bas** comme la Liste d'erreurs ?

En réalité, cela n'a aucune importance et vous pouvez organiser votre bureau comme vous le souhaitez…

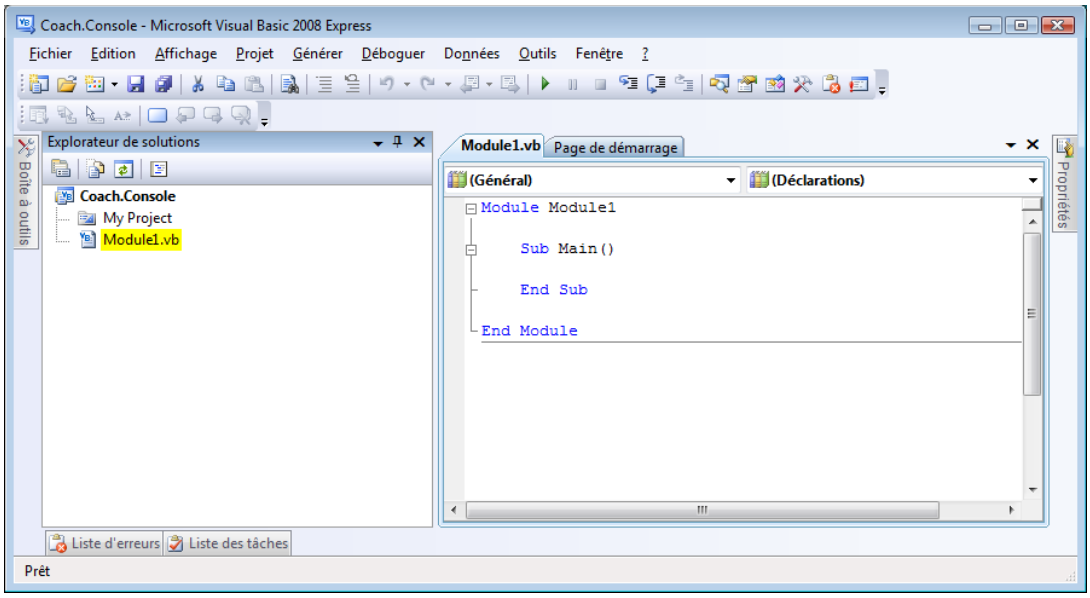
- En effet, faites un clic sur la souris sans relâcher le bouton sur la barre de titre de la fenêtre **Explorateur de solutions** que nous avons fixée précédemment sur la surface de travail.
- Puis faites glisser la souris tout doucement pour détacher la fenêtre complètement de son emplacement initial. Tirez-la par exemple sur la gauche de la surface de travail.

Lorsque vous approchez du centre ou d'un côté de l'écran, Visual Studio vous affiche des petites figurines pour vous aider à positionner la fenêtre à l'endroit souhaité.

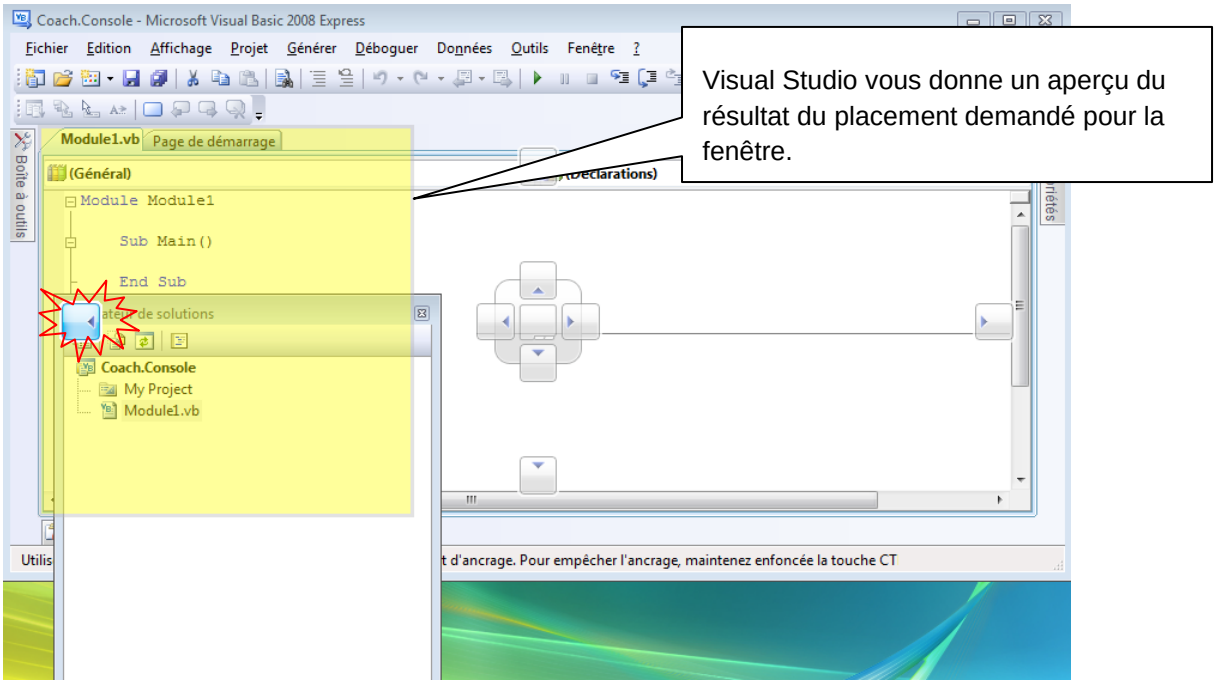
- Par exemple, pour placer la fenêtre sur la gauche de la surface centrale, relâchez la souris juste sur la petite figurine qui apparaît à gauche de l'écran :



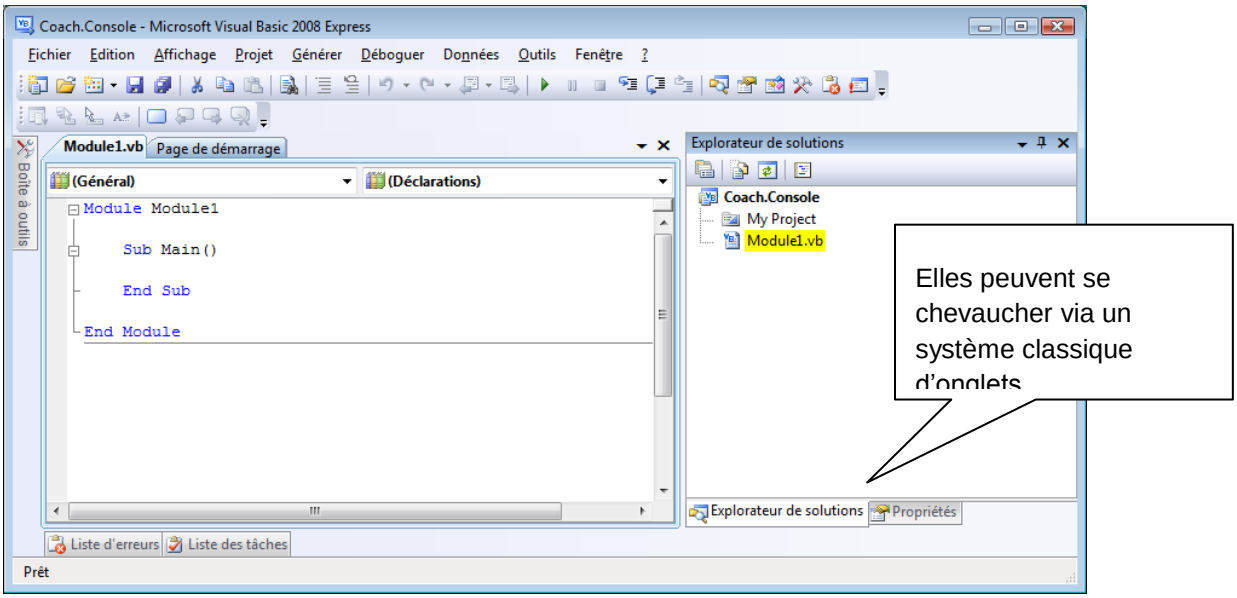
La fenêtre **Explorateur de solutions** se cale à gauche avec toujours les mêmes options de masquage que précédemment via les boutons et :



Si votre fenêtre se positionne sur la gauche mais sans s'intégrer parfaitement avec la surface centrale, c'est que vous n'avez pas relâché la souris précisément sur la figurine de positionnement de Visual Studio. Pour que ça marche à coup sûr, laissez-vous guider par Visual Studio qui vous indique par un effet de surbrillance qu'il a compris que vous vouliez insérer la fenêtre à l'emplacement dicté par la figurine. A ce moment précis, vous pouvez relâcher la souris et votre fenêtre sera correctement positionnée :



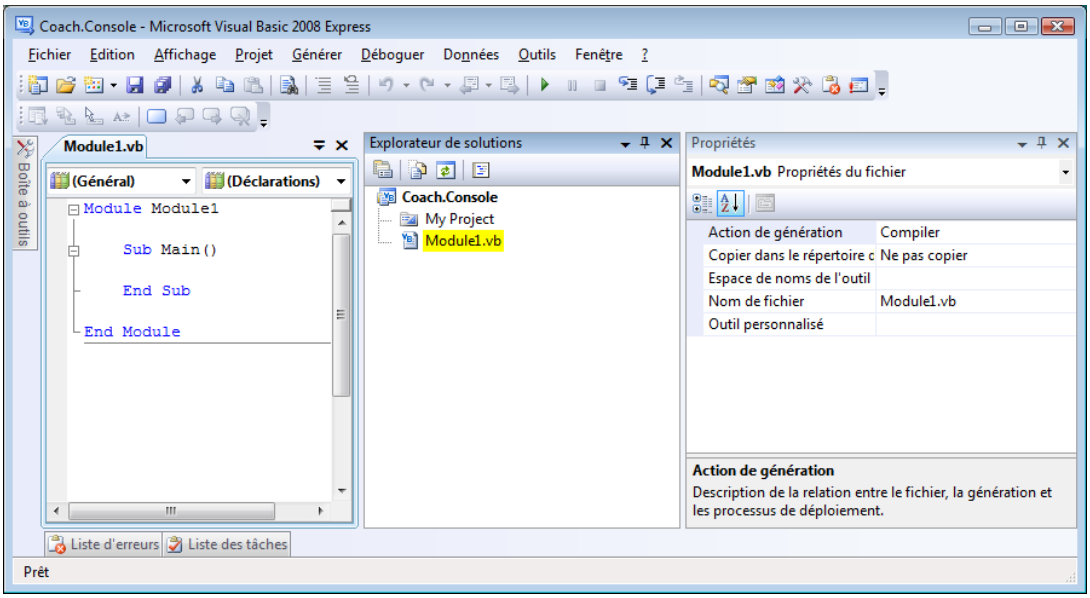
Allez ! Juste pour voir si vous avez compris, réorganisez le côté droit de la surface de travail pour faire apparaître les fenêtres **Explorateur de solutions** et **Propriétés** l'une sur l'autre comme ceci :



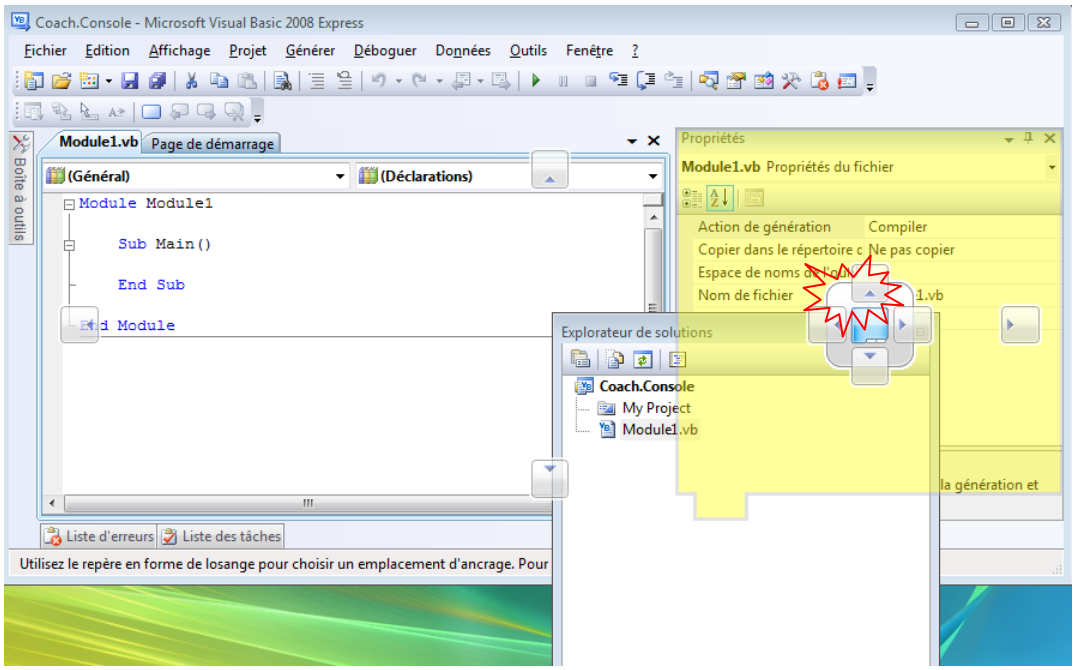
Bon ok, je vous file un tuyau : restez calme ☺…

Je plaisante parce que c'est vraiment super facile !!

- La première chose à comprendre c'est qu'il faut figer les fenêtres avec la avant de les déplacer (sinon elles se masquent automatiquement dès que vous déplacez la souris hors de leur champ…c'est logique !). Elles se retrouvent côte à côte :

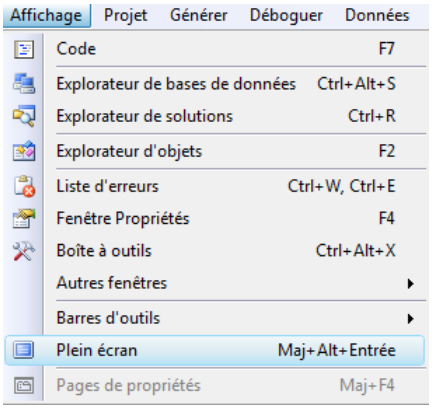


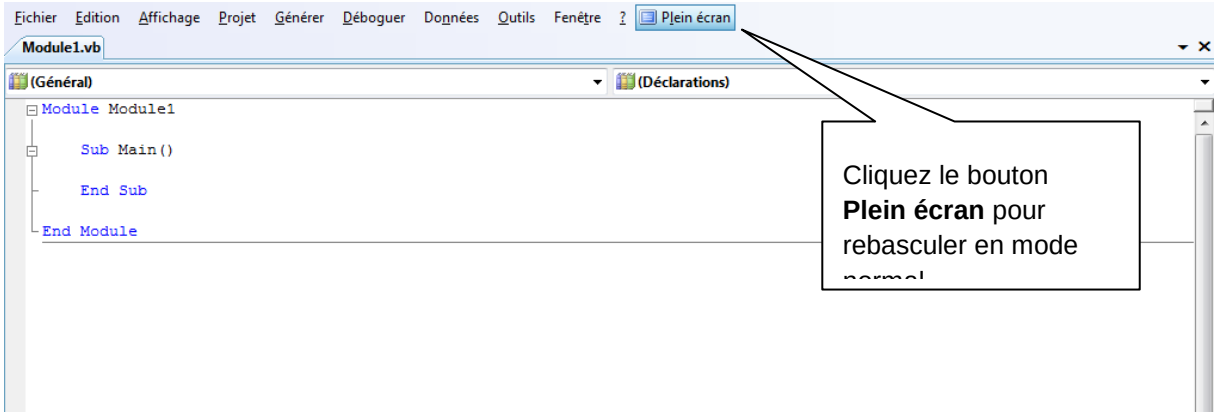
- La seconde étape consiste à positionner les fenêtres l'une sur l'autre en laissant celle la plus à droite en place, puis en déplaçant la seconde en la relâchant sur la petite figurine centrale (qui montre d'ailleurs l'organisation avec onglets). Et le tour est joué !



.

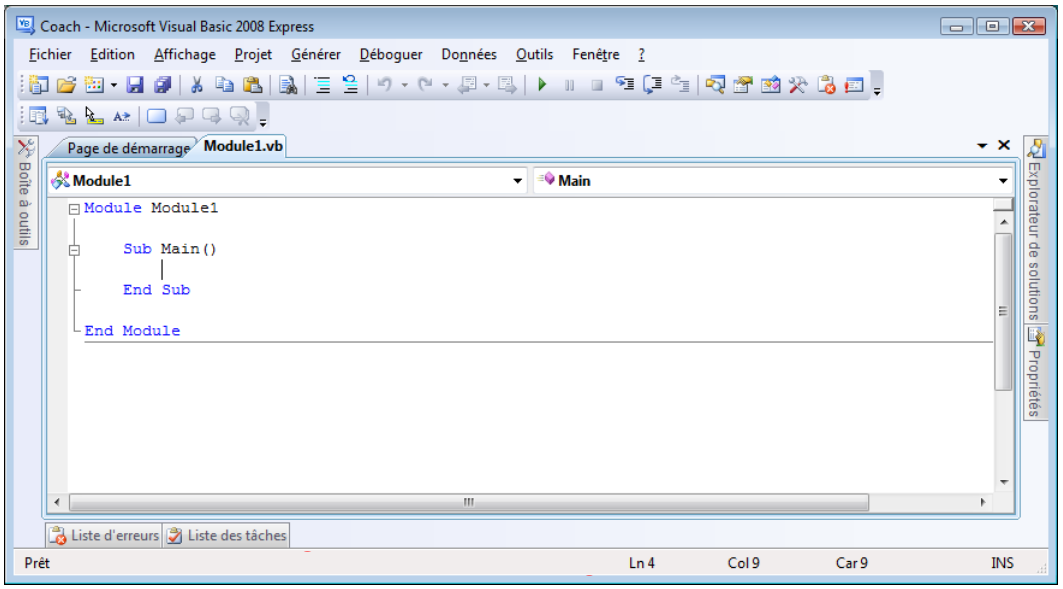
Sachez enfin qu'avec le menu **Affichage** > **Plein écran** ou le raccourci clavier **Maj+Alt+Entrée**, vous pouvez carrément passer votre surface centrale en plein écran :





De toutes les façons, c'est bien connu qu'on n'a jamais assez de place quand il s'agit de développer ☺. Un bon truc qui se répand de plus en plus dans les entreprises est d'apprendre à bosser avec plusieurs écrans si vous pouvez…et là c'est top !

Terminons notre exploration du bureau :



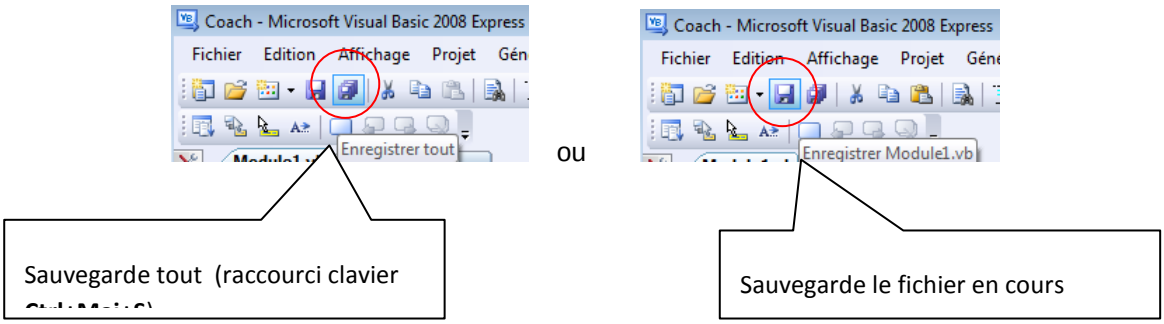
- **8** : Au bas de l'écran, vous trouvez en général des fenêtres en rapport avec la compilation, comme la Liste d'erreurs ou avec les fonctionnalités de débogage que

nous verrons plus loin dans cet atelier.

- **9** : Visual Studio a également sa barre d'état. A gauche, vous y lirez des messages tels que l'état d'une compilation et à droite des informations comme le numéro de ligne (Ln), de colonne (Col) qui caractérisent la position du curseur en cours dans l'éditeur de code (très pratique par exemple pour retrouver une erreur à un numéro de ligne spécifié par le compilateur).

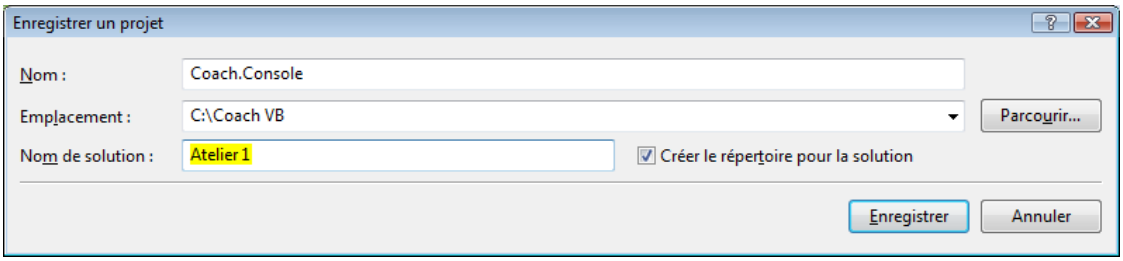
2. Enregistrez le projet dans une solution :

- Dans la barre d'outils de Visual Studio, cliquez l'icône pour sauvegarder le projet.



- Dans la boîte de dialogue **Enregistrer un projet**, indiquez votre répertoire de travail (par défaut nous retrouvons le chemin que nous avons spécifié dans la boîte de dialogue d'options de Visual Studio un peu plus haut dans cet exercice).

- C'est à ce moment là que vous pouvez demander la création d'une solution en cochant la case **Créer le répertoire pour la solution** et en saisissant un nom pour la solution par exemple : **Atelier 1**.



- Cliquez sur **Enregistrer**.

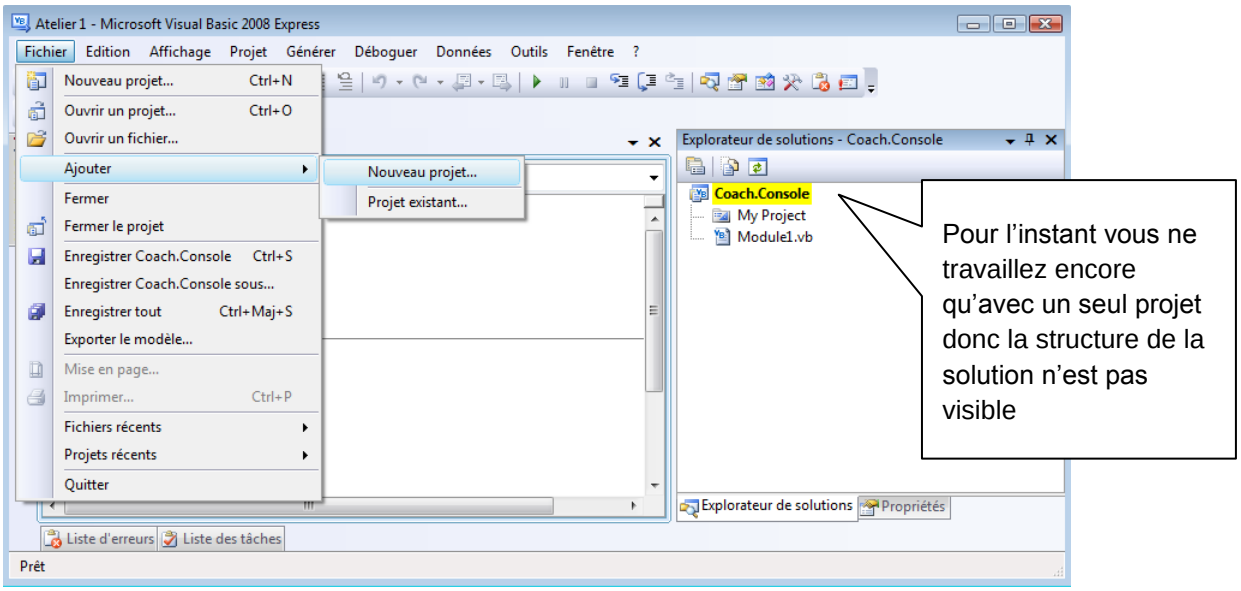
En quoi consiste une **solution** par rapport à la notion de **projet** ?

Dans le cadre d'un développement, il arrive souvent que la compilation d'une application génère plusieurs assemblies .NET. C'était d'ailleurs le cas de notre premier exercice où nous avons abouti sur la génération de deux fichiers distincts, d'extension .exe et .dll. Clairement nous avons codé deux *projets* distincts.

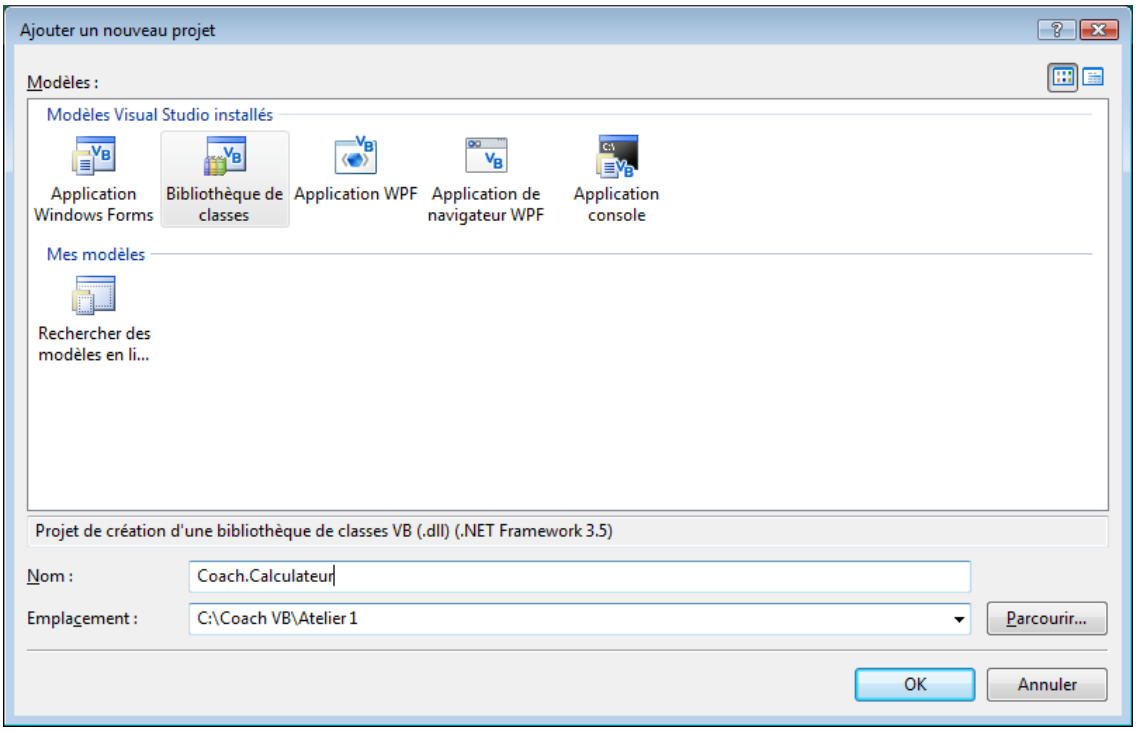
L'intérêt d'une *solution* est de rassembler d'une manière logique plusieurs *projets* répondant à une même problématique de codage de façon à travailler sur les différents projets à partir d'une seule instance de Visual Studio (rien ne vous empêche d'en ouvrir plusieurs mais comme vos projets interagissant entre eux, quelle perte de temps de basculer d'une instance de Visual Studio à une autre !).

L'outil de Visual Studio qui permet de gérer l'organisation de vos projets dans une solution est l'**Explorateur de solutions** (c'est la fenêtre que vous avez manipulée précédemment pour apprendre à positionner les fenêtres sur la surface de travail). Il affiche vos solutions comme containeurs principaux, et les projets qu'elles contiennent sous la forme d'une arborescence.

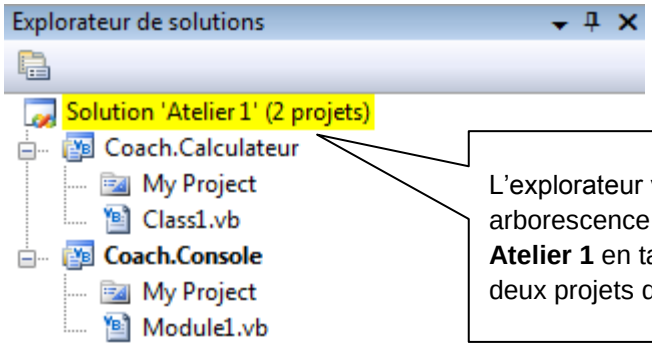
3. Créez un second projet au sein de la même solution pour développer le calculateur :
   - Affichez l'**Explorateur de solutions**. Sélectionnez le nœud racine **Coach.Console**.
   - A partir des menus de Visual Studio, sélectionnez le menu **Ajouter > Nouveau projet**… :



- Dans la boîte de dialogue **Ajouter un nouveau projet**, sélectionnez le modèle de projet **Bibliothèque de classes** et indiquez un nom de projet : **Coach.Calculateur**. Visual Studio doit vous proposer par défaut l'emplacement du répertoire (..\Atelier 1) correspondant à la solution créée précédemment :



- Validez par **OK**.
- Affichez l'**Explorateur de solutions** pour voir la solution de notre **Atelier 1** encadrant les deux projets **Coach.Console** et **Coach.Calculateur** :



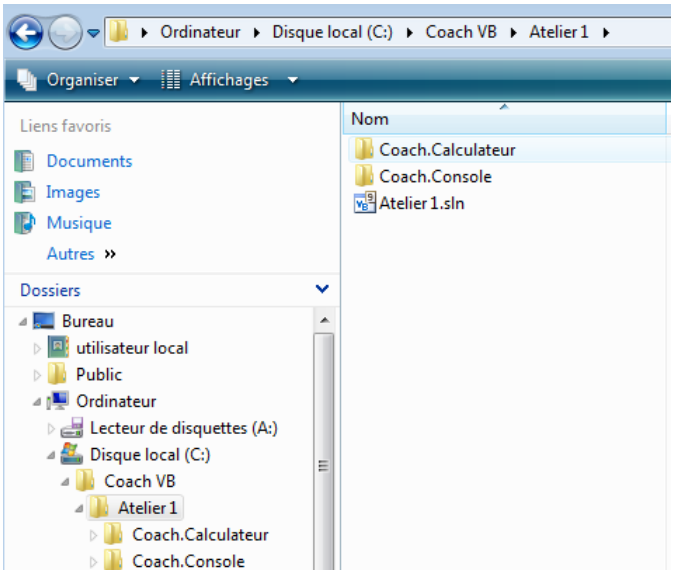
L'explorateur vous présente maintenant une arborescence avec à la racine, la solution **Atelier 1** en tant que containeur logique de vos deux projets de développement.

- Enregistrez l'ensemble de la solution en cliquant dans la barre d'outils de Visual Studio.

4. Regardons ce qui a été créé sur le disque :

   - Avec l'**Explorateur Windows**, regardez ce qu'il y a dans votre répertoire de travail (par exemple **C:\Coach VB**) ;

   - Vous devez voir un répertoire **Atelier 1** pour la solution, contenant lui-même un fichier **Coach.sln**, et deux sous répertoires **Coach.Console** et **Coach.Calculateur** avec respectivement vos deux projets.

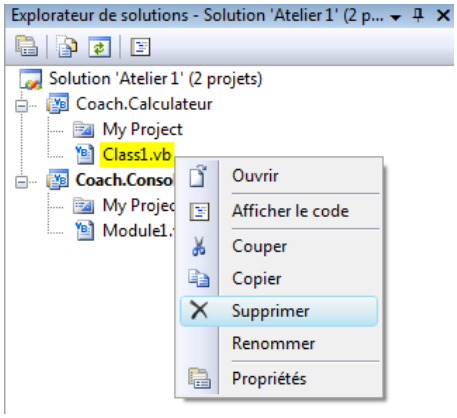


Le fichier d'extension **.sln** est le fichier de solution. Vous pouvez l'ouvrir avec le Bloc-notes. Il contient la définition de tous les projets composants la solution.

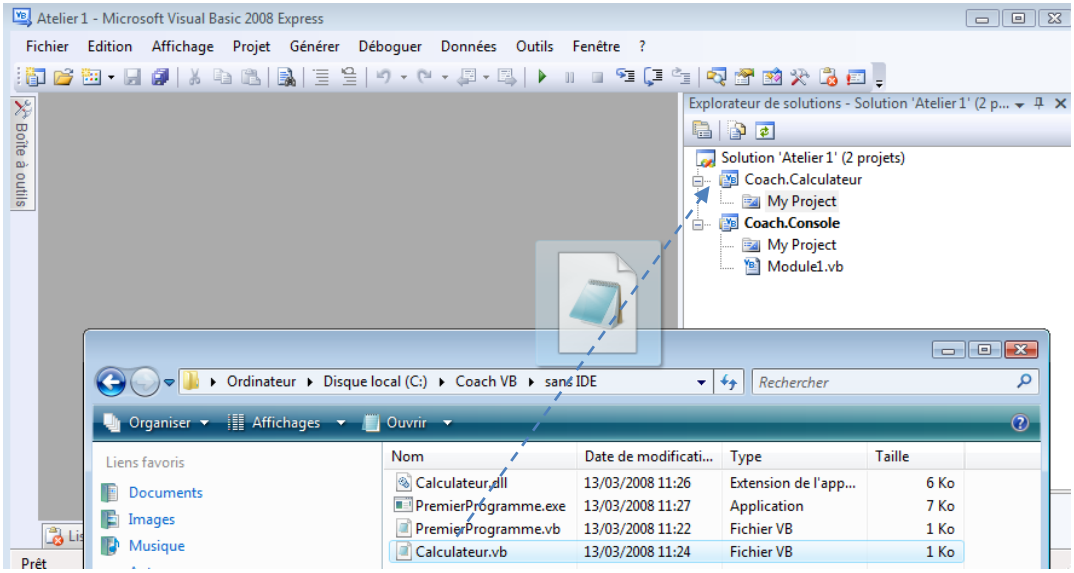
5. Récupérez le code du calculateur que vous avez programmé à l'exercice précédent pour le rapatrier dans le projet :

   - Avec l'**Explorateur Windows**, retrouvez le fichier **Calculateur.vb** de l'exercice précédent.

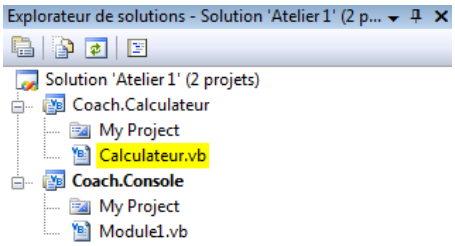
- Gardez l'Explorateur Windows ouvert et basculez sur l'**Explorateur de solutions** de Visual Studio.
- Ouvrez le nœud de projet **Coach.Calculateur**, puis faites un clic-droit sur le fichier **Class1.vb** et sélectionnez le menu **Supprimer** :



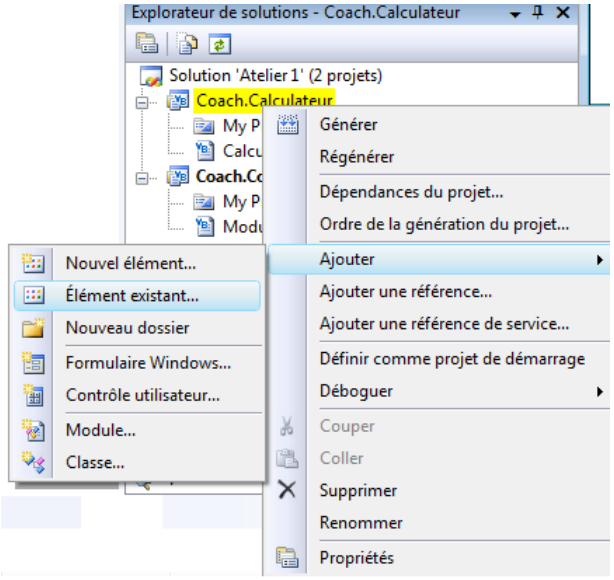
- Cliquez le bouton **OK** de la boîte de dialogue de confirmation de la suppression.
- Faites un glisser déplacer du fichier **Calculateur.vb** depuis l'**Explorateur Windows** sur le nœud de projet **Coach.Calculateur** dans l'**Explorateur de solutions**.



Vous obtenez :

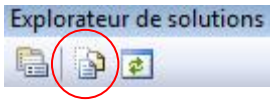


Vous constatez que pour ajouter un élément à votre projet (quelque soit le type de fichier), un simple glisser déplacer suffit ! Vous pouvez aussi utiliser le clic droit sur le nœud du projet dans l'**Explorateur de solutions** > **Ajouter** > **Elément existant**.



En revanche, le même glisser déplacer directement sur l'emplacement du projet dans l'Explorateur Windows n'est pas aussi rapide car le fichier est vu comme étant sous le dossier par Visual Studio mais n'est pas considéré comme étant inclus dans le projet tant que vous ne le spécifiez pas de manière explicite. D'ailleurs avec cette méthode, le fichier n'apparaît pas dans l'Explorateur de solutions au premier abord. Pour l'afficher, vous devez cliquer sur l'icône **Afficher tous les fichiers** de la barre d'outils de

l'Explorateur de solutions.



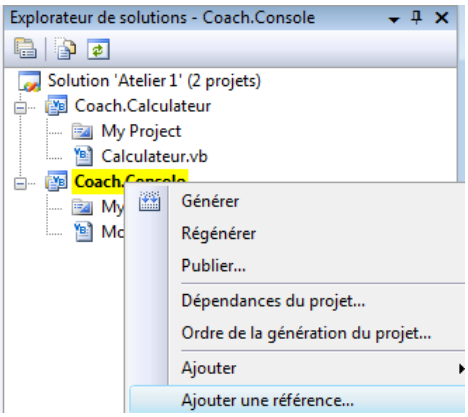
- Dans la barre d'outils de Visual Studio, cliquez le bouton pour sauvegarder toutes les modifications réalisées.

6. Modifiez le projet **Coach.Console** pour utiliser la librairie de calcul :

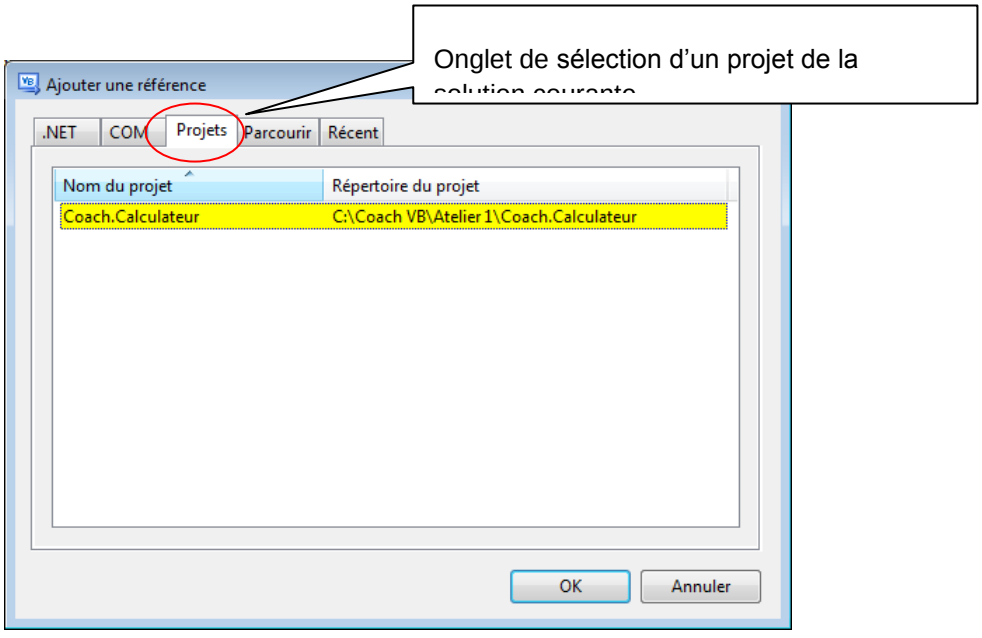
Pour nous aider efficacement dans le développement de notre solution, il manque une précision à Visual Studio qui ne sait toujours pas de quelle manière nos deux projets sont liés. Par exemple pour compiler l'ensemble ou pour nous aider à détecter d'éventuelles erreurs au moment du codage, il faut que Visual Studio comprenne que le programme *Coach.Console* a besoin d'une **référence** sur la bibliothèque *Coach.Calculateur* (exactement comme nous l'avons fait au moment de la compilation du projet dans le premier exercice de l'atelier).

Soit dit en passant, votre solution peut très bien inclure des projets sans liens particuliers ! Mais à priori, l'intérêt d'une solution réside justement dans le fait de rassembler dans un même environnement des projets qui répondent à une problématique de développement commune donc…

- Indiquez que le projet **Coach.Console** va utiliser le calculateur de la librairie **Coach.Calculateur**, en faisant un **clic-droit** sur le projet **Coach.Console** > **Ajouter une référence…** :



- Dans la boîte de dialogue **Ajouter une référence**, cliquez sur l'onglet **Projets**, et sélectionnez l'unique autre projet de la solution **Coach.Calculateur** :



- Cliquez le bouton **OK**.

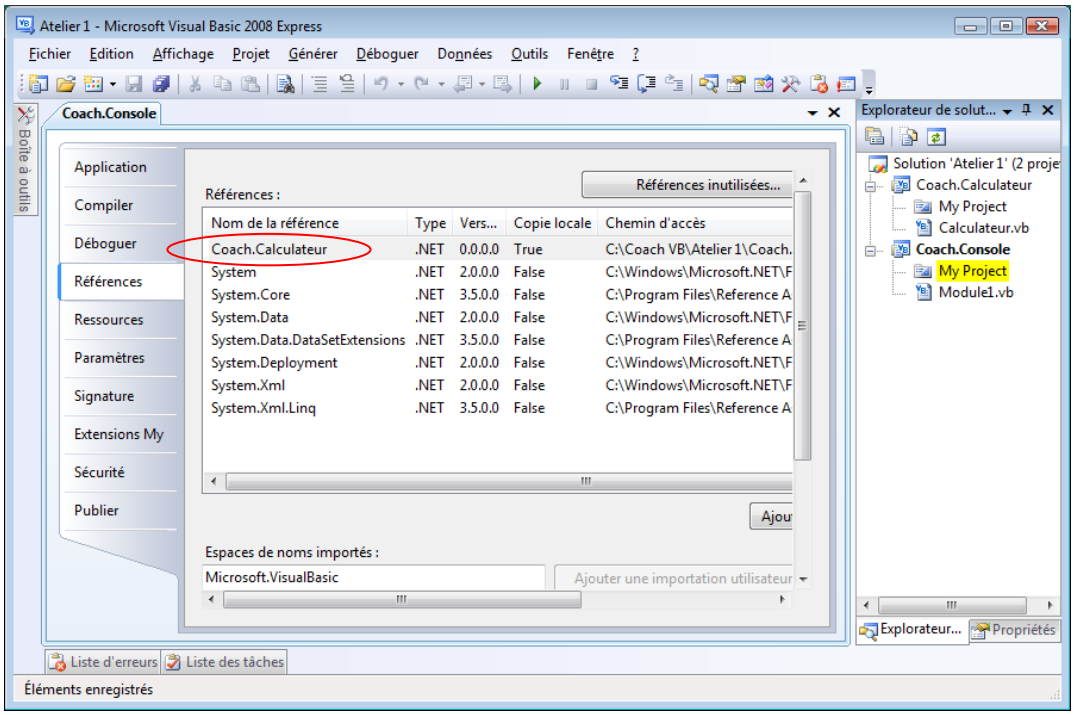
La boîte de dialogue **Ajouter une référence** propose plusieurs onglets qui suggèrent différentes méthodes pour retrouver la référence concernée, selon que vous voulez pointer sur une bibliothèque connue du .NET Framework (onglet **.NET**) ou sur une bibliothèque de composants COM (onglet **COM**) c'est-à-dire non managée, ou encore pour parcourir le disque à la recherche d'une .dll que vous avez par exemple récupérée d'un tiers (onglet **Parcourir**).

L'onglet **Projets** est un raccourci pour scruter directement la solution de développement en cours. Pratique ! En fait, cela reviendrait à pointer le fichier d'extension .dll sur le répertoire de projet du disque avec l'onglet **Parcourir**. Sauf que notre dll n'existe pas encore puisque nous n'avons pas encore compilé le projet. Mais ça ne gêne pas Visual Studio qui s'occupe de tout au travers de notre solution… ☺

Où peut-on voir la référence créée entre les deux projets ?

Visual Studio regroupe toutes les propriétés des projets VB dans une fenêtre accessible via le nœud **My Project** du projet correspondant dans l'**Explorateur de solutions**.

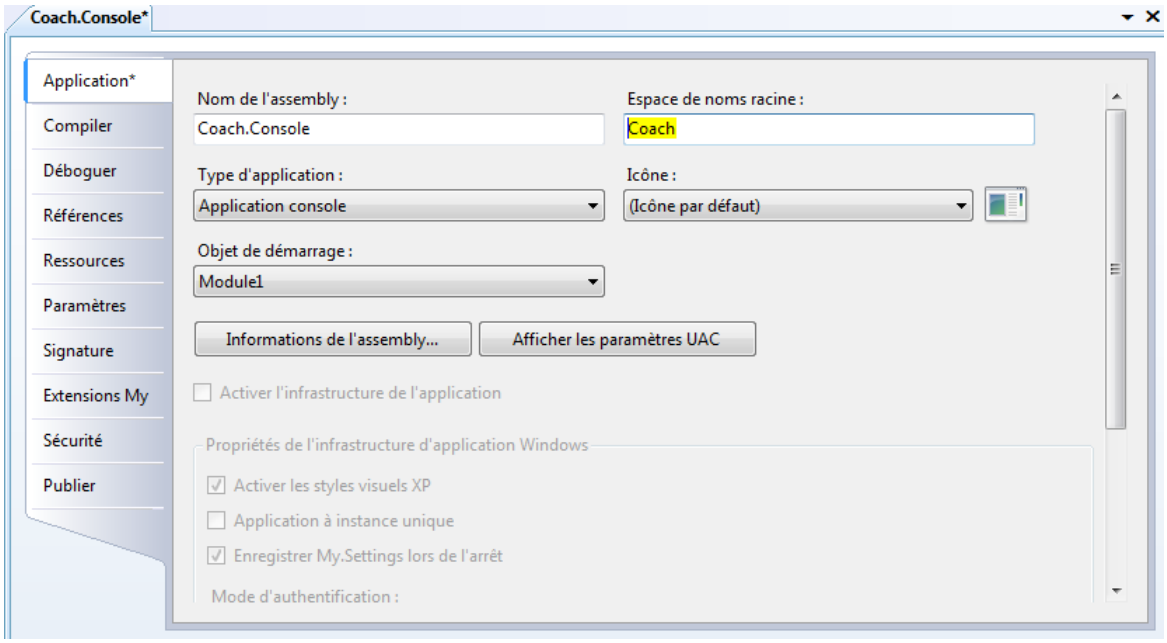
- Double cliquez le nœud **My Project** du projet **Coach.Console** dans l'**Explorateur de solutions**.
- Cliquez l'onglet **Références** sur le côté gauche de la fenêtre pour voir la liste des références du projet et retrouver ainsi la référence au projet **Coach.Calculateur** :



Vous constatez que ce n'est pas la seule référence du projet ! En effet, puisque nous avons créé le projet sur la base d'un modèle de projet de Visual Studio, des références vers tous les espaces de noms usuels du .NET Framework sont préenregistrées. Ainsi vous ne perdez pas de temps à référencer ces bibliothèques de classes qui vont vous servir quasiment à coup sûr ☺.

Vous pouvez bien sûr alléger cette liste en sélectionnant la ou les références inutiles et en cliquant sur le bouton **Supprimer**.

- Toujours dans cette même fenêtre, cliquez l'onglet **Application** pour consulter les caractéristiques générales du projet.
- Retouchez par exemple le nom de l'espace de noms racine proposé par défaut par Visual Studio et remplacez le par **Coach** :

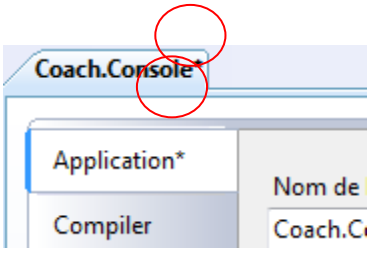


A quoi sert un espace de noms ?

Durant ce tutorial, vous allez créer de nombreux noms d'objets ou de types. Pour éviter les conflits avec des noms déjà existants, **l'espace de noms** précise à quoi se rattache le nom. C'est un peu le nom de famille des objets que vous allez créer.

Nous allons utiliser le même nom pour classer les objets du projet Calculateur, si bien que la classe **Calculateur** aura elle aussi comme nom de famille **Coach**. Son nom complet qui permettra de l'identifier comme appartenant à cette famille, sera donc **Coach.Calculateur**.

Remarquez la petite étoile ***** en haut à droite de l'onglet **Coach.Console** ainsi que dans l'onglet **Application** :

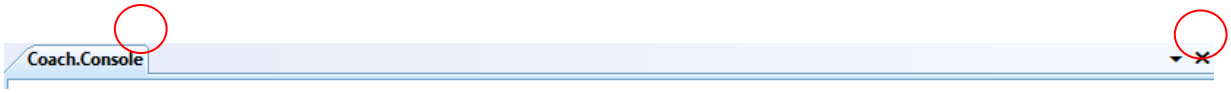


L'étoile sur l'onglet principal **Coach.Console** indique que la fenêtre regroupant les propriétés du projet a subi des modifications qui n'ont pas encore été sauvegardées dans la solution. L'étoile sur l'onglet **Application** précise que c'est dans cet onglet que des modifications ont été faites.

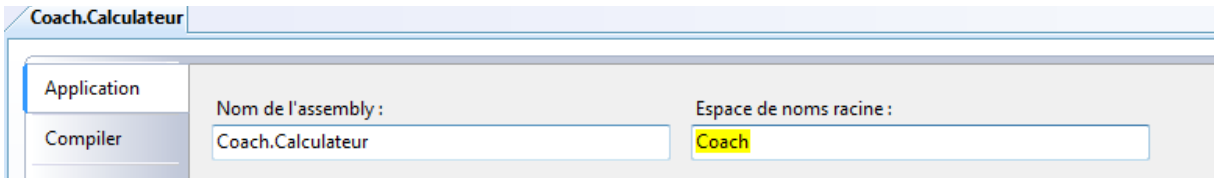
Visual Studio adopte ce principe pour tout fichier ouvert sur la surface de travail de sortes que vous pouvez toujours repérer d'un bref coup d'œil sur la zone des onglets quels sont les fichiers sur lesquels vous avez fait vos dernières modifications et qu'il faut donc enregistrer.

- Dans la barre d'outils de Visual Studio, cliquez le bouton pour sauvegarder toutes les modifications réalisées.

L'étoile disparaît dans les onglets. Pour fermer la fenêtre de propriétés du projet, cliquez sur l'icône à droite de la fenêtre :



- Dans l'**Explorateur de solutions**, double cliquez maintenant sur le nœud **My Project** du projet **Coach.Calculateur**, de façon à indiquer le même nom pour l'espace de noms :



- Enregistrez vos modifications puis fermez la fenêtre de propriétés.

Bravo ! Vous êtes maintenant prêt à coder l'application !

EDITER LE CODE

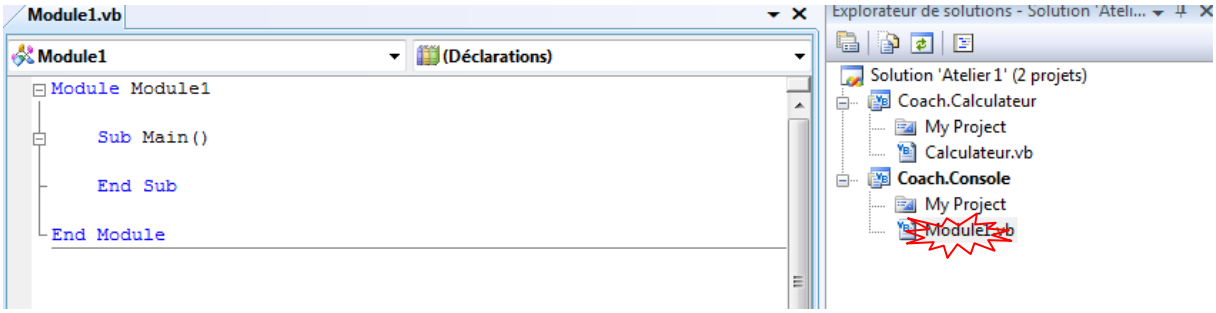
Développer avec le Bloc-notes vos lignes de code, c'est un peu comme skier sur deux planches de bois brut ! Avec Visual Studio, vous codez comme si vous skiiez avec des skis derniers cris et par-dessus tout, des skis *intelligents* ☺ !

A la fin de cet exercice, vous saurez :

- Utiliser l'IntelliSense,
- Utiliser la liste de tâches,
- Utiliser les extraits de code,
- Naviguer dans le code avec les signets et les lignes,
- Structurer votre code dans des régions,
- Exploiter les codes couleur de l'éditeur.

Déroulement de l'exercice :

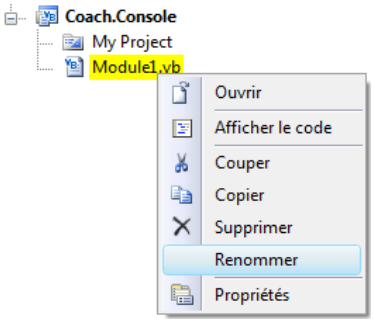
1. Partons à la découverte des différentes aides que proposent Visual Studio pour vous épauler dans l'écriture du code :
    - Editez le fichier de code **Module1.vb** du projet **Coach.Console** généré par le modèle de projet Application Console, en double cliquant sur le fichier dans l'**Explorateur de solutions**.



Le code aurait aussi pu être affiché :

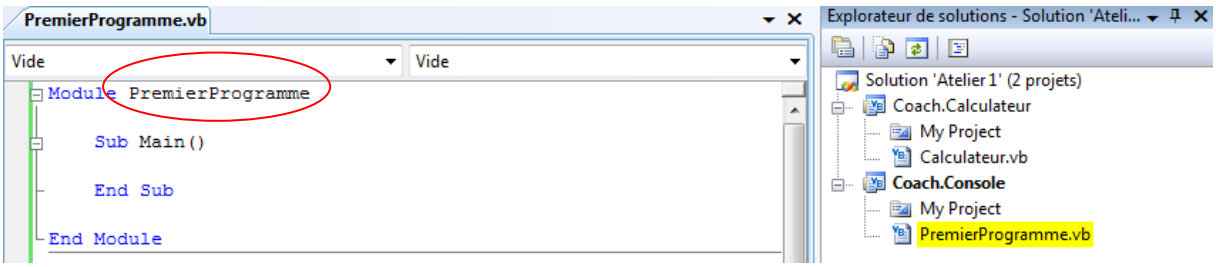
- en cliquant le bouton (**Afficher le code**) de la barre d'outils de l'Explorateur de Solutions,
- ou en sélectionnant le fichier **Module1.vb** et en cliquant sur le menu **Affichage** > **Code** de Visual Studio,
- ou encore en sélectionnant le fichier **Module1.vb** et en tapant la touche **F7**.

- Renommez le fichier depuis l'**Explorateur de solutions** en faisant un clic droit sur le fichier > **Renommer**. Nommez-le comme dans le premier exercice : **PremierProgramme.vb**.



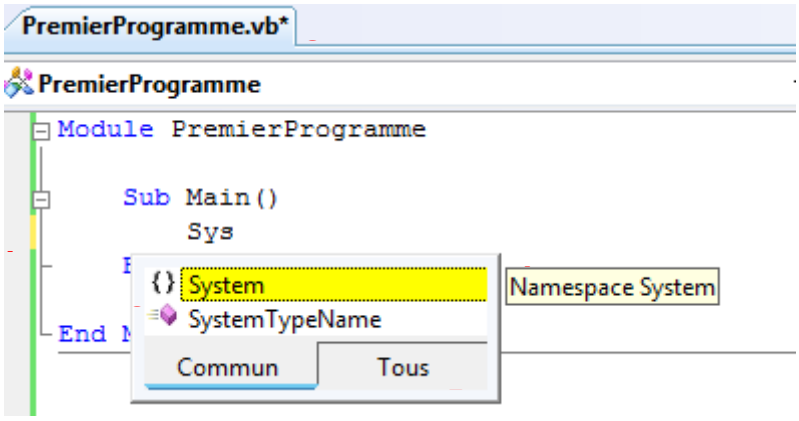
Pour aller plus vite, Visual Studio renomme du même coup le nom du module dans le fichier de code. Puisqu'initialement ils ont le même nom, Visual Studio devine que vous voulez renommer les deux !

En revanche attention, l'inverse n'est pas vrai ! Si vous renommez une classe par exemple, le nom du fichier qui la contient reste inchangé. C'est logique car bien souvent un fichier n'est pas limité à une seule classe.



- Tapons la première ligne de code !

  A l'intérieur de la fonction **Main**, commencez à taper **Sys**, c'est-à-dire les trois premières lettres de l'espace de noms **System** du .NET Framework, dans lequel nous allons utiliser la méthode **WriteLine** de la classe **Console** :



Que dit l'éditeur de Visual Studio (oui, parfaitement il vous cause)?

Plein de choses… :

- **1** : d'abord que vous êtes en train de modifier la ligne. Donc pour bien mettre en évidence que des modifications sont faites et encore non enregistrées sur cette ligne, il marque la ligne en jaune dans la marge. A la prochaine

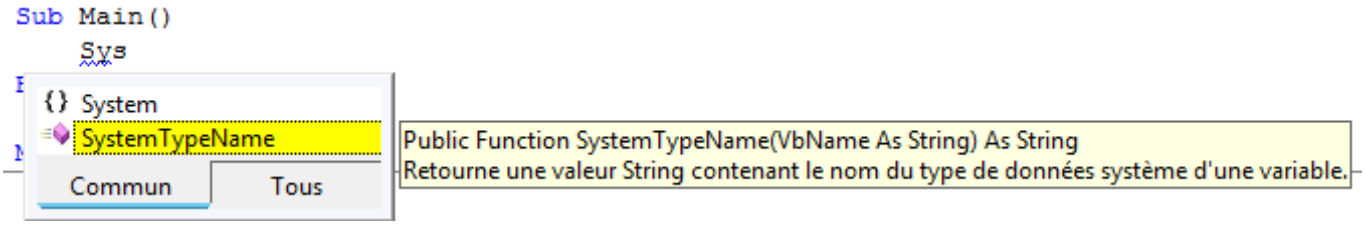
sauvegarde, la ligne repasse en vert dans la marge.

- **2** : ensuite que votre fichier subit des modifications en marquant l'onglet du fichier d'une petite étoile *.
- **3** : la petite fenêtre qui apparaît juste en dessous de la ligne de saisie vient d'une fonctionnalité essentielle de l'éditeur appelée **IntelliSense**. Je vous l'avais dit, l'éditeur est ***intelli***gent et ***sent*** les choses pour vous aider à développer vite du code juste ☺. A l'usage, vous verrez que vous vous demanderez comment vous pouviez développer jusqu'à maintenant sans elle !

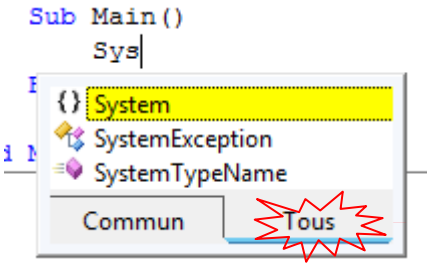
  Ici elle vous dit qu'elle suppose que vous voulez invoquer l'espace de noms (marqué de l'icône {}) nommé **System**, encore qu'il existe d'autres éléments qui commencent par **Sys**, comme **SystemTypeName** qui lui, est une méthode (reconnaissable par l'icône).

Pour avoir la signification de tous les icônes de la liste des membres : http://msdn2.microsoft.com/fr-fr/library/y47ychfe.aspx

- Cliquez sur **SystemTypeName** avec la souris. Une info bulle (**4**) apparaît pour vous donner les caractéristiques principales de l'élément en question.



Mais ces éléments ne sont que les plus *communs*. Pour voir *tous* ceux que l'IntelliSense détecte basé sur les trois lettres que vous avez saisies, cliquez sur l'onglet **Tous** (**5**) au bas de la fenêtre.



Sur quoi se base Visual Studio pour proposer ces éléments ?

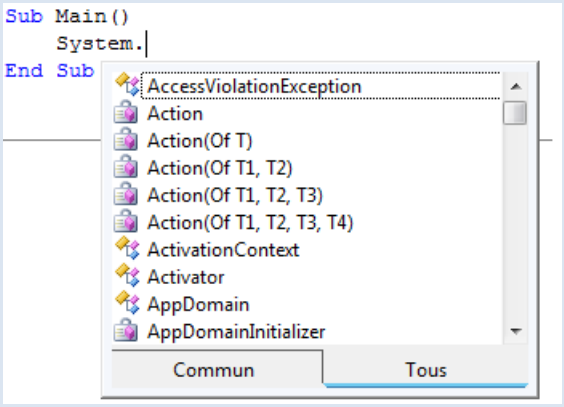
L'IntelliSense a pour objet de vous faciliter l'accès aux guides de référence du langage. Elle s'appuie également sur l'ensemble des références qui sont enregistrées sur le projet. C'est ainsi qu'elle va pouvoir nous aider à utiliser les fonctions de notre librairie de calcul **Coach.Calculateur**, pour laquelle nous avons défini une référence dans le projet Console.

Pour en savoir plus sur l'IntelliSense :

http://msdn2.microsoft.com/fr-fr/library/hcw1s69b.aspx

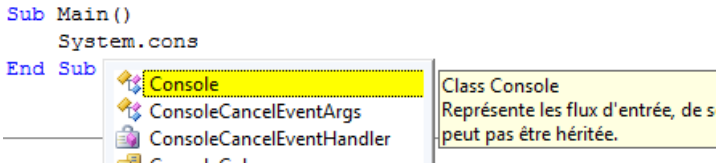
Pour désactiver certaines options de l'IntelliSense qui pourraient vous importuner : http://msdn2.microsoft.com/fr-fr/library/ecfczya1.aspx

- Tapez tout de suite après les trois lettres **Sys** un point.



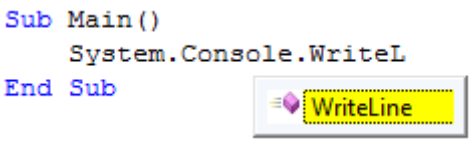
Le point indique à l'IntelliSense que vous acceptez sa première suggestion (le mot System) et que vous voulez poursuivre avec un membre de l'élément, dans notre cas, un sous élément de l'espace de noms System. Elle vous propose donc maintenant une **liste filtrée de tous les membres** de l'espace de noms System.

- Continuez en tapant **cons** pour rechercher la classe **Console** :

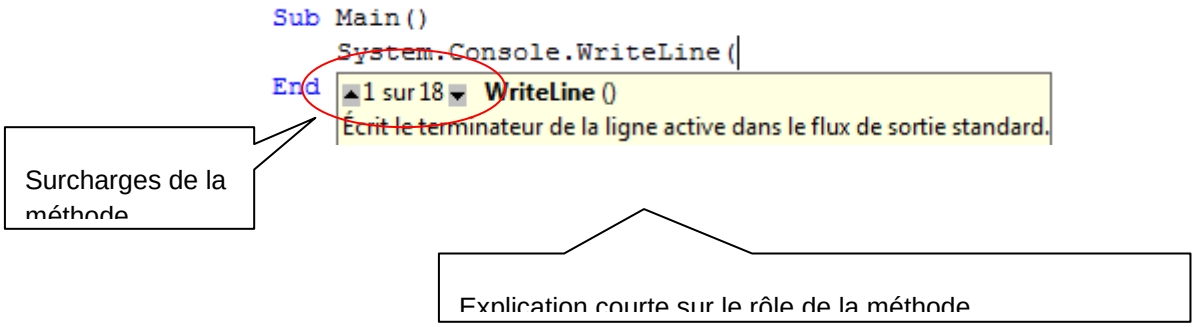


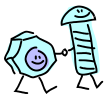
Notez que l'éditeur de code VB se moque complètement que vous respectiez la casse des noms ou pas. Même si vous saisissez **cons** en minuscules, il s'occupera de faire la correction qui s'impose à la suite de votre saisie. Plus cool que ça, ce ne serait pas supportable ☺.

- Tapez à nouveau un point puis le début du nom de la méthode jusqu'à n'avoir plus qu'un seul choix dans la liste.



- Validez en tapant la touche **TAB** (tabulation). L'IntelliSense termine automatiquement pour vous la saisie du mot **WriteLine**.
- Tapez ensuite une parenthèse pour commencer la saisie des paramètres de la méthode.



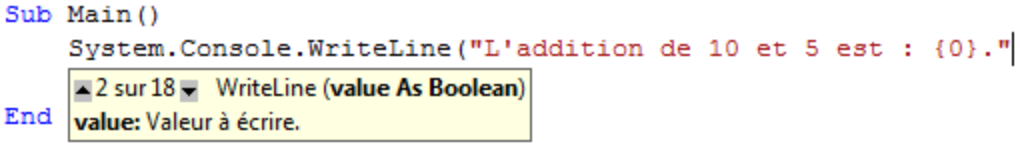


L'IntelliSense intervient à nouveau pour vous aider.

Il faut savoir qu'en programmation objet, une méthode d'une classe peut être **surchargée**, c'est-à-dire qu'elle présente différente signature incluant des paramètres différents en nombre et genre pour palier à tous les contextes d'appel dans lesquels vous pourriez vous trouvez. Ici, il n'existe pas moins de 18 combinaisons de paramètres pour cette méthode ! Imaginez la richesse du .NET Framework !

Aidez vous de l'IntelliSense pour choisir la signature qui vous convient en cliquant sur le bouton ▼ ou si vous êtes sûr de vous, lancez-vous dans la saisie des paramètres. L'IntelliSense s'adapte à chaque frappe de caractères.

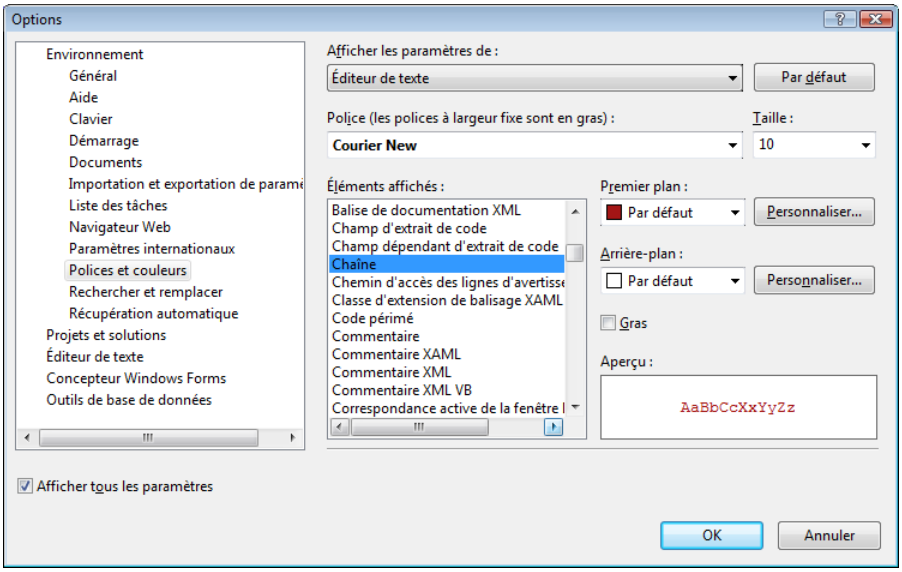
- Saisissez entre double côtes la chaîne de caractères à afficher dans la console : **« L'addition de 10 et 5 est : {0}. »**, contenant le paramètre de résultat marqué par **{0}**.



Dès qu'elle est détectée, la chaîne de caractères est automatiquement affichée en rouge. Visual Studio utilise des codes couleur pour vous donner un maximum de clarté et de visibilité sur le code saisi. Ainsi les commentaires sont en vert, les mots clés du

langage en bleu, etc…

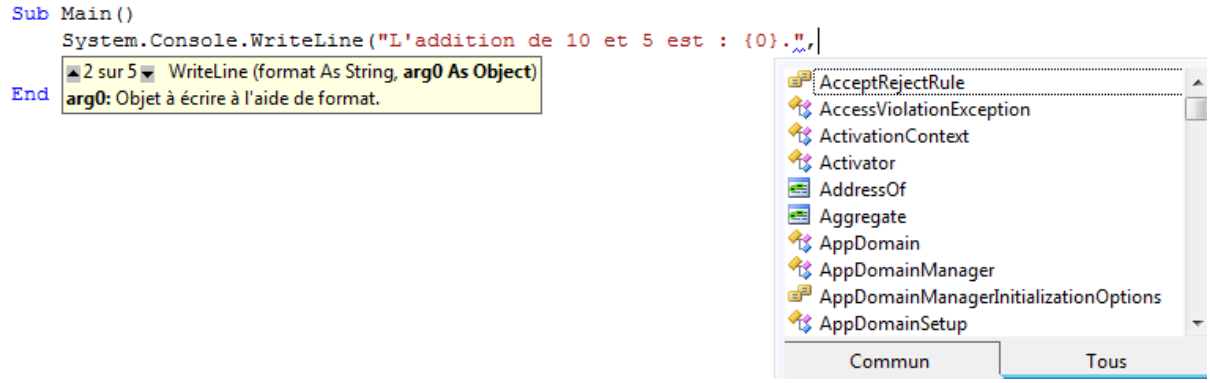
Pour consulter la signification des codes couleur, rendez-vous sur la boîte de dialogue **Options** du menu **Outils**, puis sélectionnez **Environnement** > **Polices et couleurs** :



Il faut savoir que vous pouvez complètement personnaliser l'éditeur de code à votre goût si vous le souhaitez. Que ce soit les codes couleur, la gestion du retrait des lignes, les numéros de ligne etc…, voici un lien pour creuser la question :

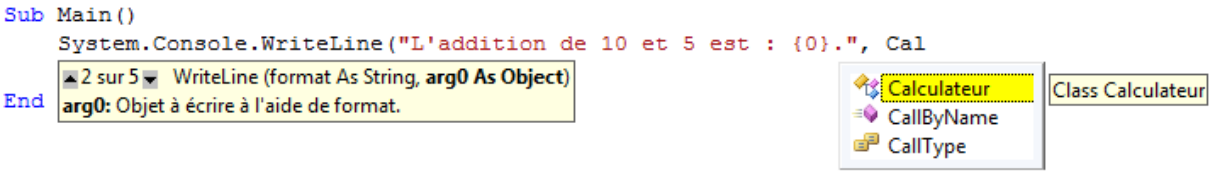
http://msdn2.microsoft.com/fr-fr/library/12199yxc.aspx

- Continuez la saisie des paramètres de la procédure en poursuivant avec une virgule de façon à entrer le second paramètre :



L'IntelliSense vous propose la signature de méthode définitive adaptée au type de paramètre saisi : le premier paramètre est une chaîne formatée à afficher sur la console (**format as String**) et le second est le paramètre dynamique à inclure du fait de l'utilisation des formats de chaîne (**arg0**). Il peut être de n'importe quel type (**as Object**). C'est maintenant que nous allons faire appel à notre classe externe **Calculateur**. Voyons si l'IntelliSense peut nous aider sur nos propres objets.

- Commencez à saisir les premières lettres du nom de la classe : **Calculateur**.



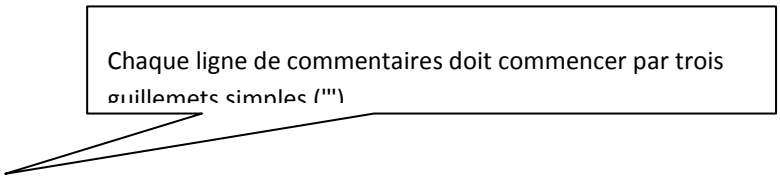
L'IntelliSense réagit puisqu'elle connait notre classe par le biais de la référence au projet **Coach.Calculateur** que nous avons fait précédemment. En revanche, vous constatez que l'aide express est bien mince…

Heureusement, il existe une fonctionnalité de documentation dans Visual Studio basée sur une grammaire XML, qui peut être utilisée en combinaison avec l'IntelliSense pour commenter efficacement vos projets.

Pour en savoir plus sur la manière d'exploiter l'IntelliSense dans vos projets à l'aide de la fonctionnalité **Documentation XML** de Visual Studio : http://msdn2.microsoft.com/fr-fr/library/s0we08bk.aspx

Pour avoir une liste des balises de commentaires recommandées pour VB : http://msdn2.microsoft.com/fr-fr/library/ms172653.aspx

- A partir de l'**Explorateur de solutions**, éditez le fichier de code **Calculateur.vb** du projet **Coach.Calculateur**.
- Ajoutez les lignes de commentaires suivantes qui utilisent les balises recommandées par le guide du langage VB :



Code VB

```
''' <summary>
''' Cette classe implémente des fonctions de calcul avancées
''' </summary>
Public Class Calculateur


    ''' <summary>
    ''' Cette fonction ajoute deux valeurs entières
```

```
    ''' </summary>
    ''' <param name="valeur1">Première valeur</param>
    ''' <param name="valeur2">Deuxième valeur</param>
    ''' <returns>l'entier de résultat</returns>
    Shared Function Ajouter(ByVal valeur1 As Integer, _
                            ByVal valeur2 As Integer) As Integer
        Return valeur1 + valeur2
    End Function

End Class
```

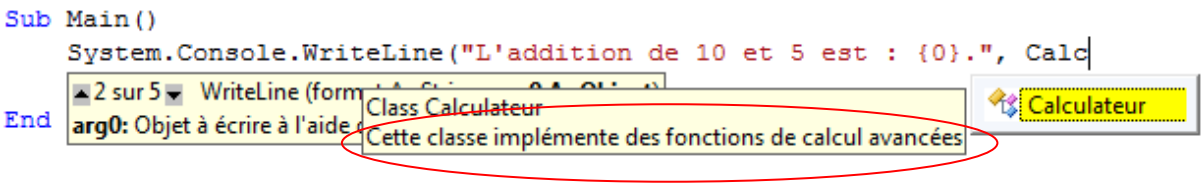
Notez que l'IntelliSense vous aide en vérifiant par exemple le nom des paramètres que vous documentez. Si vous saisissez par mégarde **valeur3**, Visual Studio vous informe qu'il ne voit aucun paramètre de ce nom dans la procédure correspondante !

Passez avec la souris sur le commentaire surligné en vert pour voir le message de Visual

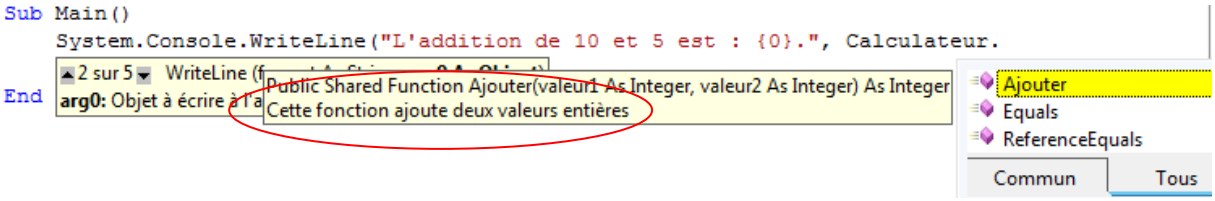
```
''' <summary>
''' Cette fonction ajoute deux valeurs entières
''' </summary>
''' <param name="valeur1">Première valeur</param>
''' <param name="Valeur3">Deuxième valeur</param>
'''
Shar                        ByVal valeur2 As Integer) As Integer
    Return valeur1 + valeur2
End Function
```

Le paramètre de commentaire XML 'Valeur3' ne correspond pas à un paramètre pour l'instruction 'function' correspondante. Le commentaire XML sera ignoré.

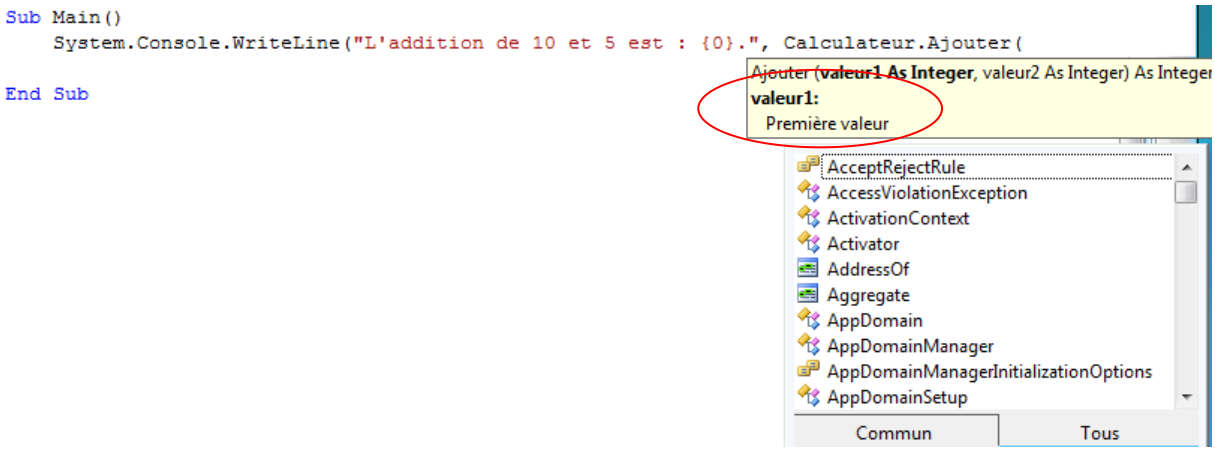
- Enregistrez les modifications du fichier **Calculateur.vb** en cliquant .
- Rebasculez dans le fichier de code **PremierProgramme.vb**.
- Ressaisissez les premières lettres de la classe **Calculateur** à la suite de la ligne, de façon à voir s'afficher l'aide personnalisée à partir des commentaires que vous avez saisis.



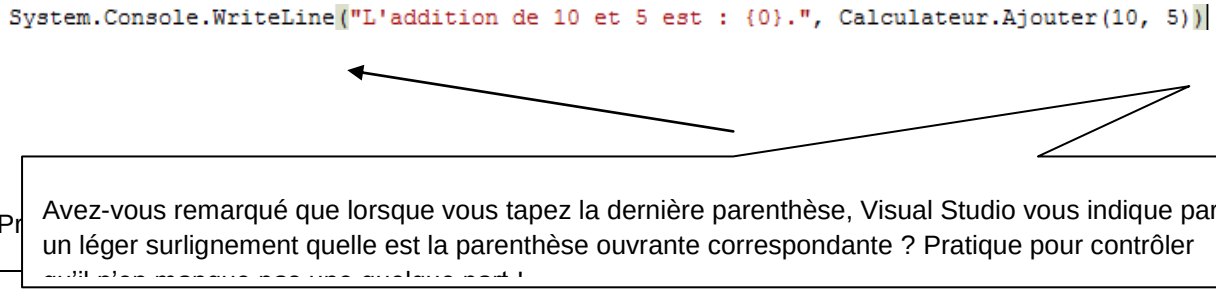
- Tapez directement un point puis commencez à taper le nom de la fonction **Ajouter** pour voir le commentaire de l'IntelliSense :



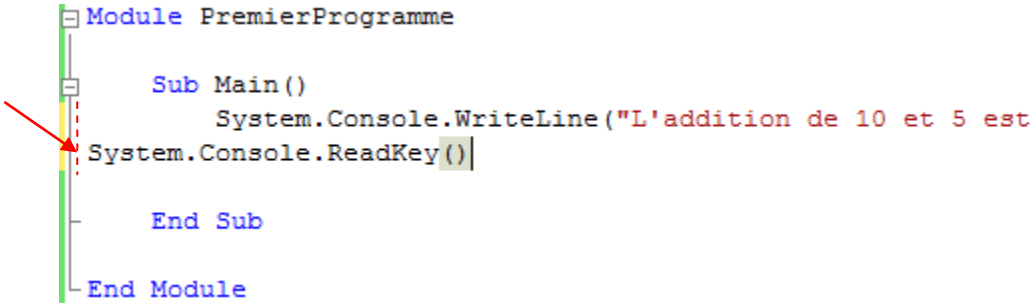
- Tapez la touche **TAB** puis une parenthèse ouverte pour la saisie des paramètres.



- Terminez sur ce principe la saisie complète de la ligne. Validez par **Entrée** :

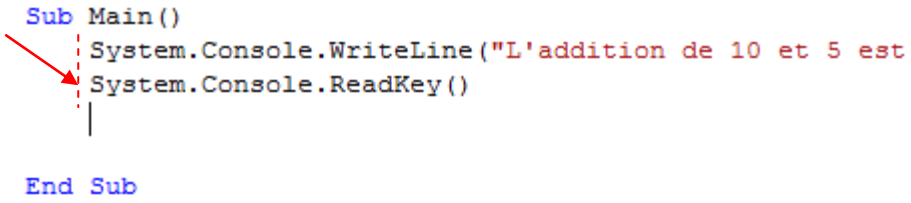


- Saisissez maintenant la seconde ligne de code qui fait appel à la méthode **ReadKey**() de la classe **Console**. Pour coder cette ligne, démarrez complètement sur le bord gauche de la fenêtre de code, comme suit :



```
Module PremierProgramme

    Sub Main()
        System.Console.WriteLine("L'addition de 10 et 5 est
System.Console.ReadKey()

    End Sub

End Module
```

- Validez par la touche **Entrée**. La ligne se cale automatiquement en retrait, directement alignée sur la position de la ligne précédente dans le même bloc de code.



```
Sub Main()
    System.Console.WriteLine("L'addition de 10 et 5 est
    System.Console.ReadKey()

End Sub
```

Visual Studio détermine automatiquement le style de mise en forme approprié pour la ligne, grâce à sa fonctionnalité de **mise en retrait intelligente**.

Pour en savoir plus les fonctionnalités de mise en forme de Visual Studio : http://msdn.microsoft.com/library/fre/default.asp?url=/library/fre/vsintro7

/html/vxtskviewingcodecomments.asp

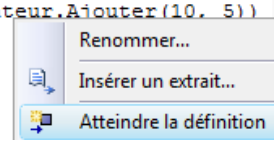
- Enregistrez vos modifications.

2. Naviguez entre les différents projets de la solution.

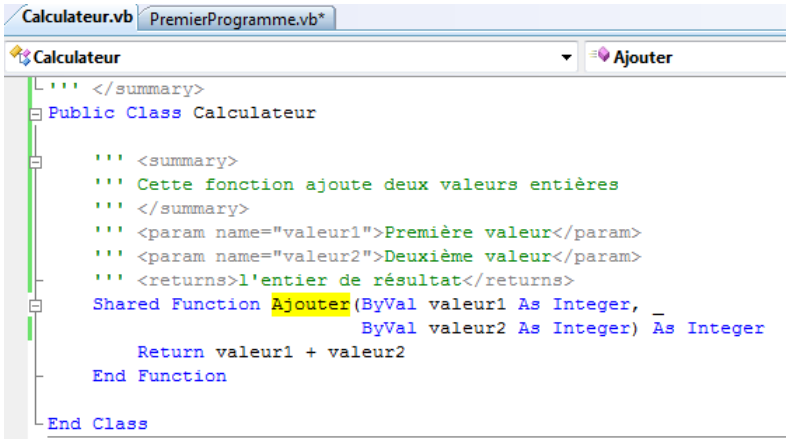
Puisque nous sommes amenés à développer momentanément principalement sur la procédure **Main** du programme **Coach.Editeur** et sur les fonctions de calcul de la librairie **Coach.Calculateur**, nous allons utiliser les fonctions de navigation de l'éditeur pour nous simplifier la vie. En effet, imaginez la difficulté pour se repérer dans une solution contenant de multiples fichiers !

- A partir de la procédure **Main** du fichier **PremierProgramme.vb**, faites un clic droit sur la fonction **Ajouter** > **Atteindre la définition** :

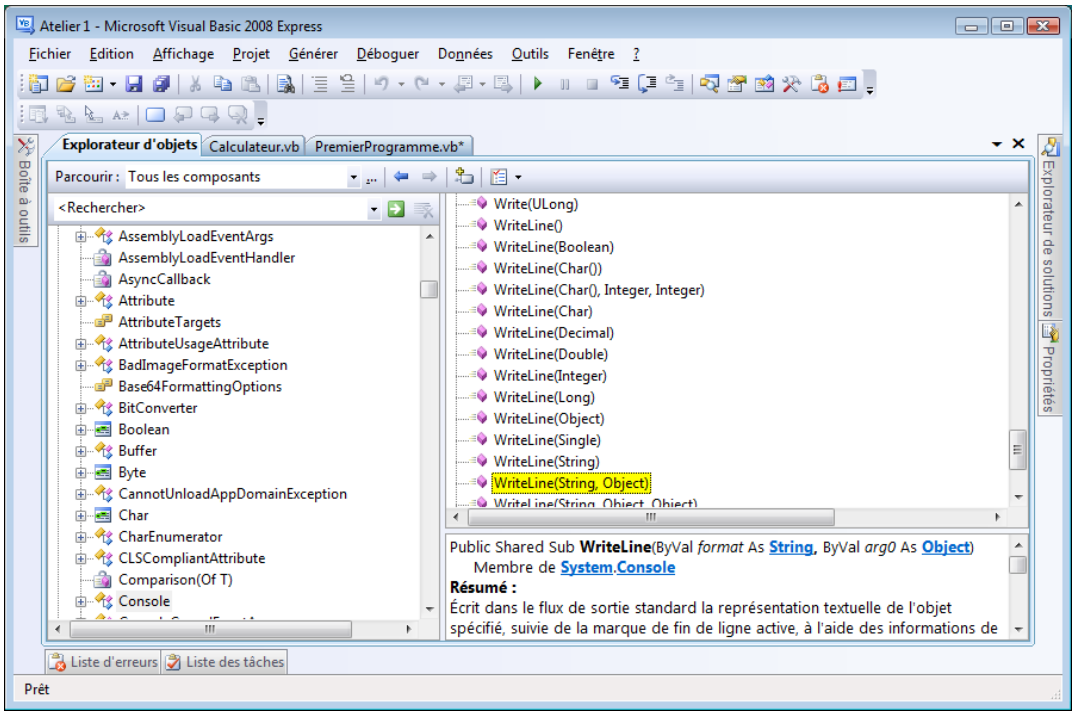
```
Sub Main()
    System.Console.WriteLine("L'addition de 10 et 5 est : {0}.", Calculateur.Ajouter(10, 5))
    System.Console.ReadKey()

End Sub
```



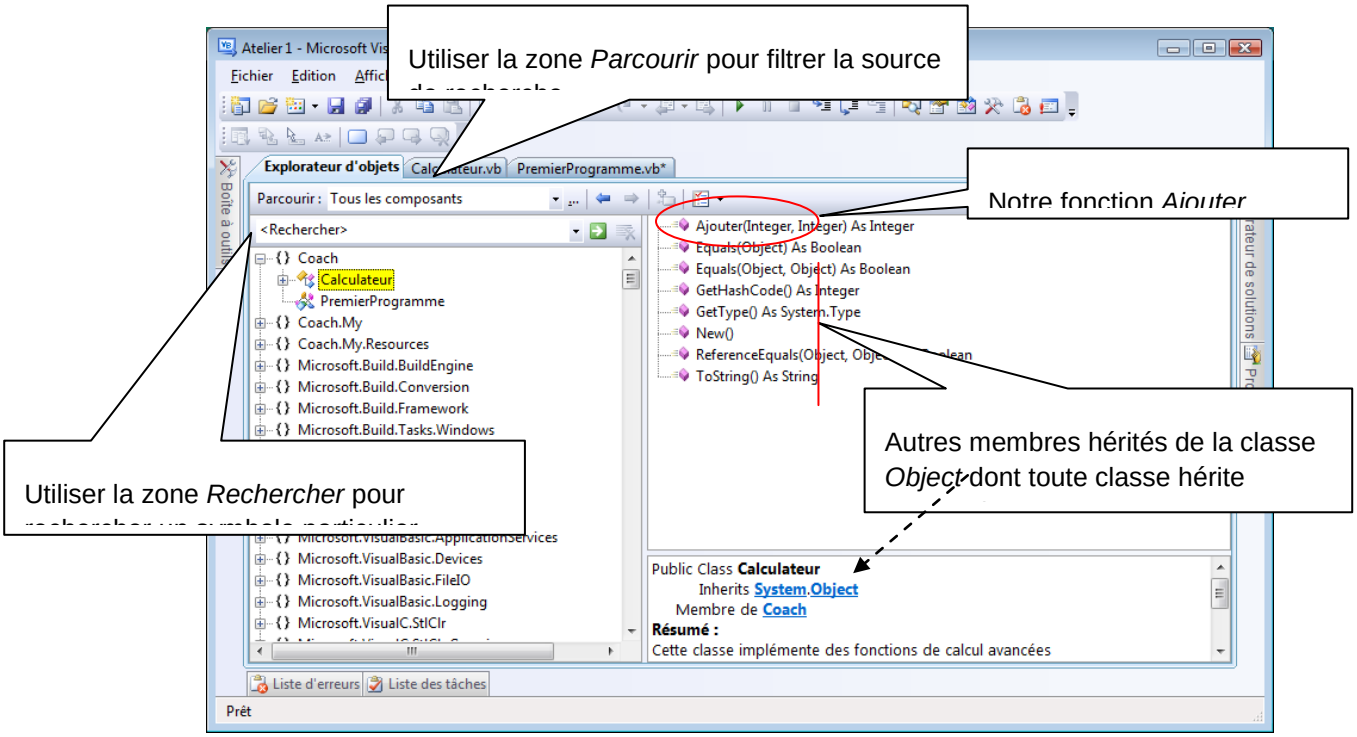
Visual Studio retrouve la définition de l'élément dans les projets de la solution en cours et vous bascule automatiquement sur la définition correspondante !



- Et si vous tentiez le coup sur une fonction du .NET Framework ? Vous ne perdez rien à essayer ☺. Toujours à partir de la procédure **Main** du fichier **PremierProgramme.vb**, faites un clic droit sur la fonction **WriteLine** > **Atteindre la définition** :



Visual Studio ouvre son **Explorateur d'objets** qui comme son nom l'indique permet d'explorer l'ensemble des bibliothèques de classes à votre disposition, que ce soit à partir du .NET Framework ou des objets de vos projets en référence. Vous y trouvez la définition des classes, des énumérations et autres symboles :



Pour en savoir plus l'utilisation de l'**Explorateur d'objets**:

http://msdn2.microsoft.com/fr-fr/library/exy1facf(VS.80).aspx

- Fermez l'**Explorateur d'objets** en cliquant classiquement sur le sigle ✕ de la fenêtre correspondante.

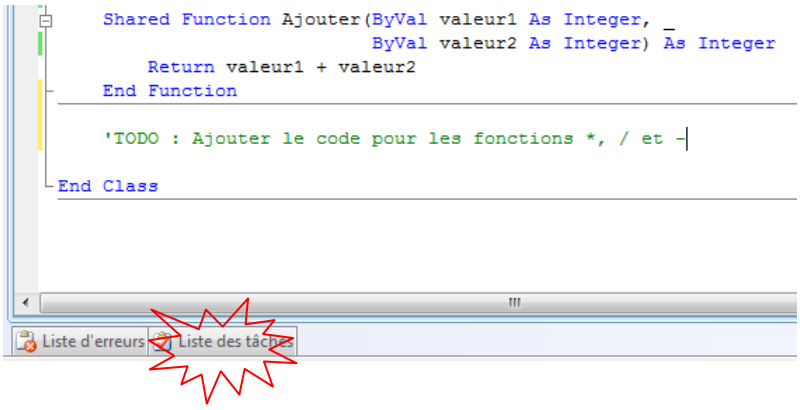
Supposons que vous projetez d'agrémenter la bibliothèque **Calculateur** de nouvelles fonctions mathématiques.

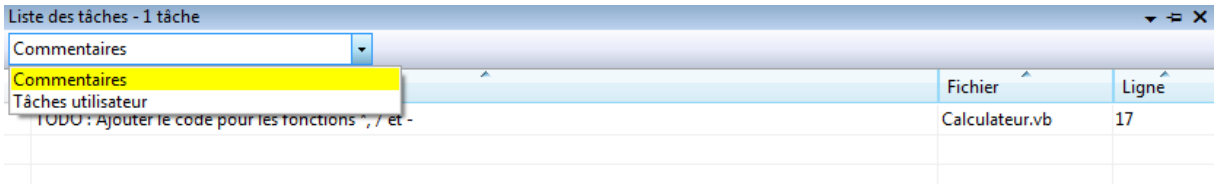
- Basculez sur le fichier **Calculateur.vb**.
- En guise de mémo, ajoutez à la suite de la fonction **Ajouter** le commentaire suivant précédé du mot **TODO** :

Code VB

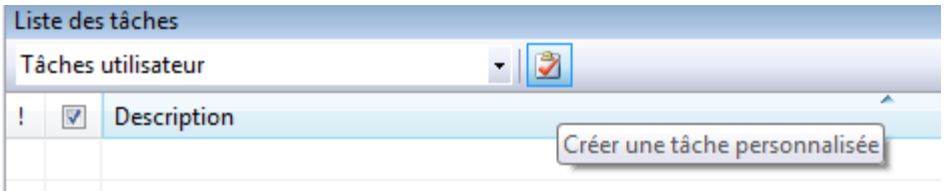
```
Public Class Calculateur


    Shared Function Ajouter(ByVal valeur1 As Integer, _
                            ByVal valeur2 As Integer) As Integer
        Return valeur1 + valeur2
    End Function


    'TODO : Ajouter le code pour les fonctions *, / et -


End Class
```

- Cliquez la « poignée » de la fenêtre **Liste des tâches** au bas de la page. Si elle n'est pas disponible, faites la apparaître via le menu **Affichage** > **Autres fenêtres** > **Liste des tâches** :





Cette fenêtre est très utile pour organiser vos tâches de programmation. Vous pouvez ajouter une tâche en sélectionnant la vue **Tâches utilisateur** dans la liste déroulante de la fenêtre puis en cliquant sur :



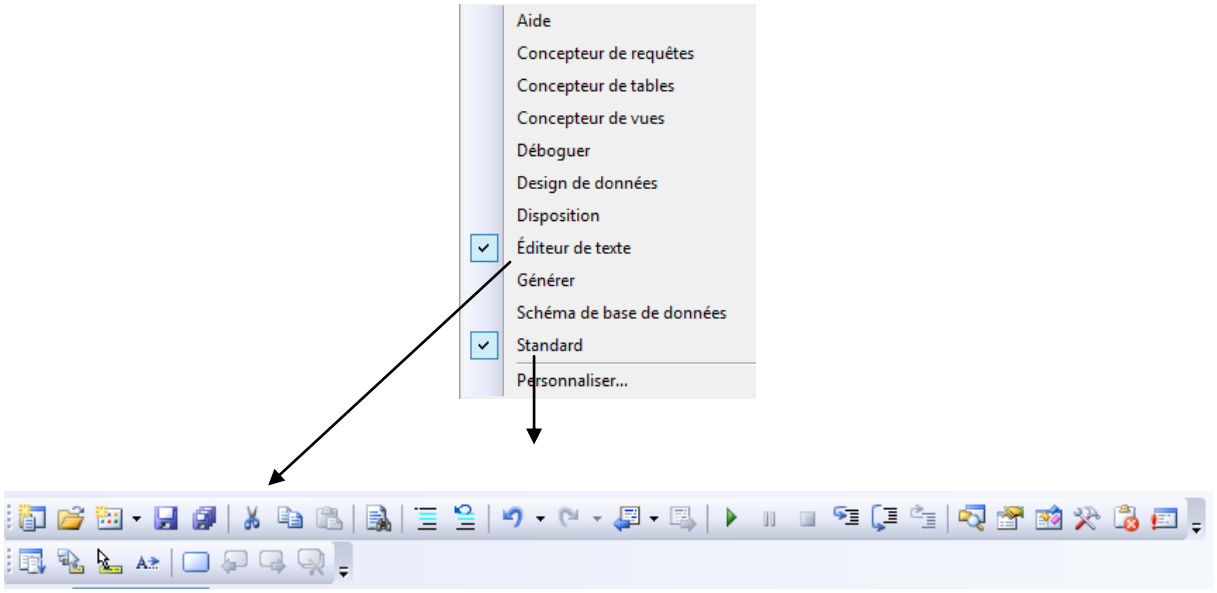
La fonctionnalité que nous venons d'exploiter consiste à utiliser les commentaires marqués d'un jeton prédéfini (par exemple la chaîne de mot clé **TODO**) pour automatiquement générer une tâche dans la liste. L'intérêt est que Visual Studio créé une sorte de raccourci vers la ligne de code correspondante. En effet, en cliquant sur le commentaire, l'éditeur s'ouvre et se positionne directement sur la ligne de commentaire correspondante :

```
    End Function

    'TODO : Ajouter le code pour les fonctions *, / et -

End Class
```

Pour consulter les jetons fournis avec Visual Studio ou pour créer vos propres jetons de commentaires personnalisés, cliquez ici.

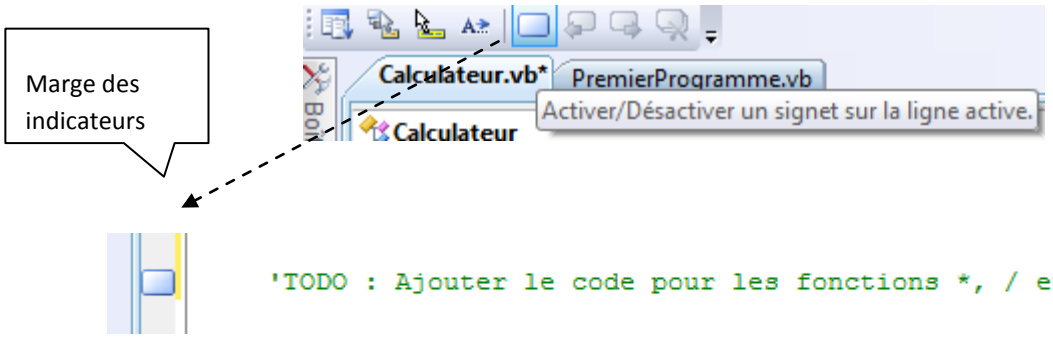
Pour basculer facilement entre les deux fichiers sur lesquels vous êtes en train de travailler, vous allez ajouter un signet en marge des lignes de code en cours de développement :

- Vérifiez que la barre d'outils de l'éditeur de texte est affichée. Si ce n'est pas le cas, sélectionnez le menu **Affichage** > **Barres d'outils** > **Editeur de texte**, ou faites un clic droit sur la barre d'outils standard et cochez la barre d'outils de l'éditeur de texte:

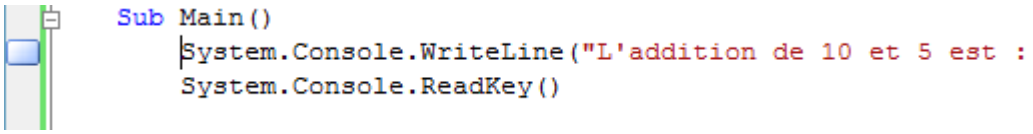


- Positionnez le curseur de la souris sur la ligne du commentaire marqué par TODO saisi précédemment.

- Cliquez sur l'icône (ou deux fois à la suite le raccourci clavier **CTRL+K**) dans la barre de l'éditeur de texte de façon à marquer la ligne d'un signet dans la marge :



- Reproduisez l'opération sur la première ligne de la procédure **Main** du fichier **PremierProgramme.vb** :

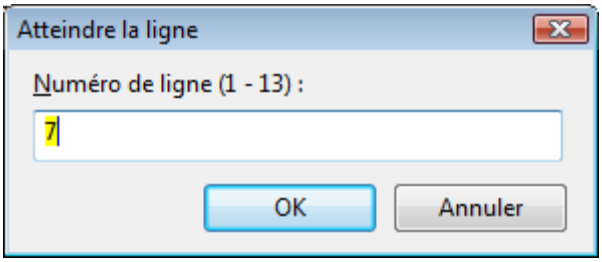


Maintenant que les lignes sont marquées d'un signet, vous allez pouvoir naviguer facilement de l'une à l'autre en utilisant les boutons (respectivement les raccourcis clavier **CTRL+K** puis **CTRL P** (pour **P**revious) ou **CTRL K** puis **CTRL N** (pour **N**ext)) de la barre d'outils de l'éditeur de texte.

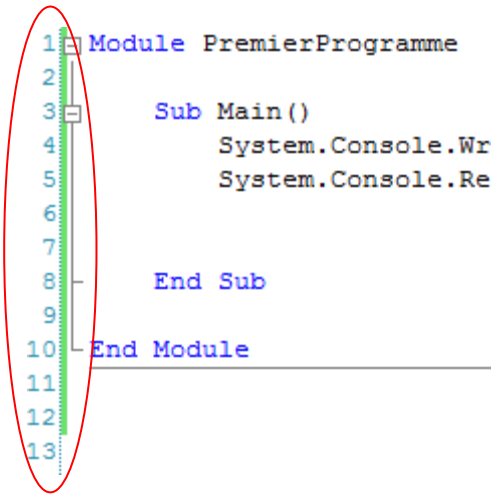
Cliquez à nouveau sur pour effacer un signet, ou sur , pour effacer en un seul clic tous les signets positionnés dans la solution.

Vous constatez que Visual Studio n'est pas en manque d'idées pour vous aider à naviguer dans le code d'une solution complexe. A vous de vous définir une stratégie efficace pour vous déplacer avec un minimum d'effort ☺.

Vous pouvez même jouer avec les numéros de ligne en utilisant le menu **Edition** > **Atteindre…**



Les numéros de ligne sont d'ailleurs affichables au niveau de la marge d'un fichier (en plus de l'indication de la barre d'état de Visual Studio qui donne la ligne en cours). Pour cela, sélectionnez le menu **Outils** > **Options** > **Editeur de texte** > **Basic** puis cochez la case **Numéros de ligne**.

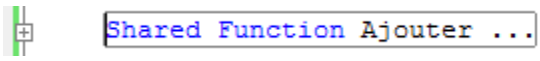


3. Structurez le code pour gagner en lisibilité.

Structurer son code peut paraître inutile mais lorsque vous multipliez les lignes de code, cela devient très vite une bonne habitude à prendre. Là encore Visual Studio propose différents mécanismes très pratiques.

- Basculez dans le fichier **Calculateur.vb**.

- Positionnez le curseur n'importe où à l'intérieur du code de la fonction **Ajouter**.
- Cliquez les touches **CTRL+M** deux fois de suite. Vous devez obtenir :

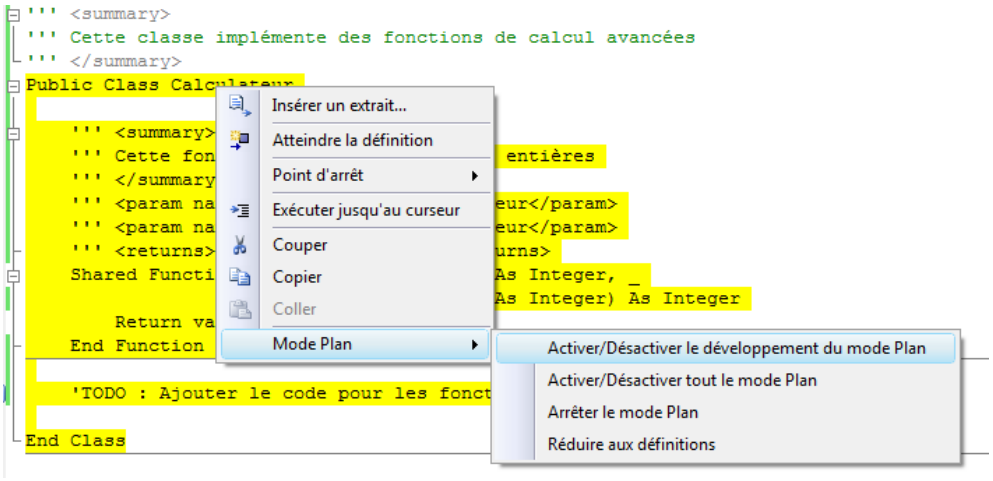


Le code de la fonction se trouve réduit à une ligne. Il n'est pas supprimé mais simplement masqué. Ce mode d'affichage s'appelle le mode **Plan**. Il permet de réduire tous les gros blocs de code délimités tels que les classes, les commentaires et les fonctions. Il suffit de refaire l'opération avec les mêmes raccourcis pour refaire apparaître le contenu de la fonction :

```
Shared Function Ajouter(ByVal valeur1 As Integer, _
                        ByVal valeur2 As Integer) As Integer
    Return valeur1 + valeur2
End Function
```

Plus simple encore, utiliser le signe plus (+) ou (- selon le contexte) situé au bord de la marge gauche pour développer ou masquer la zone.

Autre petite astuce : par exemple pour développer ou réduire plusieurs sections contigües, sélectionnez celles-ci puis utilisez le menu contextuel qui apparaît sur le clic droit de la souris > **Mode Plan** > **Activer/Désactiver le développement du mode Plan**.



Pour en savoir plus sur les commandes du **mode Plan** et les raccourcis clavier associés : http://msdn2.microsoft.com/fr-fr/library/td6a5x4s.aspx

Le must c'est que Visual Studio vous donne les moyens de décider vous-même des zones réductibles en fonction de vos critères de lecture des fichiers, à l'aide d'une directive de langage nommée #**Region**. Par exemple, pour regrouper l'ensemble des fonctions mathématiques du fichier Calculateur.vb de façon à les isoler d'un autre groupe de fonctions du même fichier, il suffirait de définir une région nommée « **Fonctions mathématiques** ».

- Encadrez la section à réduire par les directives **#Region** et **#End Region** en marquant la première avec l'intitulé qui sera visible en cas de réduction de la zone :

L'intitulé doit être en guillemets

Code VB

```
#Region "Fonctions mathématiques"

    ''' <summary>

    ''' Cette fonction ajoute deux valeurs entières
```

```
    ''' </summary>
    ''' <param name="valeur1">Première valeur</param>
    ''' <param name="valeur2">Deuxième valeur</param>
    ''' <returns>l'entier de résultat</returns>
    Shared Function Ajouter(ByVal valeur1 As Integer, _
                            ByVal valeur2 As Integer) As Integer
        Return valeur1 + valeur2
    End Function

    'TODO : Ajouter le code pour les fonctions *, / et -
#End Region
```

- A la suite ajoutez une deuxième section nommée : **Autres fonctions de calcul**. Pour cela, ajoutez la ligne suivante :

Code VB

```
#Region "Fonctions mathématiques"
…
#End Region

#Region "Autres fonctions de calcul"


End Class
```

- Tapez **Entrée** en fin de ligne. Le complément de fin de directive **#End Region** est automatiquement ajouté par Visual Studio et la région correctement positionnée en retrait dans la classe.

```
#Region "Autres fonctions de calcul"

#End Region
```

Cela veut dire que chaque fois que nous allons devoir écrire un bloc de code en VB, du type **N/End N**, il suffira de taper le mot clé du début du bloc (N) puis de valider avec **Entrée**, et Visual Studio générera automatiquement le mot clé correspondant de fin de bloc (End N).

- Réduisez les deux régions en les sélectionnant puis en cliquant **CTRL M** puis **CTRL M**. Vous obtenez un code clair et lisible :

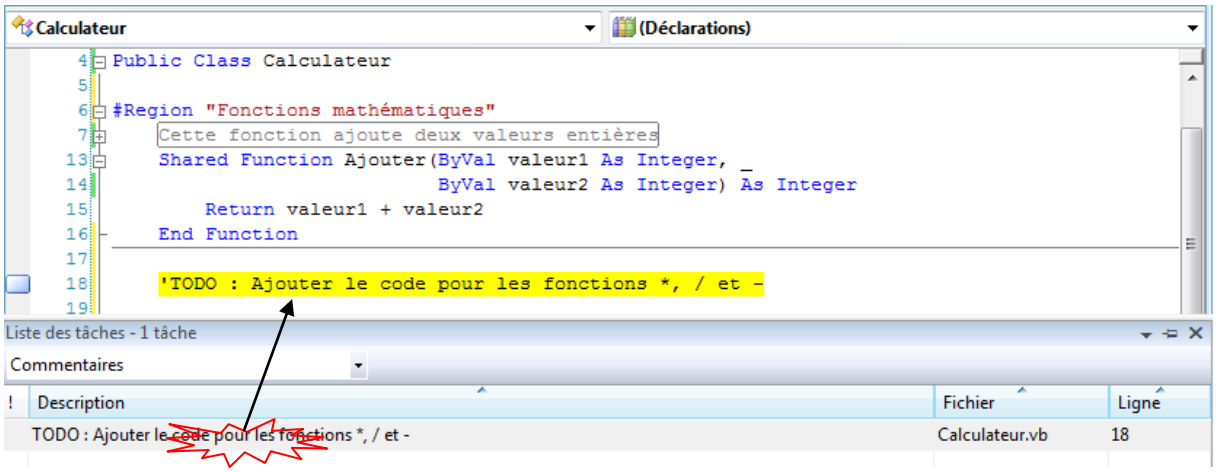
```
1  ''' <summary>
2  ''' Cette classe implémente des fonctions de calcul avancées
3  ''' </summary>
4  Public Class Calculateur
5
6      Fonctions mathématiques
20
21     Autres fonctions de calcul
24
25 End Class
```

Pour travailler sur une fonction d'une catégorie particulière, il suffit d'afficher la région concernée en gardant les autres régions masquées.

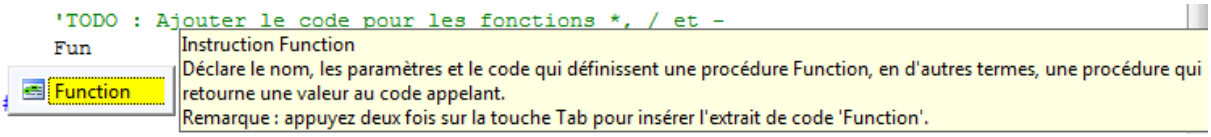
4. Utilisez les extraits de code pour coder encore plus vite.

Pour illustrer cette partie, nous allons tout simplement rajouter une nouvelle fonction **Multiplier** dans la région **Fonctions mathématiques** du **Calculateur**.

- Retrouvez le commentaire saisi précédemment à l'aide du jeton de commentaire **TODO** dans la liste des tâches et double cliquez sur la ligne correspondante pour ouvrir l'éditeur directement sur la ligne prévue pour le codage :



- Saisissez les trois premières lettres du mot clé **Function** pour commencer à coder la fonction :



Une info bulle apparaît dans laquelle l'éditeur nous engage à *appuyer deux fois sur la touche Tab pour insérer l'extrait de code 'Function'*.

Qu'est ce qu'un **extrait de code** (snippet en anglais) ?

Un extrait de code est un bloc préprogrammé reprenant les structures de base du langage pour vous aider à aller plus vite dans vos développements. Il ne vous reste à saisir que les parties du code qui varient selon le contexte.

- Suivons le guide en appuyant deux fois de suite sur la touche **TAB**.

```
'TODO : Ajouter le code pour les fonctions *, / et -

Function MyFunc() As Integer

    Return 0
End Function
```

Les paramètres variables de l'extrait de code sont en surbrillance (Le premier est présélectionné).

```
Function MyFunc() As Integer

    Return 0
End Function
```

Naviguez de l'un à l'autre avec la touche **TAB**.

- Arrêtez le pointeur de la souris sur la première zone en surbrillance **MyFunc**. Il s'agit d'une zone qui nécessite votre intervention, et Visual Studio vous donne des consignes sur la manière de la compléter via une info bulle :

```
Function MyFunc() As Integer
```

Remplacez par le nom de la fonction.

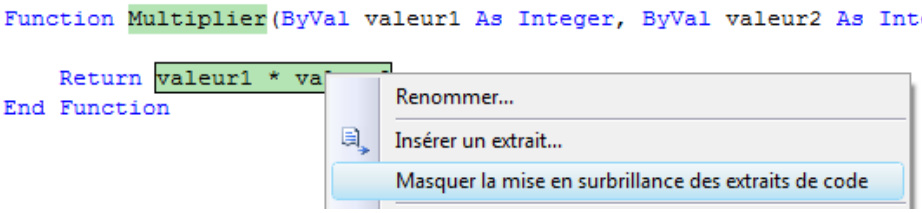
- Saisissez directement le nom **Multiplier** puis appuyez sur la touche **Tab** pour passer au paramètre suivant.

- Conservez le type de valeur de retour par défaut (ici **Integer**).
- Passez au paramètre suivant avec la touche **Tab**.
- Saisissez l'opération de multiplication entre deux paramètres **valeur1** et **valeur2**.
- Complétez le code de la fonction avec la définition des paramètres de celle-ci. Au final, vous devez obtenir :

```
Function Multiplier(ByVal valeur1 As Integer, ByVal valeur2 As Integer) As Integer

    Return valeur1 * valeur2
End Function
```

Ne vous épuisez pas à saisir **ByVal**. Visual Studio l'ajoutera pour vous par défaut.

Pour supprimer la surbrillance des paramètres, faites un clic droit n'importe où dans le code de la fonction > **Masquer la mise en surbrillance des extraits de code** ou recommencez à taper du code ailleurs dans le programme :

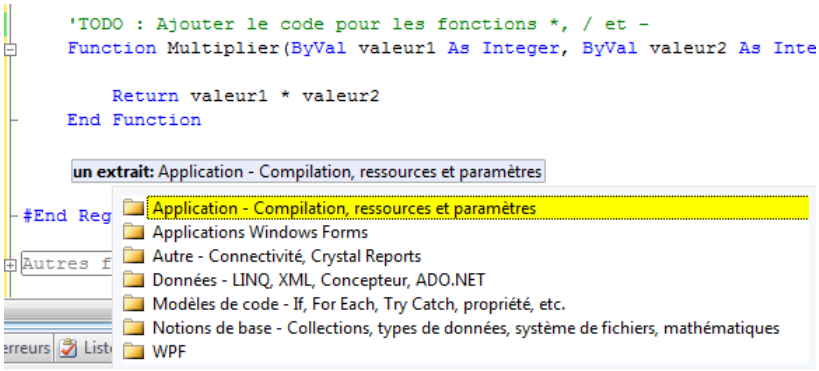


Quels sont les extraits de code fournis par Visual Basic ?

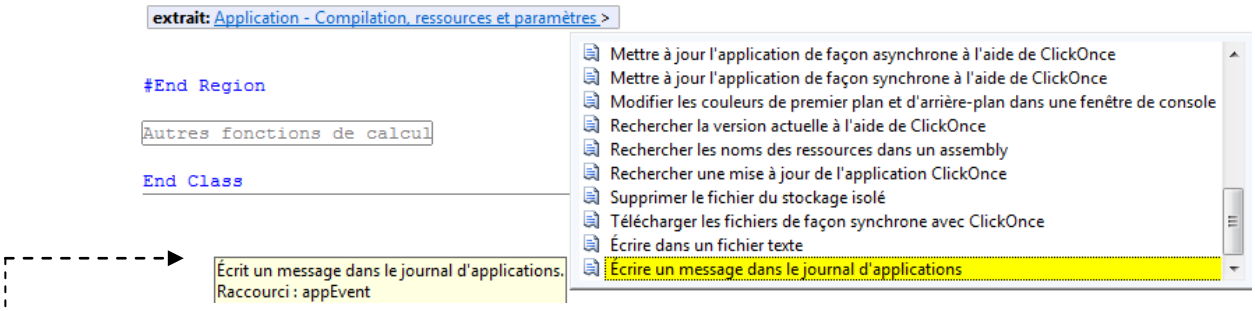
Il se trouve que dans le cas présent, nous savions qu'une fonction commence par le mot clé **Function**. En fait nous avons utilisé le *raccourci* de l'extrait pour l'afficher.

Pour consulter la liste de tous les extraits de code disponible, il suffit de vous positionner à l'endroit dans l'éditeur de code où vous souhaitez insérer un extrait, faites un clic-droit et sélectionnez le menu **Insérer un extrait…**

Une fenêtre d'aide à l'insertion apparaît :



Visual Basic fournit en standard toute une batterie d'extraits de code classés par catégories. Par exemple, sélectionnez **Application – Compilation ressource et paramètres**, puis appuyez la touche **TAB**, puis **Ecrire un message dans le journal d'applications** :



En appuyant une dernière fois sur **TAB**, vous obtenez :

```
My.Application.Log.WriteEntry("Action complete.")
```

Notez que l'info bulle de l'extrait donne une courte description et fournit le raccourci clavier pour utiliser l'extrait directement. En effet, en tapant **appEvent** à l'emplacement où vous souhaitez inclure l'extrait, suivi de deux fois la touche **TAB**, vous obtenez exactement le même résultat que par le menu contextuel. C'est ce que nous avons fait pour utiliser l'extrait de code de la fonction précédemment.

Pour tout savoir sur les extraits de code :

http://msdn2.microsoft.com/fr-fr/library/ms165392.aspx

Pour créer vos propres extraits de code :

http://msdn2.microsoft.com/fr-fr/library/ms165393.aspx

COMPILER LE CODE

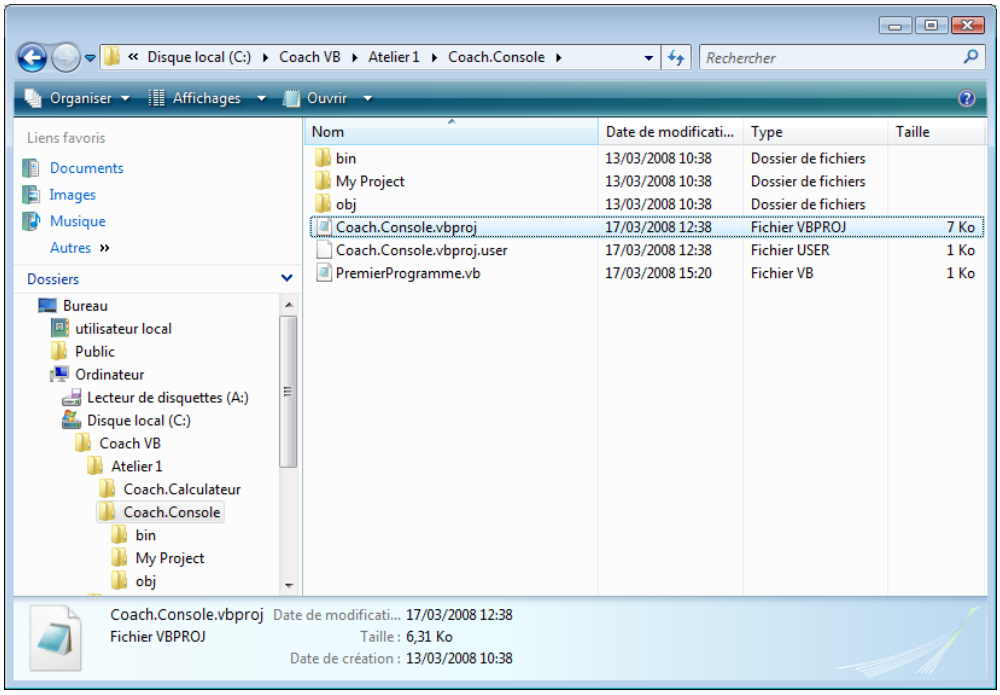
L'objectif de cet exercice est de compiler les deux projets avec Visual Studio, afin de tester leur fonctionnement.

A la fin de cet exercice, vous saurez :

- Générer une solution,
- Générer une documentation au format XML.

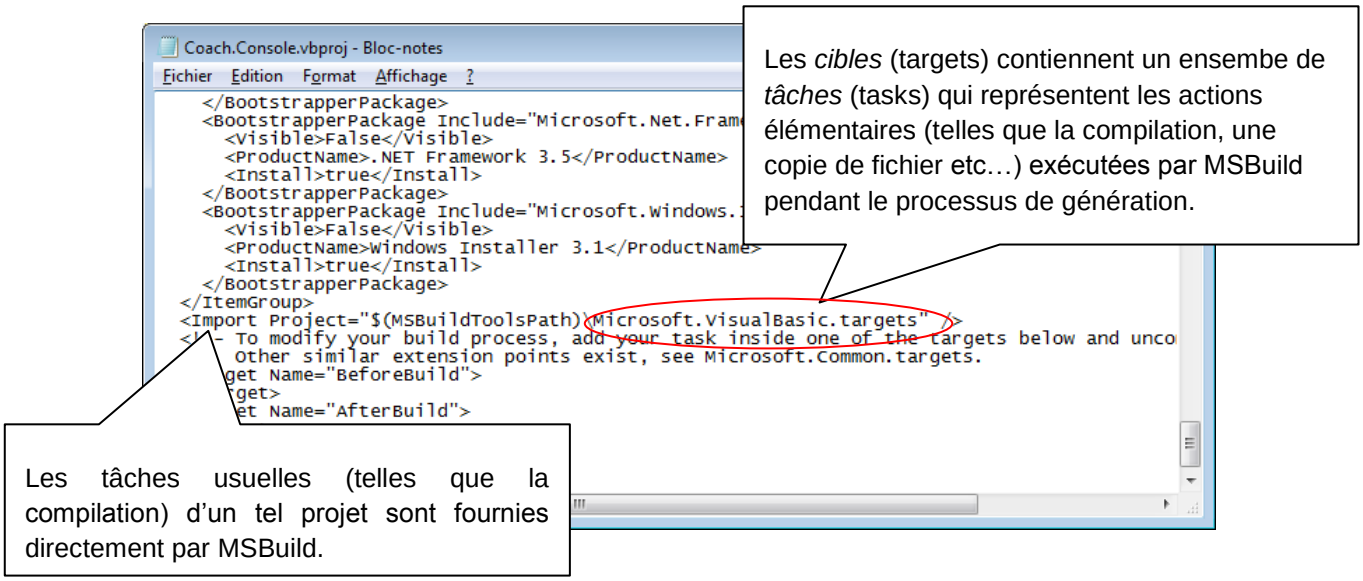
Déroulement de l'exercice :

1. Préparez la génération de la solution :
    - Basculez sur le disque pour observer le contenu des répertoires de projets de la solution. Ouvrez par exemple le répertoire du projet **Coach.Console** :

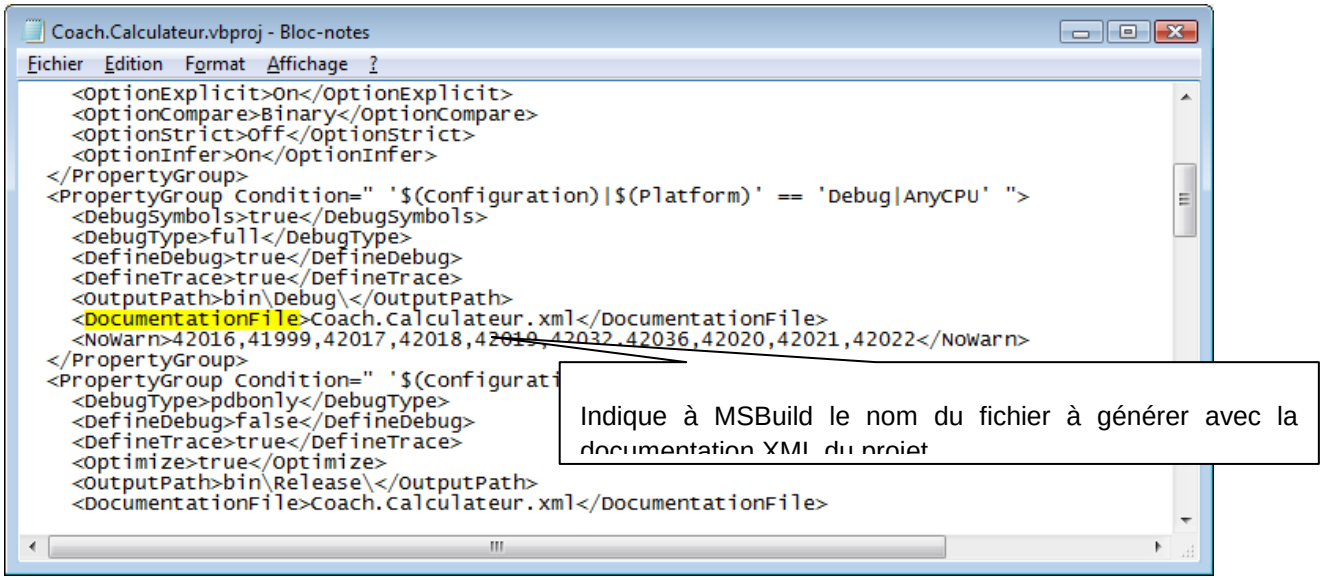


Dans chacun des sous-répertoires de projet, vous trouvez un fichier d'extension **.vbproj**. Il s'agit du fichier de projet qui est utilisé par le moteur de génération de Microsoft pour générer le ou les assembly(ies) correspondant(s). Vous pouvez l'ouvrir avec le Bloc-notes car il est au format XML.

*Exemple de contenu du fichier **Coach.Console.vbproj** :*



Autre exemple de contenu du fichier ***Coach.Calculateur.vbproj*** *:*



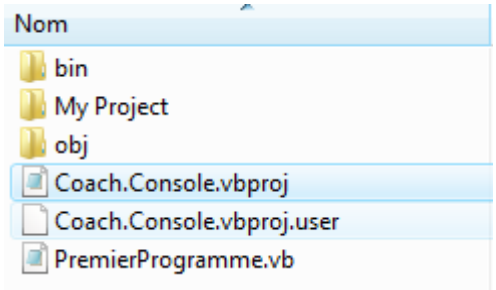
Dans l'exercice précédent vous avez compilé le projet en ligne de commandes. Or dans le cas du projet console, nous avons vu que cela demandait une référence à la bibliothèque du calculateur et que par conséquent, la ligne de commande se compliquait un peu. Il est facile d'imaginer que pour des projets complexes cela peut vite se corser…

C'est pourquoi le processus de génération de projet Microsoft est pris en charge par un moteur puissant appelé **MSBuild**. Ce moteur utilise de manière sous-jacente le compilateur que nous avons vu dans l'atelier précédent pour compiler les fichiers du projet.

Le fichier de projet d'extension **.vbproj** décrit à MSBuild les éléments à prendre en considération pour effectuer la génération. Là encore vous pourriez vous passer de Visual Studio pour créer ce fichier, mais pourquoi faire compliquer quand on peut faire simple ! Laissons Visual Studio créer ce fichier pour nous !

Pour en savoir plus sur **MSBuild**, le moteur de génération de Microsoft et Visual Studio : http://msdn2.microsoft.com/fr-fr/library/ms171452.aspx

- Revenons à la structure du projet sur le disque. Vous constatez également que le projet contient deux répertoires **bin** et **obj** :



Quelle différence y a-t-il entre les répertoires **bin** et **obj** ?

Le répertoire **obj** contient les versions temporaires des fichiers permettant de générer le projet définitif dans le répertoire **bin**. Plus exactement, à partir du répertoire **obj\debug** est généré le répertoire **bin\debug**, de même qu'à partir du répertoire **obj\release** est généré le répertoire **bin\release**.

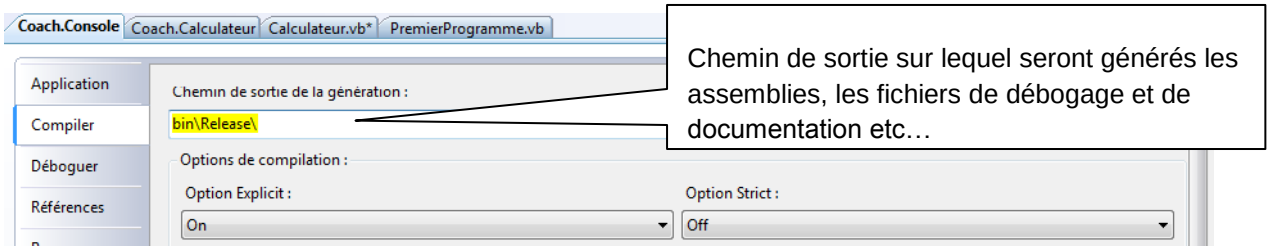
Quelle différence y a-t-il entre les répertoires **debug** et **release** ?

Dans ces répertoires sont générées des versions différentes de votre projet. Comme son nom l'indique, une version **Debug** est optimisée pour la phase de débogage, par opposition à la version **Release** qui est au contraire optimisée pour la distribution finale du programme.

En gros, en configuration **Debug**, votre programme est compilé avec des informations de débogage et n'est pas du tout optimisé (ça ne servirait à rien). A contrario, en configuration **Release**, votre programme est entièrement optimisé et ne contient pas du tout d'informations de débogage (pourrait toutefois en contenir si vous avez besoin de déboguer le projet après déploiement).

Pour savoir dans quelle configuration vous êtes, reportez-vous à la fenêtre de propriétés des projets de la solution.

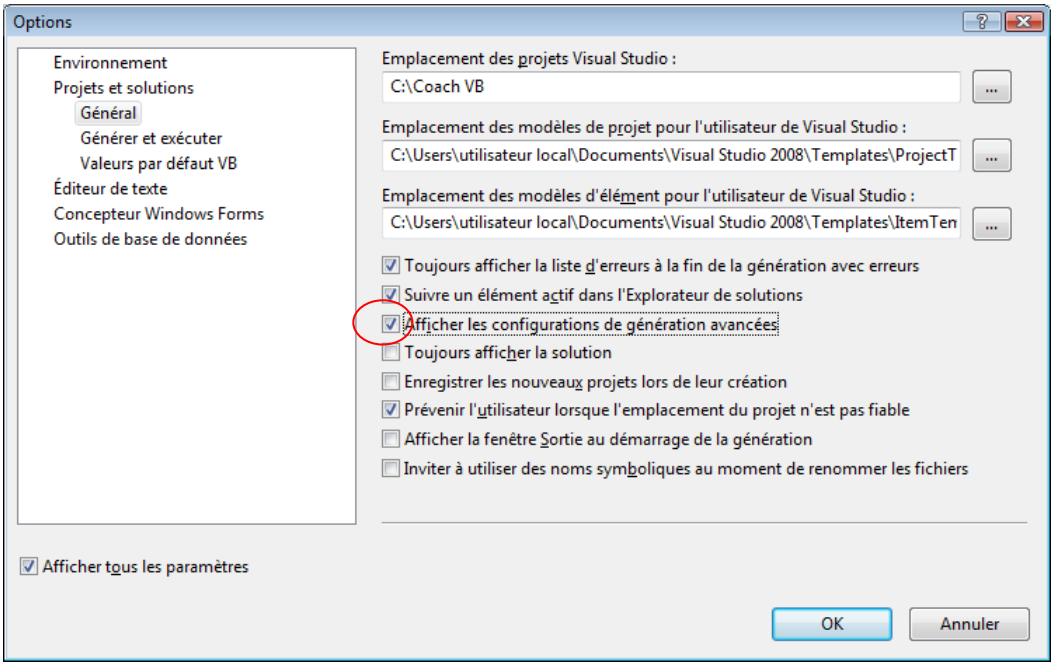
- Dans l'**Explorateur de solutions** de Visual Studio, double cliquez sur **MyProject** respectivement dans les projets **Coach.Console** et **Coach.Calculateur** pour afficher la fenêtre de propriétés des projets.
- Sélectionnez l'onglet **Compiler** pour voir les caractéristiques de la configuration de génération du projet. Notez le chemin de sortie indiqué et les principales options de compilation :



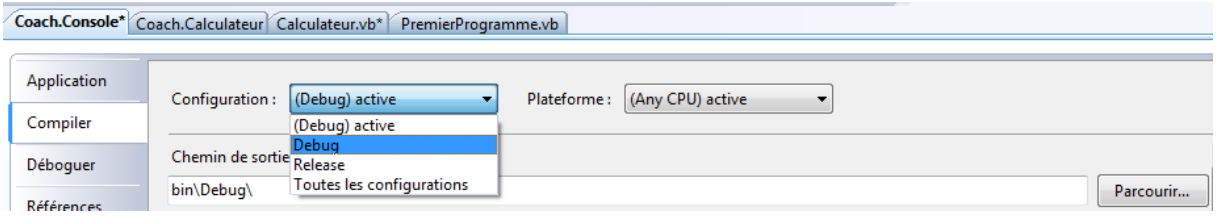
Cette option impose la déclaration explicite de toute variable dans le code

Cette autre option limite les conversions implicites de type de données (nous aurons l'occasion d'y revenir ultérieurement dans ce tutorial)

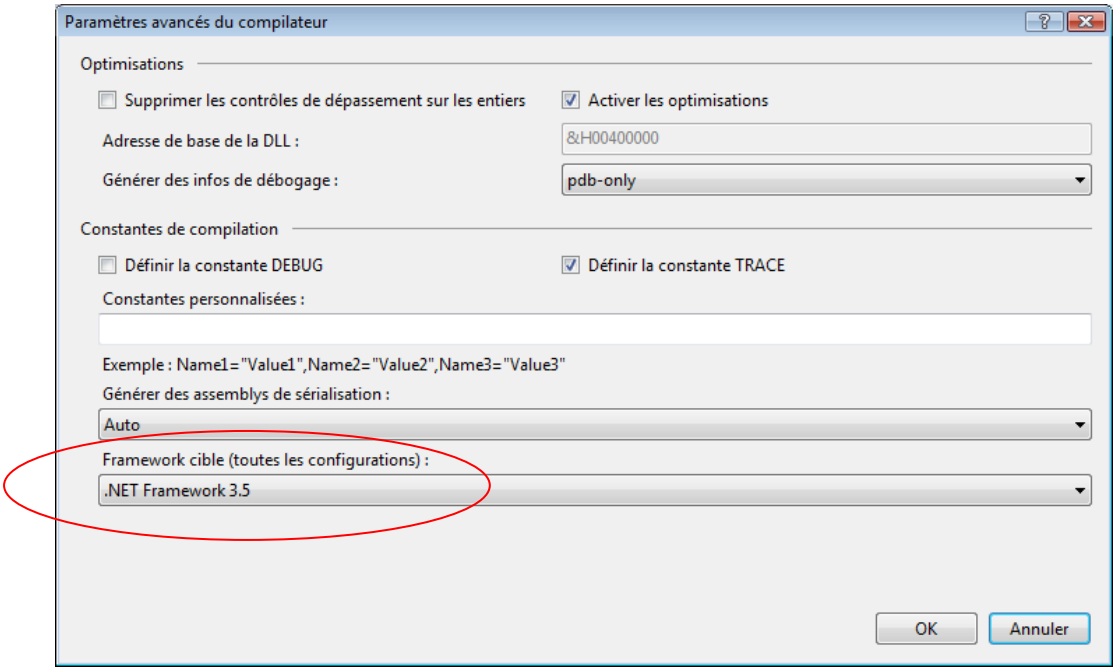
Pour simplifier, le système de projet de Visual Studio décide par défaut de la version du programme à générer, **Debug** ou **Release**. Pour afficher une configuration plus fine, sélectionnez le menu **Outils** > **Options** > **Projets et solutions** > **Général** et cochez **Afficher les configurations de génération avancées**.



- Rebasculez sur les fenêtres de configuration des options de compilation des projets. Vous devez maintenant pouvoir choisir la version du projet à générer comme suit. Sélectionnez la version **Debug** pour les deux projets **Coach.Console** et **Coach.Calculateur** :



- Toujours à partir de la fenêtre de propriétés et l'onglet **Compiler**, cliquez le bouton **Options avancées de compilation…**
- Dans la fenêtre **Paramètres avancés du compilateur**, notez la version du Framework cible enregistrée :



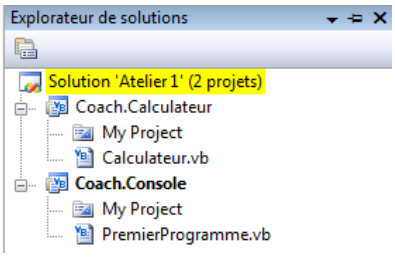
Il s'agit d'une nouvelle fonctionnalité très sympa de Visual Studio 2008, appelée **multi-ciblage**, qui consiste à délier l'environnent de développement de la plate-forme d'exécution. En clair, ce n'est pas parce que vous avez installé la dernière version de Visual Studio que vous êtes contraint de développer avec la toute dernière version du .NET Framework. A vous de configurer la version du Framework adéquate en fonction de la plate-forme cible sur laquelle vous projetez de déployer votre projet ! Souvenez-vous que le .NET Framework fournit entre autres le compilateur et l'environnement

d'exécution (runtime).

- Enregistrez les modifications de configuration que vous avez faites sur les fenêtres de propriétés des deux projets de la solution.

2. Générer la solution :

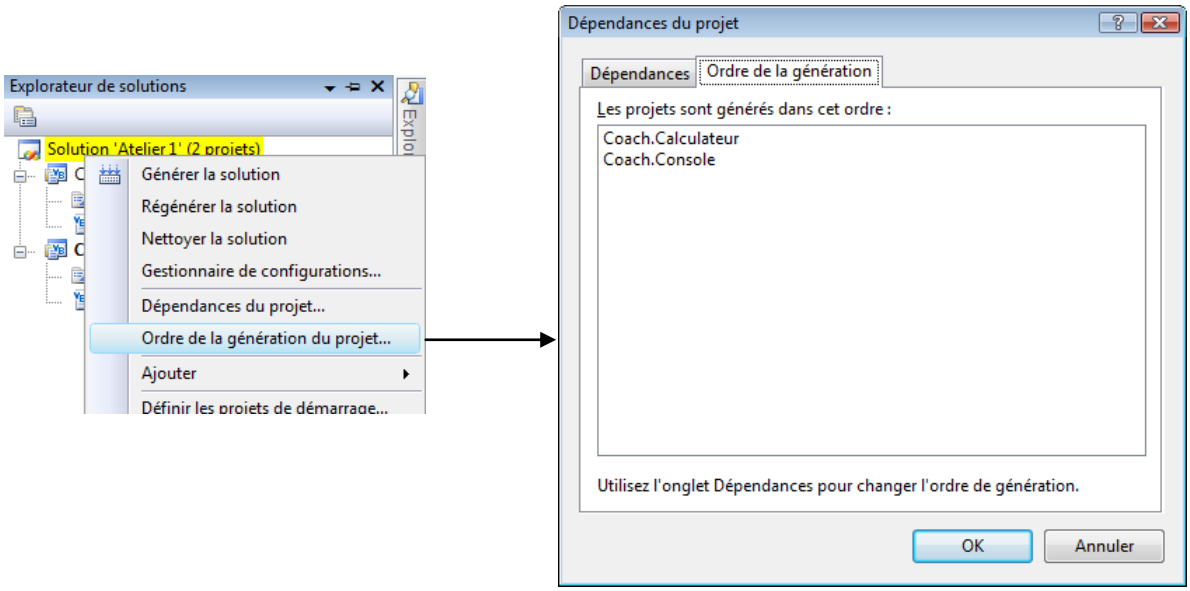
   - Ouvrez l'**Explorateur de solutions**.



Comme vous avez plusieurs projets, on peut se demander dans quel ordre Visual Studio va les compiler et s'ils doivent être compilés ensemble ou séparément ?

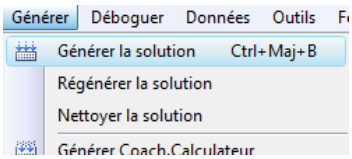
A priori, comme le projet Console a besoin du Calculateur, ce serait judicieux de les compiler ensemble, en commençant bien sûr par la bibliothèque du calculateur.

- Faites un clic droit sur la solution > **Ordre de la génération du projet…** pour vérifier l'ordre de compilation des projets :

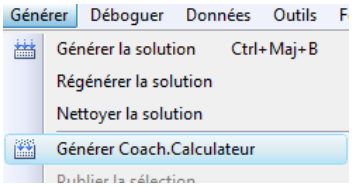


Pour procéder à la génération, vous avez plusieurs choix :

- Soit vous générez toute la solution en cliquant la racine de la solution dans l'Explorateur de solutions puis le menu de Visual Studio **Générer** > **Générer la solution.** La génération des projets se fait dans l'ordre vu précédemment.



- Soit vous générez les projets séparément en cliquant le dossier du projet dans l'Explorateur de solutions puis le menu **Générer** > **Générer <LeNomDuProjetsélectionné>**



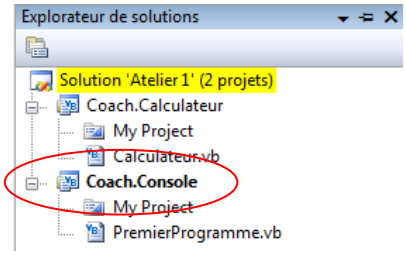
Le clic droit sur les dossiers correspondants de l'Explorateur de solutions propose dans un menu contextuel exactement les mêmes options de génération que le menu principal de Visual Studio.

Pour gagner du temps, vous pouvez bien sûr lancer directement l'exécution du projet dans la foulée, directement à la suite de la génération. Nous y reviendrons dans un instant. Il faut utiliser :

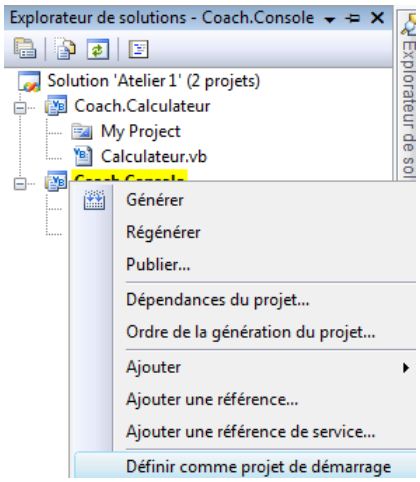
- le menu **Déboguer** > **Démarrer le débogage**,
- ou l'icône de la barre d'outils standard,
- ou encore le raccourci clavier **F5**.

Mais dans ce cas, quel est le projet qui est lancé le premier ?

Visual Studio nous l'indique en affichant le nom du projet en caractères gras dans l'Explorateur de solutions.



- Si le projet **Coach.Console** n'est pas configuré en tant que projet de démarrage, faites un clic droit à la racine du projet > **Définir en tant que projet de démarrage** :

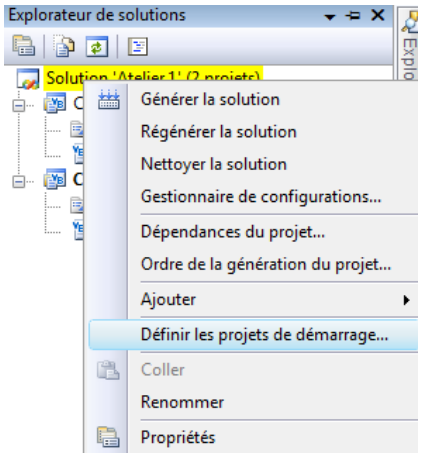


Est-ce que cela aurait un sens de démarrer sur notre projet **Calculateur** ?

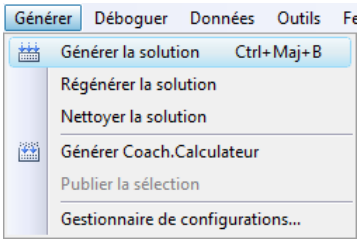
Non bien sûr, puisqu'il s'agit d'une bibliothèque de classes donc d'une dll. D'ailleurs Visual Studio ne nous l'autoriserait pas en affichant un message d'erreur au lancement de l'application.

Est-ce qu'on peut démarrer plusieurs projets en même temps ?

Et pourquoi pas ? Vous pourriez avoir plusieurs couches de présentation (interfaces) sur un projet multi-tiers. Dans ce cas, faites un clic droit sur la solution > **Définir les projets de démarrage** pour programmer tous les projets à démarrer à l'exécution de la solution.



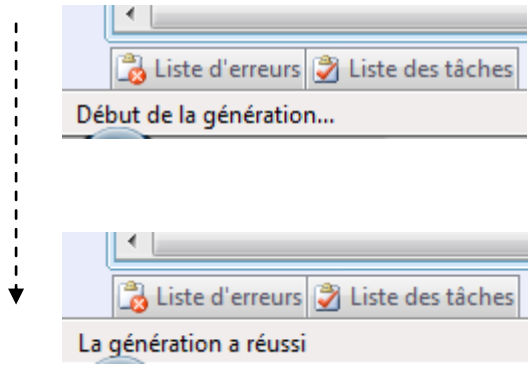
- Pour générer la solution et vérifier que votre code ne comporte pas d'erreur, sélectionnez le menu **Générer** > **Générer la solution** de Visual Studio :



Quelle est la différence entre les options de menu **Générer la solution** et **Régénérer la solution** ?

- **Générer la solution** : effectue une génération « différentielle » en ne compilant que les fichiers qui ont été modifiés depuis la dernière génération.
- **Régénérer la solution** : effectue une régénération complète de la solution entière, c'est-à-dire en incluant l'ensemble des fichiers des projets (qu'ils aient été modifiés ou non depuis la dernière génération) et surtout en procédant à un *nettoyage* des fichiers intermédiaires et de sortie (dont les assemblies, les fichiers de débogage et les fichiers de documentation xml) sur le disque.
- **Nettoyer la solution** : effectue un *nettoyage* de tous les fichiers intermédiaires et de sortie sur le disque.

- La barre d'état de Visual Studio indique (en bas à gauche de l'écran) les étapes successives de la génération jusqu'à sa réussite :

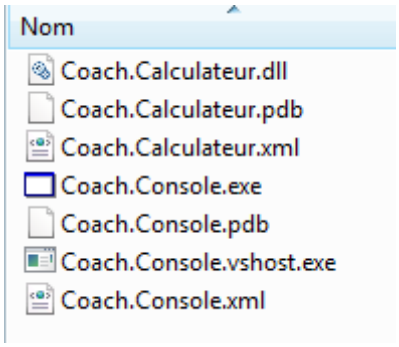


- Basculez dans l'**Explorateur de Windows** pour voir ce qui a été généré sur le disque :

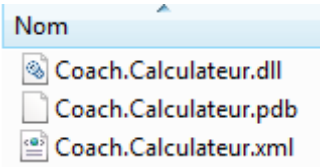
Où ont été générés les fichiers de sortie ?

Dans le répertoire **\bin\Debug** de chaque projet puisque nous avons configuré les projets en version **Debug** au début de cet exercice.

*Contenu du répertoire **Coach.Console\bin\Debug** :*



*Contenu du répertoire **Coach.Calculateur\bin\Debug** :*



Quels types de fichiers avez-vous en sortie ?

- ***.dll/*.exe** : ce sont les assemblies de chaque projet : Coach.Console.exe pour le projet d'application de commandes, et Coach.Calculateur.dll pour la bibliothèque de classes.
- ***.pdb** : ce sont des fichiers contenant les informations de débogage des assemblies.
- ***.vshost.exe** : ce sont des fichiers destinés uniquement à l'usage de Visual Studio. (Par exemple, ils ne doivent pas être déployés avec l'application). Ils servent au **processus d'hébergement** de Visual Studio, mécanisme destiné à améliorer les performances du débogage des applications.

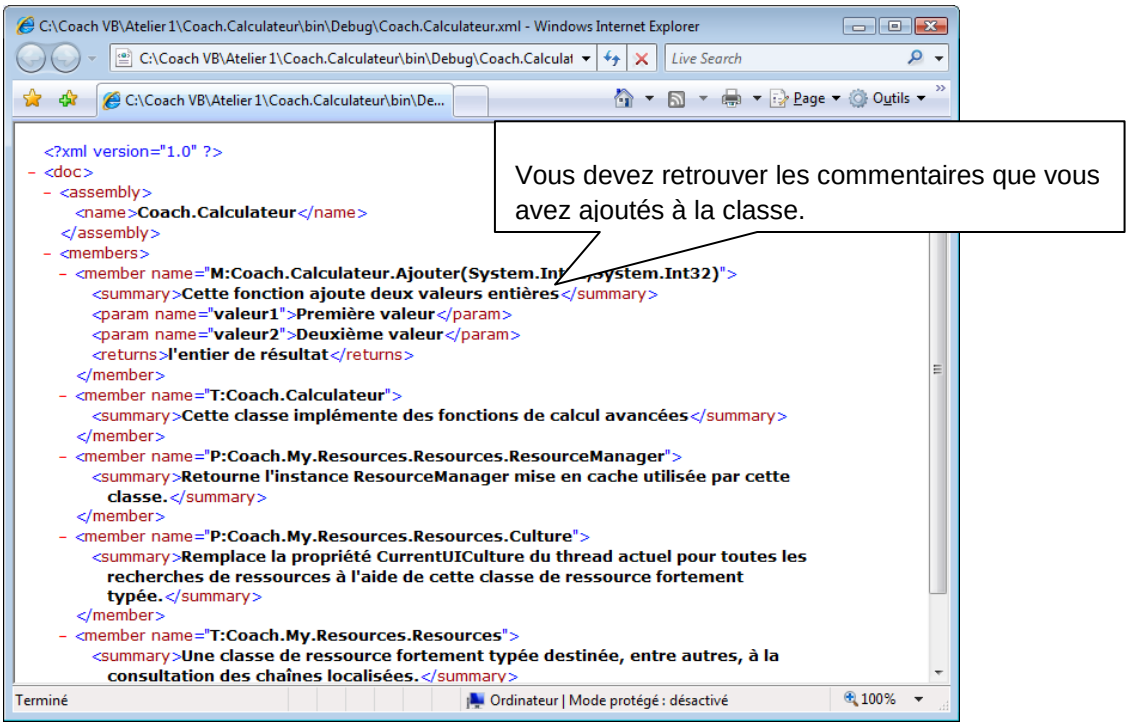
Pour en savoir plus sur le processus d'hébergement :

http://msdn2.microsoft.com/fr-fr/library/ms185331.aspx

- ***.xml** : ce sont les fichiers contenant les documentations au format XML, extraites des fichiers de code.

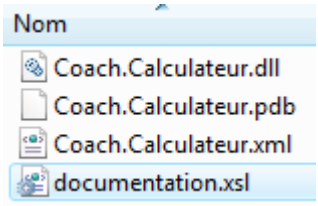
- Faites un **double clic** sur le fichier **Coach.Calculateur.xml** du dossier

  ..**Coach.Calculateur\bin\Debug**. Il doit s'ouvrir dans votre navigateur Internet.



Il faut bien reconnaître que pour une documentation, ce n'est pas très lisible ☺. Nous allons le rendre plus facile d'utilisation en lui appliquant une feuille de style XSL.

- Faites un copier/coller du fichier **documentation.xsl**, à partir du répertoire contenant les fichiers utiles de l'atelier (**..\Atelier 1\Fichiers utiles**), à destination du répertoire contenant le fichier de documentation XML :



- En utilisant le **Bloc-notes**, ouvrez le fichier **Coach.Calculateur.xml** ;

- Juste en dessous de la balise de définition **<?xml**, ajoutez la ligne de référencement de la feuille de style de transformation :

  <?xml-stylesheet type="text/xsl" href="documentation.xsl"?>

Code XML de calculateur.xml

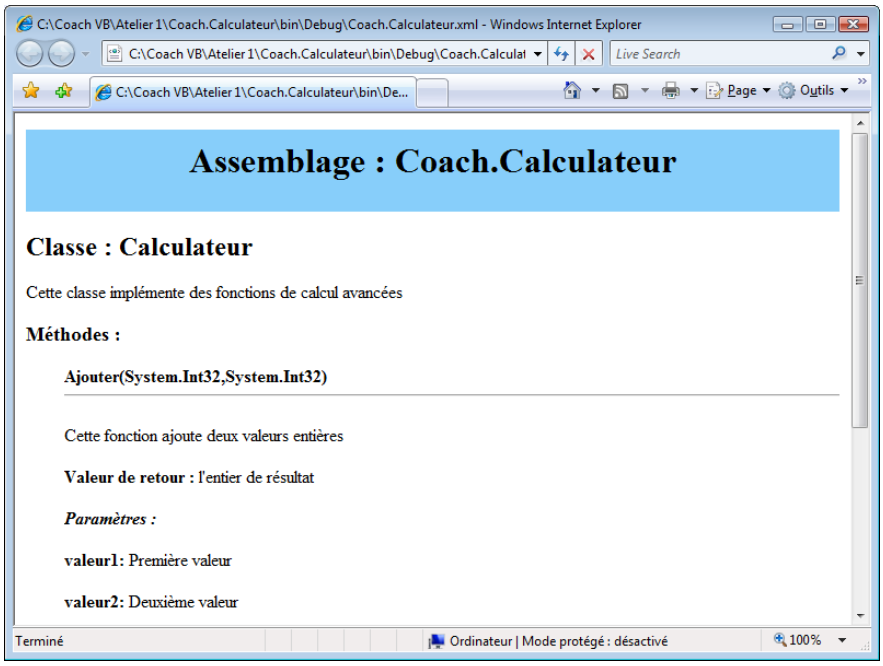
```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="documentation.xsl"?>
<doc>
<assembly>
<name>
Coach.Calculateur
</name>
</assembly>
<members>
<member name="M:Coach.Calculateur.Ajouter(System.Int32,System.Int32)">
        <summary>
 Cette fonction ajoute deux valeurs entières
 </summary>
        <param name="valeur1">Première valeur</param>
        <param name="valeur2">Deuxième valeur</param>
        <returns>l'entier de résultat</returns>
</member><member name="T:Coach.Calculateur">
        <summary>
 Cette classe implémente des fonctions de calcul avancées
 </summary>
```

```
…
</doc>
```

Notez que le générateur de documentation assigne des ID aux noms des éléments. Par exemple, la lettre **T** est le préfixe des types de données. Voici la liste des principaux préfixes :

Préfixe	***Elément***
E	Evénement (Event)
F	Champ (Field)
M	Méthode (Method)
N	Espace de noms (Namespace)
P	Propriété (Property)
T	Type
!	Erreur

- Sauvegardez le fichier **Coach.Calculateur.xml** ;
- Faites un **double clic** sur votre fichier **Coach.Calculateur.xml** :



C'est déjà plus lisible ☺ !

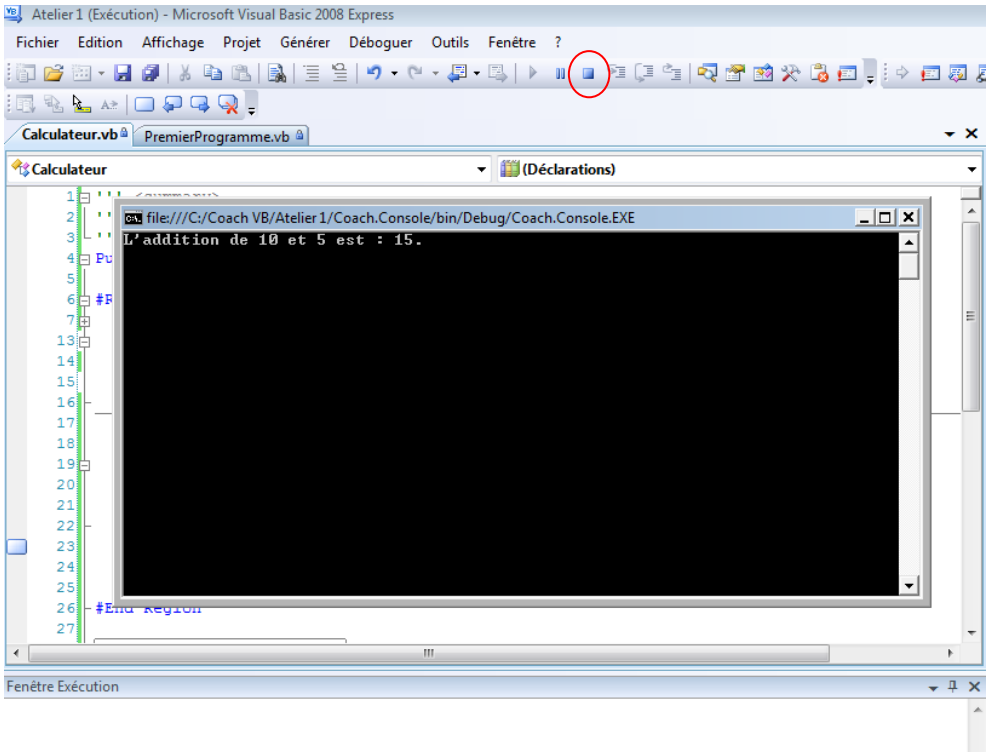
La feuille de style fournie en exemple est simple, mais vous pouvez la modifier pour l'adapter à vos besoins. Il est aussi possible de faire un petit outil Windows ou un petit site web de visualisation des fichiers de documentation, en associant automatiquement le fichier XML avec la feuille de style XSL.

Voici un article (en anglais) de juin 2002 mais toujours d'actualité (la feuille de style de l'atelier s'en inspire) :

http://msdn.microsoft.com/msdnmag/issues/02/06/XMLC/

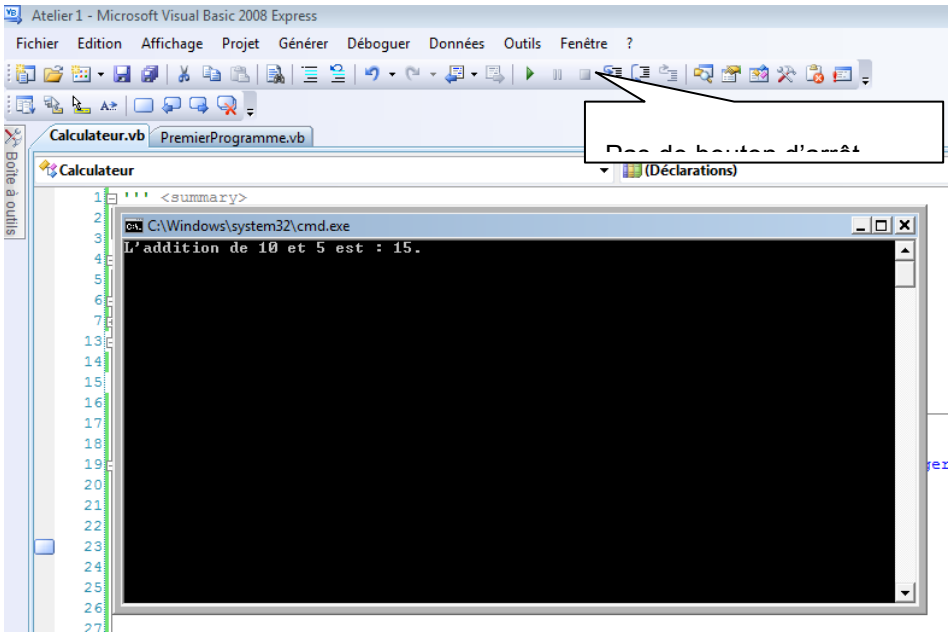
3. Testez maintenant le fonctionnement de l'application :

    - Lancez l'application en utilisant le menu **Débogage**, la flèche dans la barre d'outils ou via le raccourci **F5**.



Cliquez sur l'icône dans la barre d'outils standard de Visual Studio pour arrêter le mode débogage ou fermez l'application en cours d'exécution.

Si vous pensez ne pas avoir besoin de déboguer (ce qu'on espère toujours au fond de nos âmes fières), appuyez **CTRL + F5** (de toutes façons, il est toujours temps de travailler avec le mode débogage une fois que l'application a explosé en plein vol). L'application se lance sans charger les informations de débogage donc plus rapidement.



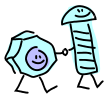
4. Et si le projet contient des erreurs de compilation…, que se passe-t-il ?

   - Editez le fichier de code **Calculateur.vb** du projet **Coach.Calculateur**.
   - Supprimez le mot clé **Shared** au début de la fonction **Ajouter**.
   - Enregistrez vos changements.

```
Function Ajouter(ByVal valeur1 As Integer, _
                 ByVal valeur2 As Integer) As Integer
    Return valeur1 + valeur2
End Function
```

A priori, ni vu ni connu, tout se passe bien ☺.

Mais les choses vont commencer à se gâter si vous éditez le fichier **PremierProgramme.vb** du projet **Coach.Console**, qui utilise la fonction **Ajouter** du **Calculateur**.



En effet, le mot clé **Shared**, comme nous aurons l'occasion de le revoir plus tard dans ce tutorial, sert à indiquer que la fonction est directement utilisable sans besoin d'instancier la classe qui la contient, l'objectif étant de fournir une sorte de bibliothèque de fonctions (type API). Donc sans le mot clé **Shared**, il faudrait passer par l'instanciation d'un objet de type **Calculateur** avant de pouvoir prétendre utiliser la fonction.

- Basculez dans le fichier **PremierProgramme.vb** du projet **Coach.Console** :

```
Sub Main()
    System.Console.WriteLine("L'addition de 10 et 5 est : {0}.", Calculateur.Ajouter(10, 5))
    System.Console.ReadKey()
End Sub
```

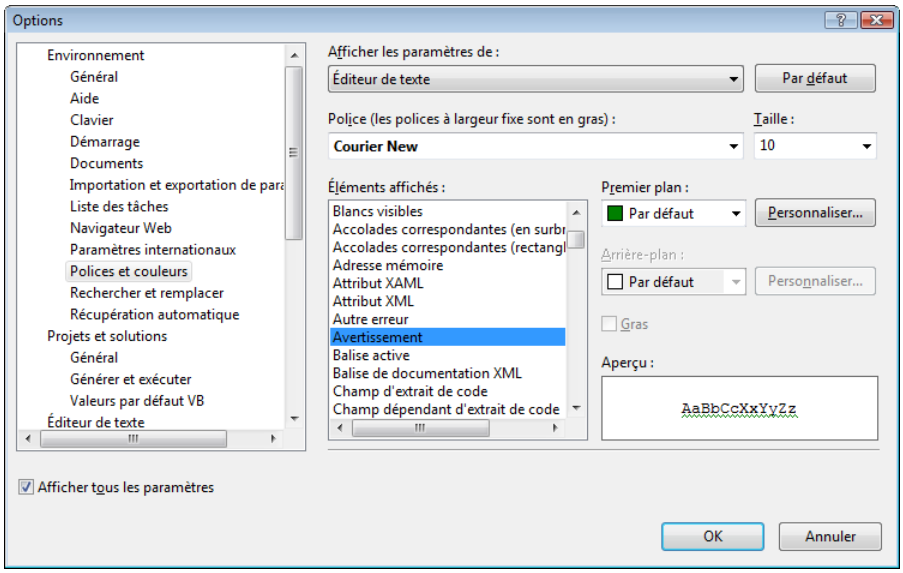
Vous avez remarqué le surligné bleu en dessous de l'appel à la méthode ?

Il faut comprendre que Visual Studio n'attend pas que vous génériez la solution pour vous prévenir qu'il y a un malaise !

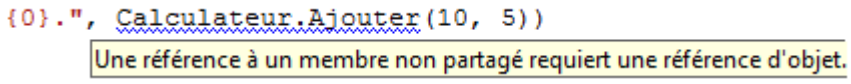
A l'aide de codes couleurs, il attire votre attention sur les morceaux de code qu'il détecte comme pouvant être source d'ennui (notamment à la compilation).

Où sont référencés les codes couleurs utilisés ?

Souvenez-vous, nous en avons déjà parlé. Tous les codes couleurs sont modifiables dans le menu **Options** > **Outils** > **Environnement** > **Polices et couleurs**. Par exemple un surligné vert indique un avertissement, un surligné bleu (comme ici) une erreur de compilation, un surligné rouge, une erreur de syntaxe :



- Corrigez le problème en arrêtant la souris sur le code surligné pour voir le problème identifié par Visual Studio :



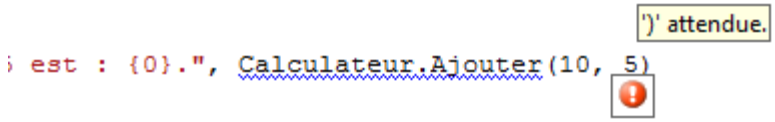
Comme prévu, Visual Studio s'étonne que vous ne définissiez pas une instance d'objet avant de vous précipiter à utiliser la fonction **Ajouter** ☺. Evidemment à vous de déterminer une stratégie pour corriger le souci. Soit vous déclarez l'instance d'objet attendue, soit vous ajoutez le mot clé *Shared* dans la définition de la fonction !

- Titillons encore un peu Visual Studio si vous voulez bien ☺. Supprimez maintenant la **parenthèse de fin** de cette même ligne de code, puis valider la ligne par **Entrée** :

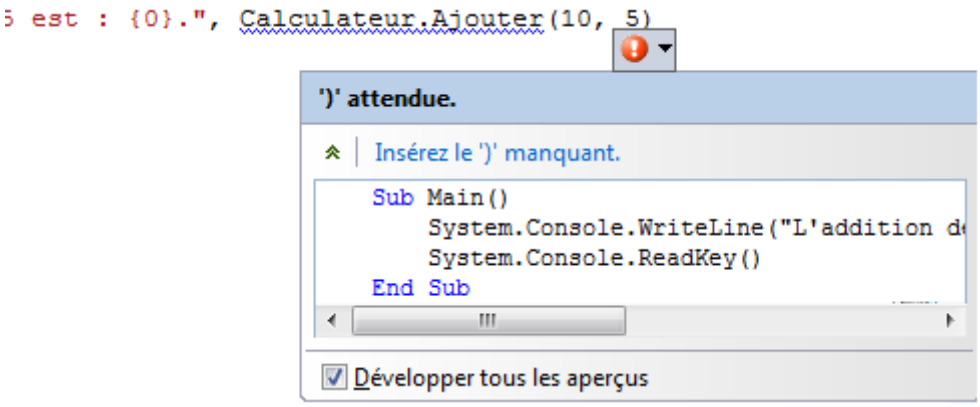
```
Sub Main()
    System.Console.WriteLine("L'addition de 10 et 5 est : {0}.", Calculateur.Ajouter(10, 5)
    System.Console.ReadKey()
End Sub
```

Ici, Visual Studio réagit différemment car le problème est tellement évident (franchement vous n'assurez pas ☺) qu'il se paie le luxe de vous proposer directement la solution et de corriger le problème pour vous (décidemment c'est agaçant cette manie de croire que nous les développeurs nous avons un poil dans la main ☺). Enfin, il nous demande (quand même !) de valider la solution avant de l'appliquer, en utilisant le mécanisme des balises actives (smart tags) qui existent depuis la sortie d'Office XP.

- Positionnez la souris sur le petit trait rouge qui apparait en bout de ligne. La balise active apparaît :



- Cliquez ensuite sur l'icône pour faire apparaître le détail de la solution proposée :

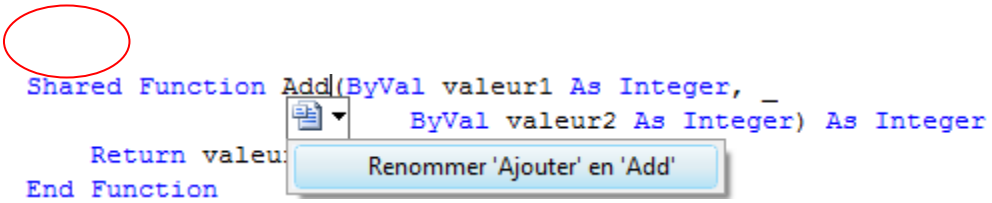


- Cliquez sur **Insérer le ')' manquant** dans la fenêtre active pour déclencher la correction. Visual Studio rajoute automatiquement la parenthèse.

Oui, évidemment juste pour une parenthèse, vous n'êtes pas très convaincu…

Dans le même genre, essayez de renommer la fonction **Ajouter** dans le fichier **Calculateur.vb**. Une balise active apparaît pour vous proposer de renommer la fonction partout où elle est utilisée dans la solution.

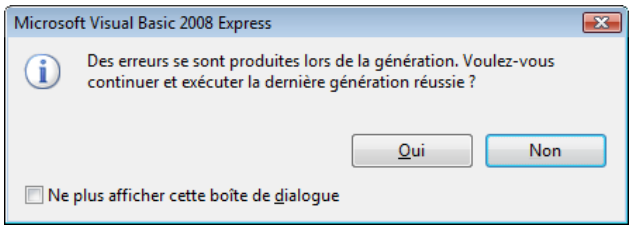
Vous imaginez tous les domaines d'application d'une aide à la saisie de ce type ?



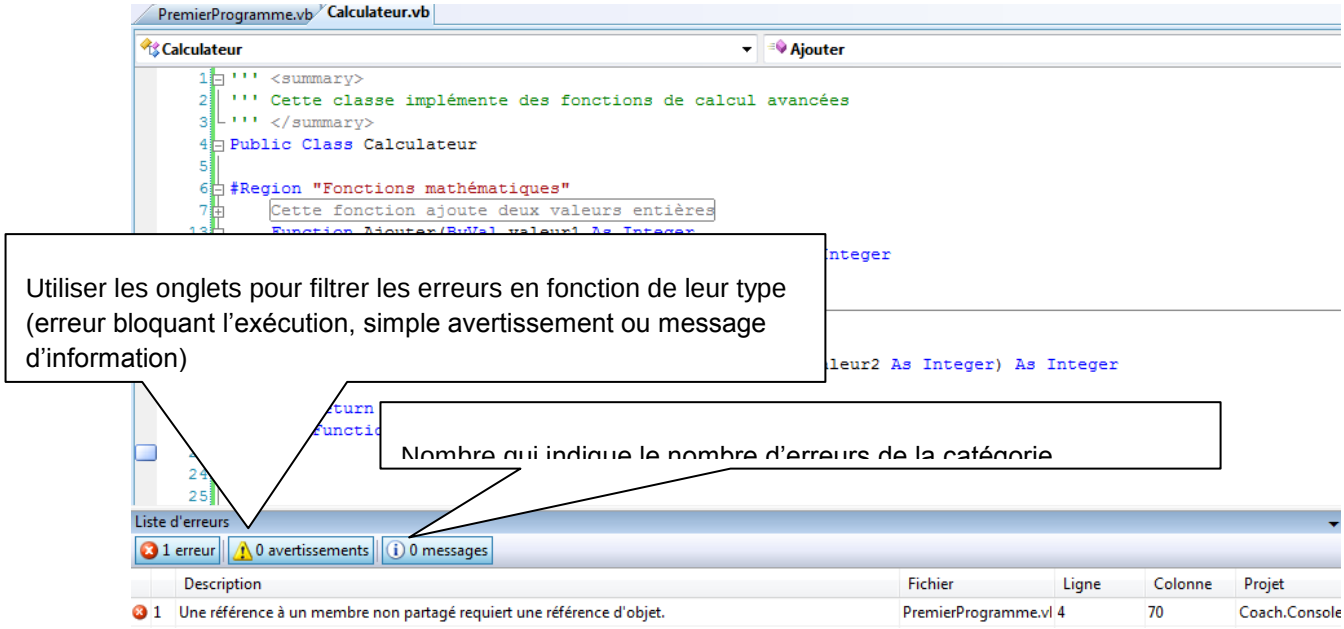
Un bon réflexe est de corriger les problèmes les uns après les autres si vous voulez exploiter ces fonctionnalités d'aide à la saisie. En effet, si vous ne rajoutez pas le mot clé **Shared** dans la déclaration de la fonction, la ligne d'appel de la fonction dans le fichier **PremierProgramme.vb** reste en erreur donc Visual Studio ne peut pas la traiter dans le renommage que vous lui demandez.

A l'inverse, vous pouvez utiliser ce mécanisme comme un feu orange. Typiquement si l'IntelliSense ne se déclenche pas lorsque vous tapez un point pour accéder à la liste des membres d'un type, c'est qu'il y a un souci, probablement à la ligne précédente, qui empêche Visual Studio d'analyser votre frappe !

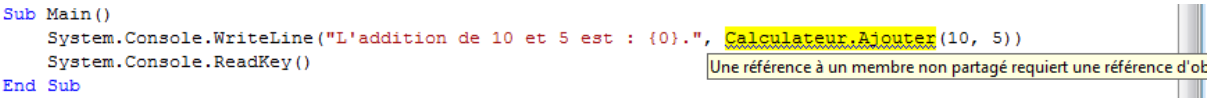
- Voyons comment se comporte maintenant la génération de la solution. Supprimez le mot clé **Shared** dans la définition de la fonction **Ajouter** puis enregistrez vos changements.
- Lancez la génération en même temps que l'exécution via la combinaison de touches **CTRL+F5**.
- Un message d'erreur apparaît vous donnant la possibilité de démarrer quand même l'application sur la base de la dernière génération réussie. Répondez **Non** (option par défaut) à la question posée pour stopper le processus de lancement et corriger les erreurs :



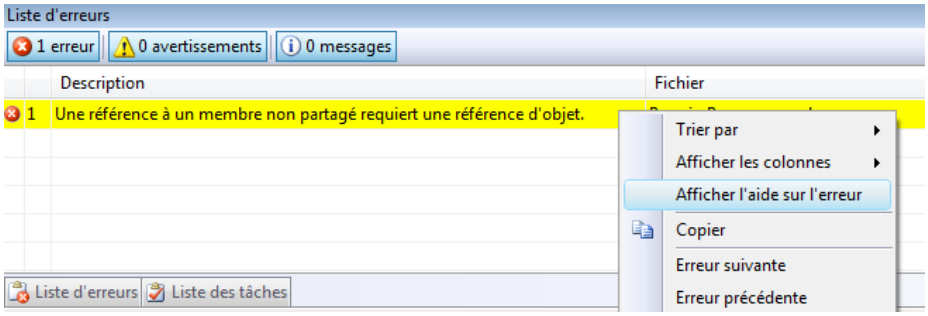
- Visual Studio affiche automatiquement la fenêtre **Liste d'erreurs** avec les erreurs trouvées :



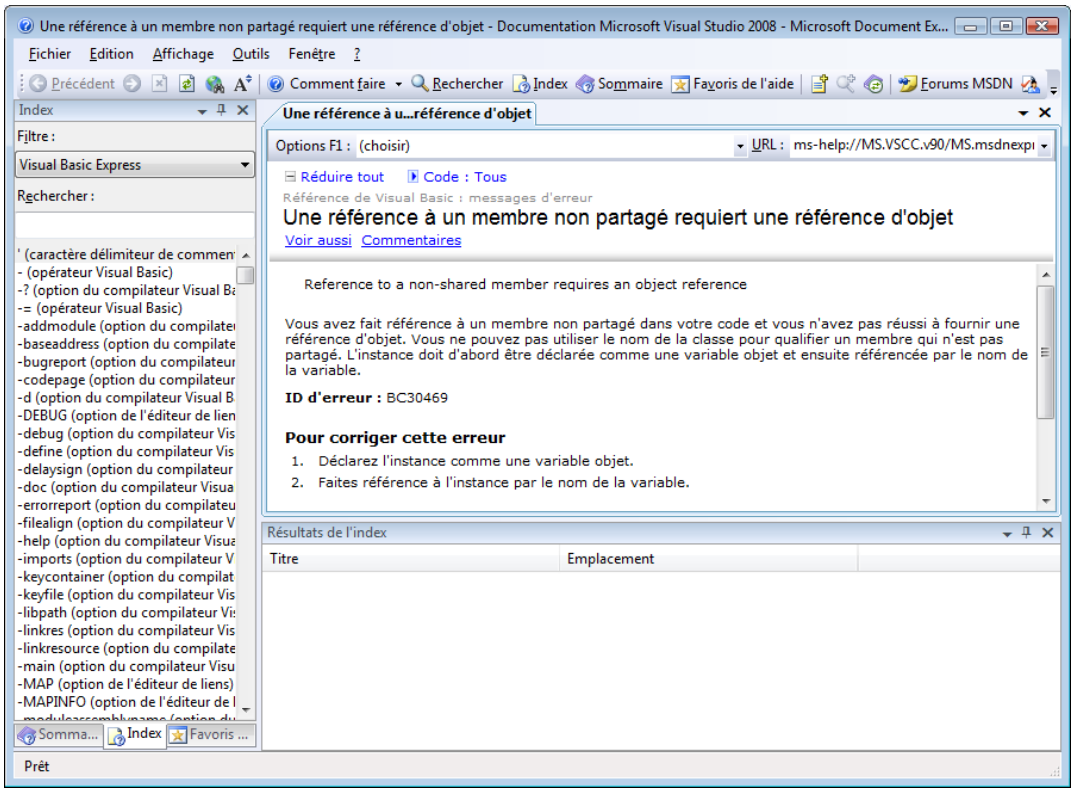
- Consultez la description de l'erreur, le fichier concerné et la position donnée par les numéros de ligne et colonne dans le code.
- Double cliquez sur la ligne d'erreur. Visual Studio ouvre le fichier correspondant et surligne le code posant le problème.



Vous pouvez aussi bénéficier d'une aide pour corriger le problème en faisant un clic droit directement sur la ligne d'erreur dans la Liste d'erreurs > **Affichez l'aide sur l'erreur** :



Selon que vous travaillez sur la base de l'aide en ligne ou de l'aide installée localement sur votre poste, vous obtenez une page d'aide du type :



- Corrigez l'erreur puis régénérer la solution.

DÉBOGUER LE CODE

Maintenant que la solution s'est générée avec succès, il vous reste à corriger les erreurs qui ne sont pas détectées par le processus de génération, à savoir les erreurs de logique qui provoquent des disfonctionnements de l'application.

Visual Studio intègre des outils de débogage puissants qui vous permettent de suspendre l'exécution du programme afin d'examiner le code.

A la fin de cet exercice, vous saurez :

- Contrôler l'exécution du code avec le débogueur de Visual Studio,
- Différencier les principales fenêtres du débogueur de Visual Studio.

Déroulement de l'exercice :

1. Préparez le débogage du programme :

Que signifie **contrôler l'exécution du code** ?

Le principe consiste à demander au débogueur de s'arrêter à un endroit précis dans le code afin d'examiner celui-ci. Ensuite vous pouvez poursuivre l'exécution du code normalement ou bien demander une exécution ligne par ligne de façon à pouvoir examiner chaque instruction de code.

- Editez le fichier de code **PremierProgramme.vb**.

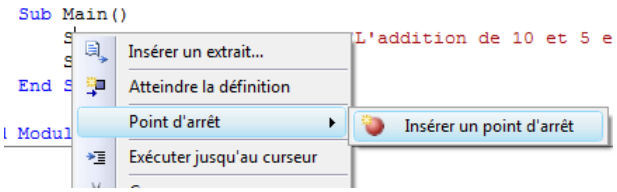
Supposons que l'affichage du résultat dans la console de l'application ne soit pas cohérent. Il faut donc demander l'arrêt de l'exécution du programme au niveau de l'appel de la méthode **WriteLine** pour vérifier ce qui se passe.

- Positionnez un point d'arrêt sur la ligne d'appel à la méthode **WriteLine** en cliquant dans la marge des indicateurs de Visual Studio.

```
1  Module PremierProgramme
2
3      Sub Main()
4          System.Console.WriteLine("L'addition de 10 et 5 est : {0}.", Calculateur.Ajouter(10, 5))
5          System.Console.ReadKey()
6      End Sub
7
8  End Module
```

La demande d'arrêt de l'exécution peut se faire également :

- En positionnant le curseur n'importe où sur la ligne concernée et en pressant la touche **F9** (Une seconde pression sur la touche F9 supprime le point d'arrêt et ainsi de suite).
- En faisant un clic droit sur la ligne concernée > **Point d'arrêt** > **Insérer un point d'arrêt**.



- En codant directement dans le code l'instruction **Stop**, qui est spécifique au langage VB. Le résultat est le même que le point d'arrêt.

Code VB

```
…
   Sub Main()
      Stop
      System.Console.WriteLine("L'addition de 10 et 5 est : {0}.",  _
                                    Calculateur.Ajouter(10, 5))
      System.Console.ReadKey()
   End Sub
…
```

Attention ! En ajoutant ce type d'instruction dans le code, vous risquez de l'oublier au moment du déploiement du projet ☹ alors qu'un point d'arrêt est mémorisé dans la solution Visual Studio mais n'a pas d'effet une fois le projet déployé. Une solution consiste à utiliser la *compilation conditionnelle* qui fournit des directives que vous pouvez positionner dans le code pour éviter que des blocs de code soient compilés dans un contexte autre que le débogage. Cela donnerait :

Code VB

```
…
   Sub Main()
      #If DEBUG Then
            Stop
      #Else
            'On ne s'arrête pas
      #End If
```

```
        System.Console.WriteLine("L'addition de 10 et 5 est : {0}.", _
                                 Calculateur.Ajouter(10, 5))
        System.Console.ReadKey()
    End Sub
…
```

Pour en savoir plus sur la compilation conditionnelle :

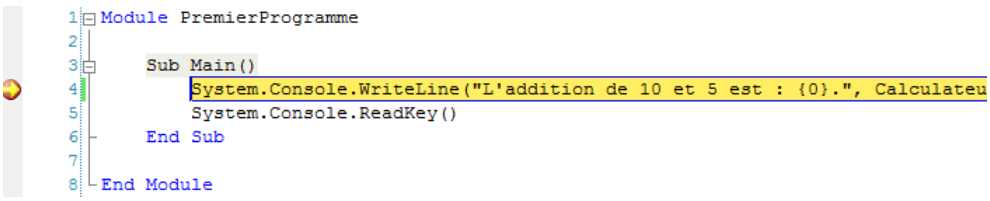
http://msdn2.microsoft.com/fr-fr/library/x435tkbk.aspx

Pour en savoir plus sur la mise en place de points d'arrêts, notamment si vous travaillez avec d'autres éditions de Visual Studio que les éditions Express, pour lesquelles il existe d'autres possibilités très intéressantes :

http://msdn2.microsoft.com/fr-fr/library/ktf38f66.aspx

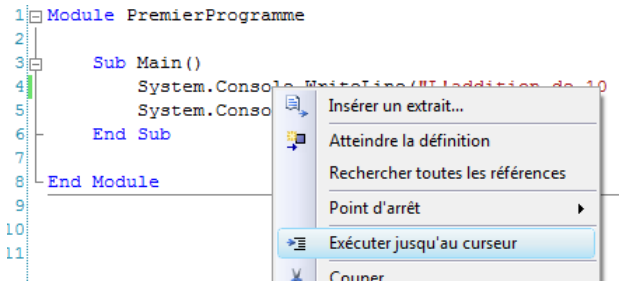
2. Exécutez l'application en mode débogage :

   - Lancez l'exécution de l'application en mode débogage avec la touche **F5** (ou le menu **Déboguer** > **Démarrer le débogage** ou l'icône dans la barre d'outils standard).
   - L'application s'exécute jusqu'au premier point d'arrêt que le runtime rencontre dans le code.



La flèche dans la marge des indicateurs, indique l'instruction en cours c'est-à-dire la *prochaine ligne à exécuter*.

Si vous n'avez qu'un seul point d'arrêt dans toute l'application, une alternative encore plus rapide consiste à faire un clic droit sur la ligne de code sur laquelle vous demandez l'arrêt de l'exécution puis à cliquer **Exécuter jusqu'au curseur**.



Cette option fait d'une pierre deux coups en lançant l'exécution du programme en mode débogage et en arrêtant, dans la foulée, le pointeur d'exécution sur la ligne spécifiée.

Est-ce qu'on pourrait démarrer l'exécution pas à pas du code dès le début du programme à l'appel de la procédure **Main** ?

Oui bien sûr, en lançant l'exécution avec la touche **F8** (ou le menu **Déboguer** > **Pas à pas détaillé**).

- A partir de là, vous pouvez exécuter pas à pas chaque ligne d'instruction en prenant le temps d'examiner le code correspondant de façon à détecter la source du disfonctionnement observé, et appuyez la touche F5 à nouveau pour sauter jusqu'au prochain point d'arrêt.

La ligne qui nous intéresse comporte un appel à la fonction **Ajouter** du projet de librairie **Coach.Calculateur**. Aussi deux stratégies de pas à pas s'offre à vous :

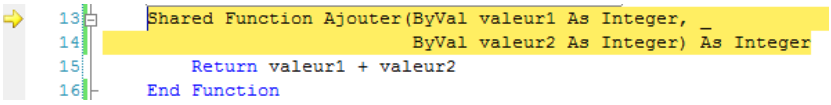
- Soit vous voulez que Visual Studio vous emmène dans le détail de la fonction **Ajouter** de façon à examiner ligne à ligne également ce qui s'y passe. Dans ce cas, utilisez la commande de *pas à pas détaillé* : touche **F8** ou menu **Déboguer** > **Pas à pas détaillé**.

- Soit vous ne voulez pas connaître le détail de la fonction et préférez rester dans le contexte de la procédure en cours. Cela revient à exécuter l'appel de la fonction **Ajouter** mais à passer le pointeur d'exécution directement sur la ligne suivante de la fonction en cours. Utilisez la commande de *pas à pas principal* : touches **Maj+F8** ou menu **Déboguer** > **Pas à pas principal**.

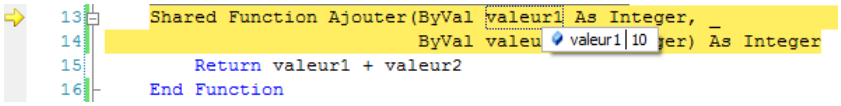
Pour en savoir plus sur les modes d'exécution pas à pas :

http://msdn2.microsoft.com/fr-fr/library/ek13f001.aspx

- Cliquez **F8** pour rentrer dans la fonction **Ajouter** du **Calculateur**.

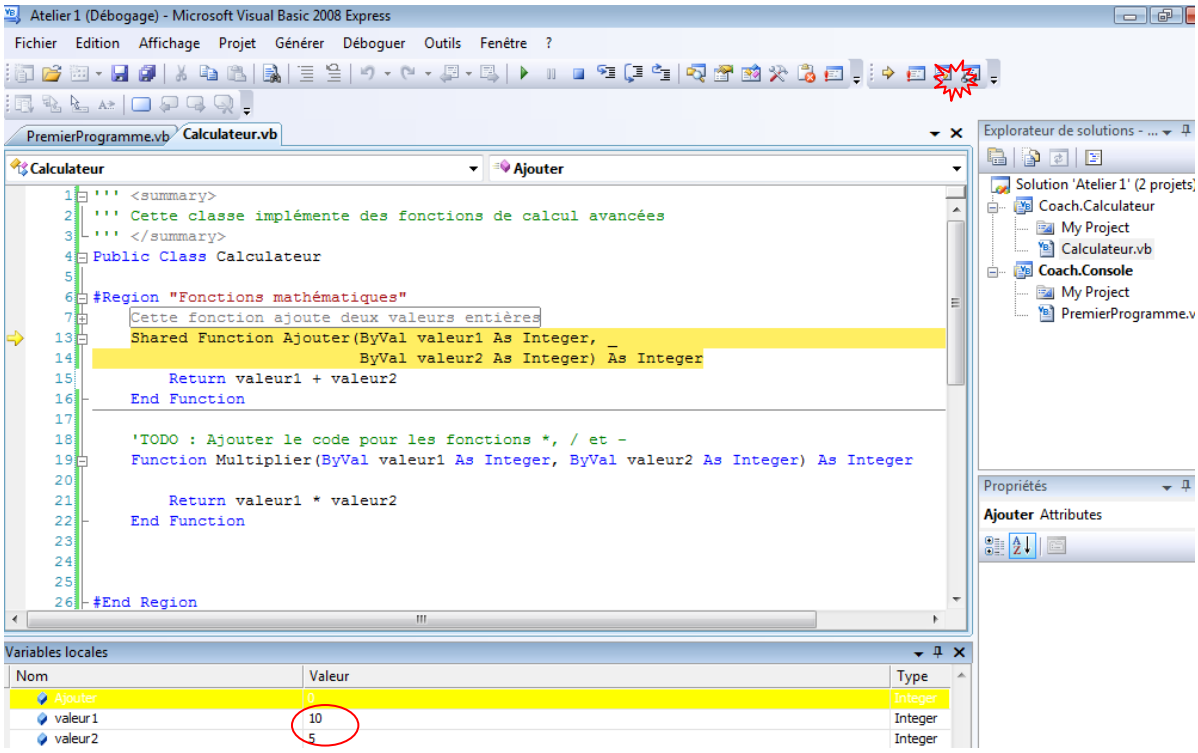


- Stoppez la souris sur le premier paramètre. Une fenêtre s'affiche avec la valeur en cours de la variable.

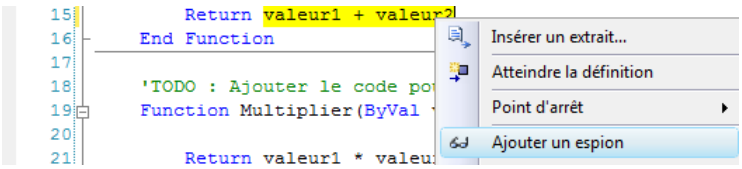


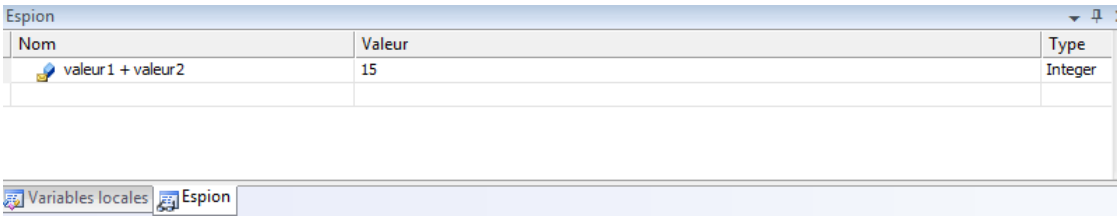
Cette variable étant locale à la procédure en cours, vous pouvez également l'observer dans la fenêtre **Variables locales** qui s'affiche en cliquant l'icône de la **Barre d'outils Déboguer** (qui

apparaît automatiquement au lancement du mode débogage sur la droite de la barre standard).



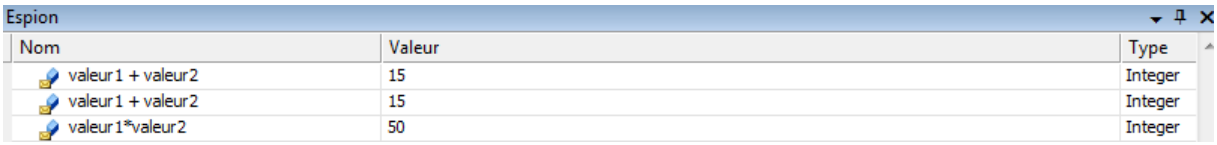
Si l'information qui vous intéresse n'est pas disponible au travers de cette fenêtre, utilisez la fenêtre **Espion** du débogueur pour l'observer (icône dans la barre d'outils de débogage). Par exemple, pour avoir une vue sur le résultat du calcul **valeur1 + valeur2**, procédez comme suit : sélectionnez le code avec la souris puis faites un clic droit > **Ajouter un espion**.



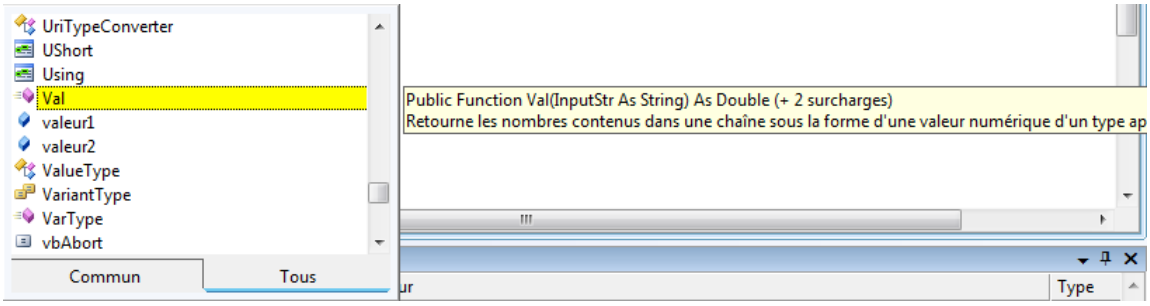


Un simple glisser/déplacer du calcul **valeur1 + valeur2** sur la surface de la fenêtre **Espion** aurait eu le même effet.

De même que vous pouvez saisir directement dans la fenêtre **Espion** (par exemple pour évaluer **valeur1 multiplié par valeur2**).



Avez-vous remarqué que l'IntelliSense fonctionne aussi dans ce type de fenêtre ? Tapez par exemple **val** puis la combinaison de touches **CTRL+Espace** et la liste de suggestions apparaît :



Pour exécuter des instructions plus complexes, le débogueur dispose de deux autres fenêtres équipées également de l'IntelliSense : la **fenêtre d'exécution** (pour évaluer des expressions) et la **fenêtre de commande** (pour exécuter les commandes de menu de

Visual Studio par exemple, telles que l'ouverture d'un fichier).

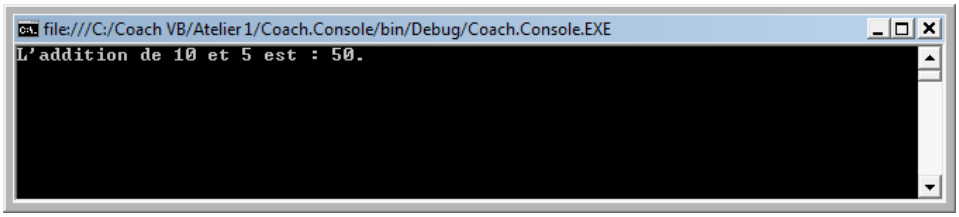
Pour en savoir plus sur ces deux fenêtres :

http://msdn2.microsoft.com/fr-fr/library/kcc7tke7.aspx

- Supposons que vous avez identifié que le problème vient de l'opération d'ajout. Vous pouvez directement modifier le calcul et par exemple entrer : valeur1 ***** valeur2 pour faire une multiplication plutôt qu'une addition.

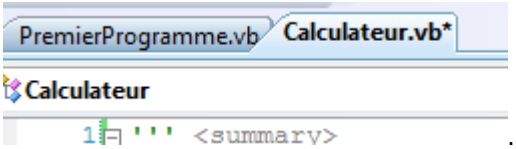
```
13      Shared Function Ajouter(ByVal valeur1 As Integer, _
14                              ByVal valeur2 As Integer) As Integer
15          Return valeur1 * valeur2
16      End Function
```

- Poursuivez l'exécution du code avec la touche **F5**. Le résultat est **50** au lieu de **15**.



- Lorsque vous refermer la console ou si vous cliquez l'icône dans la barre d'outils standard de Visual Studio, le programme sort du mode débogage et stoppe son exécution.

Notez que le fichier **Calculateur.vb** apparaît bien comme devant être sauvegardé suite aux modifications que vous avez faites.



Voilà donc un petit aperçu des nombreuses fonctionnalités du débogueur de Visual Studio. Cela vaut vraiment le coup de creuser un peu la question si vous voulez déboguer vos programmes avec un maximum d'atouts en main (encore que vous et moi savons que nous développons sans jamais faire de bogue). Visual Studio propose également une fenêtre de consultations de la pile des appels, une fenêtre de sortie dans laquelle vous pouvez écrire des informations de débogage etc…

Pour creuser la question et apprendre par exemple à déboguer une application web ou pour déboguer vos procédures stockée rendez-vous sur : http://msdn2.microsoft.com/fr-fr/library/sc65sadd.aspx

ET PLUS ENCORE

Bon, clairement Visual Studio sait encore faire bien d'autres choses ☺.

Nous verrons dans le prochain atelier comment *dessiner une interface Windows* plutôt que de travailler sur un simple projet d'application console.

Nous aurons également l'occasion dans ce tutorial de jouer un peu avec les *outils d'accès aux données* de Visual Studio pour gérer une base de données SQL Server.

Et si vous voulez vous préparer au développement en entreprise de solutions professionnelles complexes et apprendre à travailler en équipe, suivez le coach VSTS qui vous guide dans l'apprentissage de VSTS (Visual Studio Team System). Vous verrez que dans cette édition professionnelle, Visual Studio fournit aussi des outils de test, des outils d'analyse de la performance des applications, un contrôle de code source etc…

Créer sa première application VB.NET

http://msdn.microsoft.com/vbasic

SOMMAIRE

1 INTRODUCTION 3

1.1 CONTEXTE FONCTIONNEL 3

1.2 CONTEXTE TECHNIQUE 6

2 CRÉER LE PROJET ET LA FENÊTRE PRINCIPALE 7

2.1 CRÉER LA SOLUTION DE PROJET 8

2.2 CONTRÔLER LE DÉMARRAGE DE L'APPLICATION 12

2.3 COMPRENDRE LE FONCTIONNEMENT D'UN FORMULAIRE 26

2.4 CONTRÔLER L'AFFICHAGE ET L'ARRÊT DE L'APPLICATION 40

3 TRAVAILLER À BASE DE CONTRÔLES (COMPOSANTS) 42

3.1 CONFIGURER LES CARACTÉRISTIQUES DE LA FENÊTRE PRINCIPALE 44

3.2 CONSTRUIRE LE MENU DE L'APPLICATION 48

3.3 CODER LA FERMETURE DE L'APPLICATION 57

3.4 AFFICHER L'APPLICATION DANS LA ZONE DE NOTIFICATION 73

4 POUR ALLER PLUS LOIN… 94

INTRODUCTION

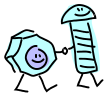
CONTEXTE FONCTIONNEL

Rappel du contexte fonctionnel du tutorial du coach VB

L'objectif du tutorial du Coach VB est d'accompagner les développeurs à la découverte et la prise en main du langage Visual Basic (VB) pour la construction d'applications avec une approche orientée objet.

Pour rappel, vous pouvez repérer facilement deux caractéristiques importantes du langage à l'aide des logos suivants en marge :

Ce logo marque une fonctionnalité de VB ou de Visual Studio qui permet de développer vite (et juste ☺).



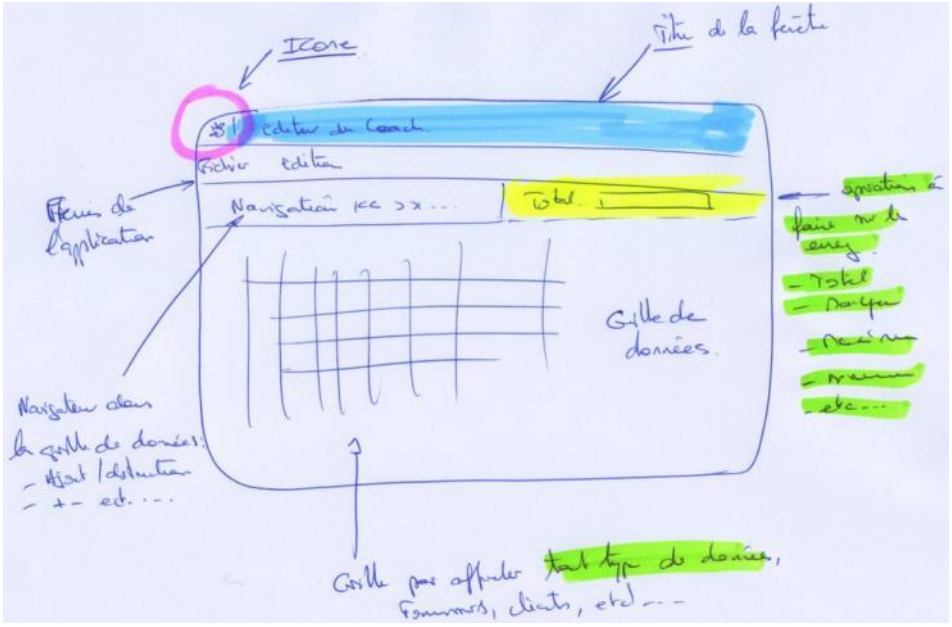
Ce logo met en évidence une caractéristique de la programmation orientée objet.

Contexte fonctionnel du deuxième atelier

Ce deuxième atelier décrit la création d'une première application Visual Basic de type Windows.

L'objectif est de construire une application présentant une fenêtre avec une grille de travail sur des données. Nous l'appellerons **Editeur du coach VB**.

Au mieux vous devez disposer d'un bout de feuille issu d'une réunion qui décrit l'interface de l'application que vous devez réaliser, sur laquelle a abouti l'analyse fonctionnelle. Cela ressemblerait à ceci :

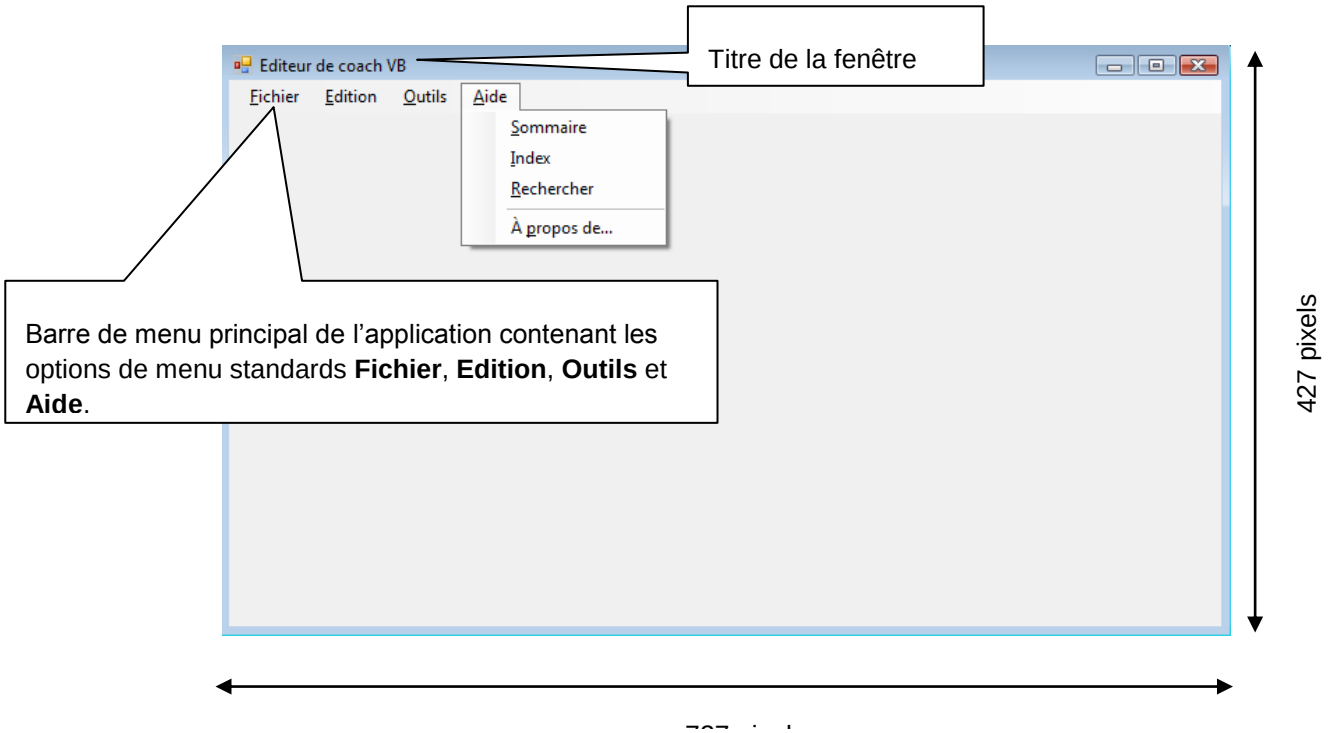


Ca va encore être simple ☺…

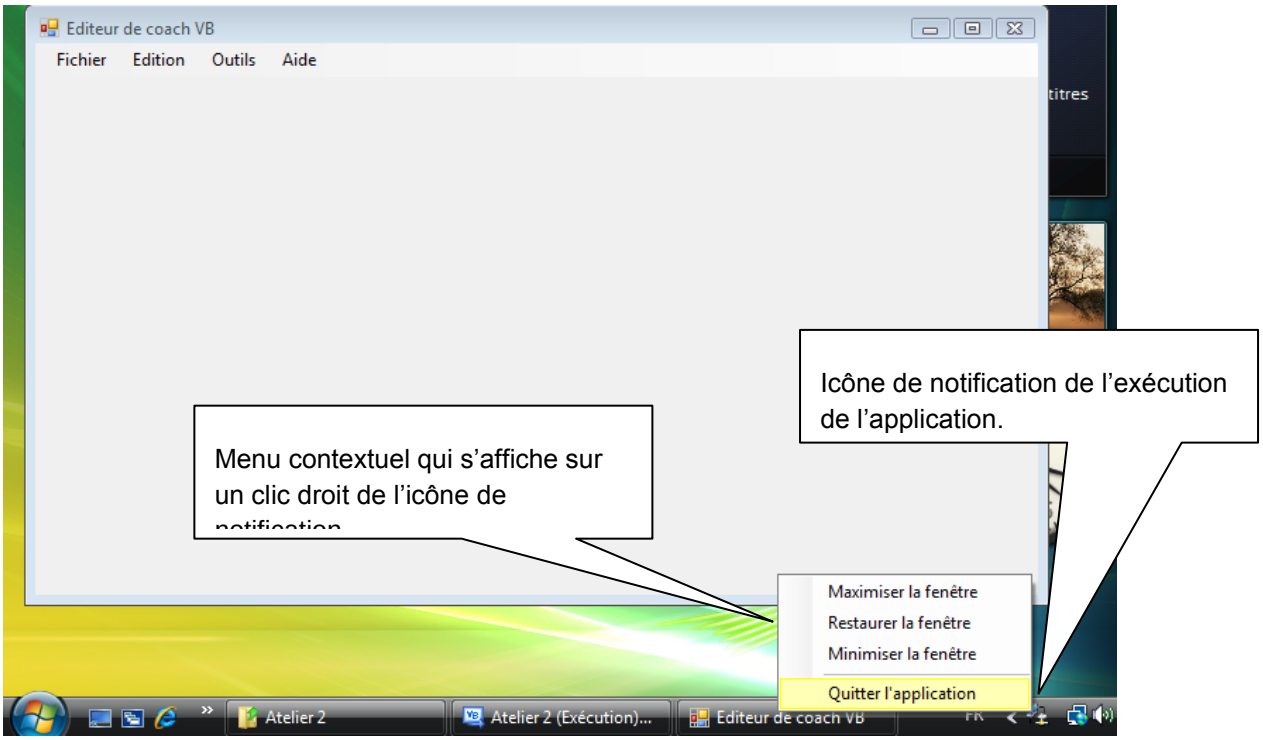
Pour l'instant dans cet atelier, nous nous attacherons à construire la « charpente » de l'application sans nous préoccuper de l'affichage des données.

Au lancement de l'application, nous allons afficher un cours instant un écran de démarrage donnant le titre, la version et le propriétaire de l'application :

Ensuite l'application devra afficher le formulaire principal de l'application avec la taille définie et une barre de menu contenant les éléments standards que l'on trouve dans les applications Windows.



Enfin, nous mettrons en évidence l'exécution de l'application par une petite icône dans la zone de notification d'état de la barre des tâches de Windows, sur laquelle nous accrocherons un menu contextuel pour permettre à l'utilisateur de fermer l'application.



CONTEXTE TECHNIQUE

Dans cet atelier nous allons mettre de côté (momentanément car nous traiterons le sujet dans les prochains ateliers de ce tutorial) le traitement des données pour nous concentrer essentiellement sur la mise en place de la structure de base d'une application Windows écrite en Visual Basic.

Nous allons créer le formulaire principal, configurer son mode de démarrage ainsi que la façon dont s'arrête l'application. Nous en profiterons pour aborder les principes de programmation essentiels de la programmation dite *évènementielle*.

A la fin de cet atelier, vous saurez comment :

- Créer une application simple à base de formulaire,
- Développer des gestionnaires d'évènement associés à différents types d'évènement,
- Concevoir un formulaire en utilisant des contrôles et des composants Windows Form,
- Utiliser la boîte d'outils et la fenêtre de propriétés de Visual Studio,

- Programmer la zone de notification de Windows,
- Utiliser des classes partielles.

La solution de cet atelier est disponible dans le répertoire **..\Atelier 2\Solution**. Les fichiers utiles, auxquels font référence les exercices sont disponibles dans le répertoire **..Atelier 2\Fichiers utiles**.

CREER LE PROJET ET LA FENETRE PRINCIPALE

Dans cet exercice, vous allez apprendre à :

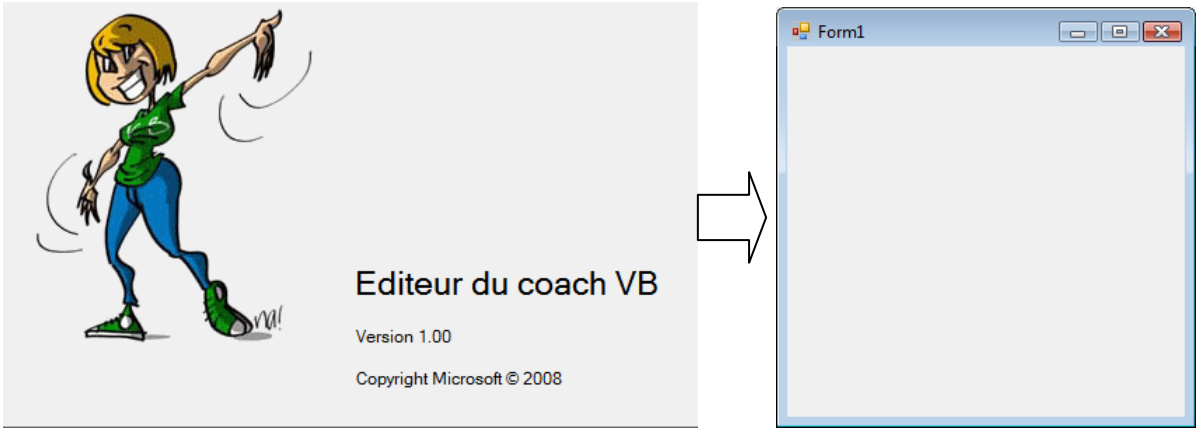
- Créer un projet de type Application Windows Forms,
- Créer un écran de démarrage,
- Utiliser le Concepteur de projets de Visual Studio pour configurer le démarrage et l'arrêt d'une application,
- Visualiser les différents fichiers qui constituent un formulaire Windows Form,
- Utiliser les classes partielles.

Objectif

L'objectif de ce premier exercice est de démarrer un premier projet de développement de type *Application Windows Forms* et d'en comprendre les principes de base.

Contexte fonctionnel

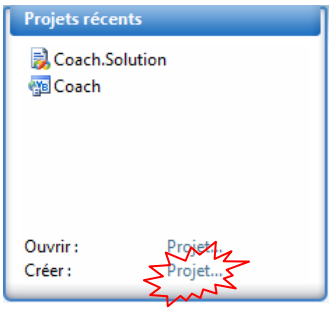
Cet dans ce premier exercice que nous allons créer un écran de démarrage qui s'affichera quelques secondes seulement avant l'affichage du formulaire principal de l'application :



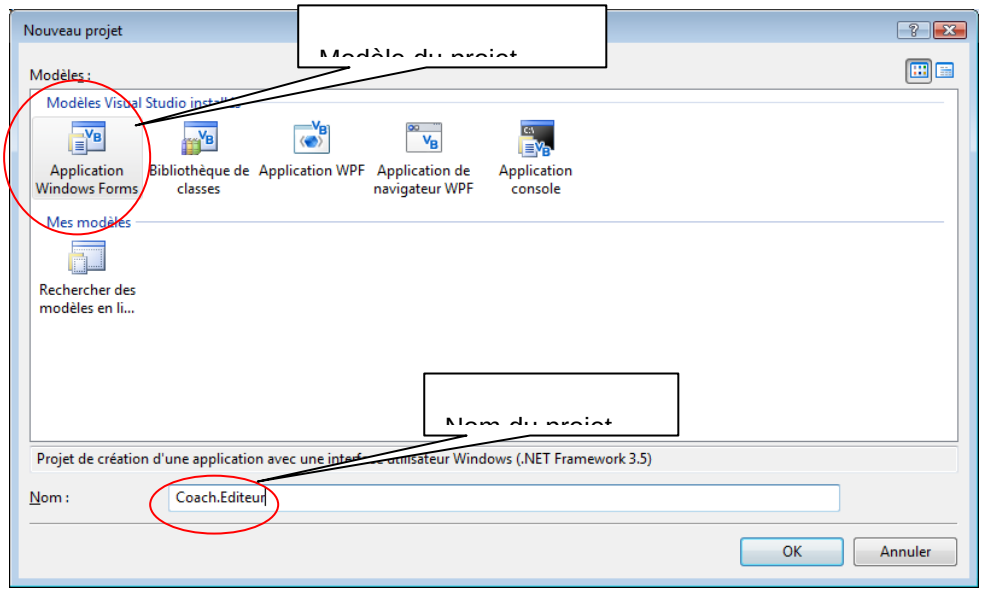
CREER LA SOLUTION DE PROJET

Déroulement de l'exercice :

1. Créez une solution de développement nommée Atelier 2 avec un projet de type Application Windows :
    - Lancez Visual Studio à partir du menu **Démarrer > Tous les programmes > Microsoft Visual Basic 2008 Express Edition**.
    - Créez un nouveau projet depuis l'option **Créer :** > **Projet…** de la page de démarrage ou à partir du menu **Fichier** > **Nouveau projet...**



- Dans la fenêtre **Nouveau projet**, sélectionnez le modèle de projet **Application Windows Forms** et indiquez **Coach.Editeur** comme nom de projet.



Ce modèle de projet fait référence aux *Windows Forms*. Savez-vous en quoi consiste cette technologie ?

Dans *Windows Forms*, il y a :

- le mot **Windows** qui fait référence aux applications de bureau riches et interactives que l'on déploie sur un poste client,
- et le mot **Forms** qui fait référence à la notion de formulaire.

Qu'est-ce qu'on entend par *formulaire* ?

Un formulaire est une fenêtre Windows dans laquelle l'utilisateur peut consulter et saisir des informations de manière simple et interactive.

Donc très succinctement, développer une application Windows Forms revient à dessiner un ou plusieurs formulaires puis à coder des traitements pour répondre aux différentes actions réalisées par l'utilisateur avec la souris ou le clavier sur ces formulaires.

Pour vous aider dans cette tâche, le Framework .NET vous fournit toute la « mécanique de base » de fonctionnement de ce type de formulaire de façon à ce que vous perdiez le moins de temps possible à les dessiner et surtout pour que vous puissiez vous concentrer sur la logique métier de votre application. Ce sont les **Windows Forms** comprenant de multiples classes, un Concepteur de formulaire et tout ce qui peut vous aider dans les tâches de programmation courantes de ce type d'application.

Pour avoir une vue d'ensemble des Windows Forms :

http://msdn.microsoft.com/fr-fr/library/8bxxy49h.aspx

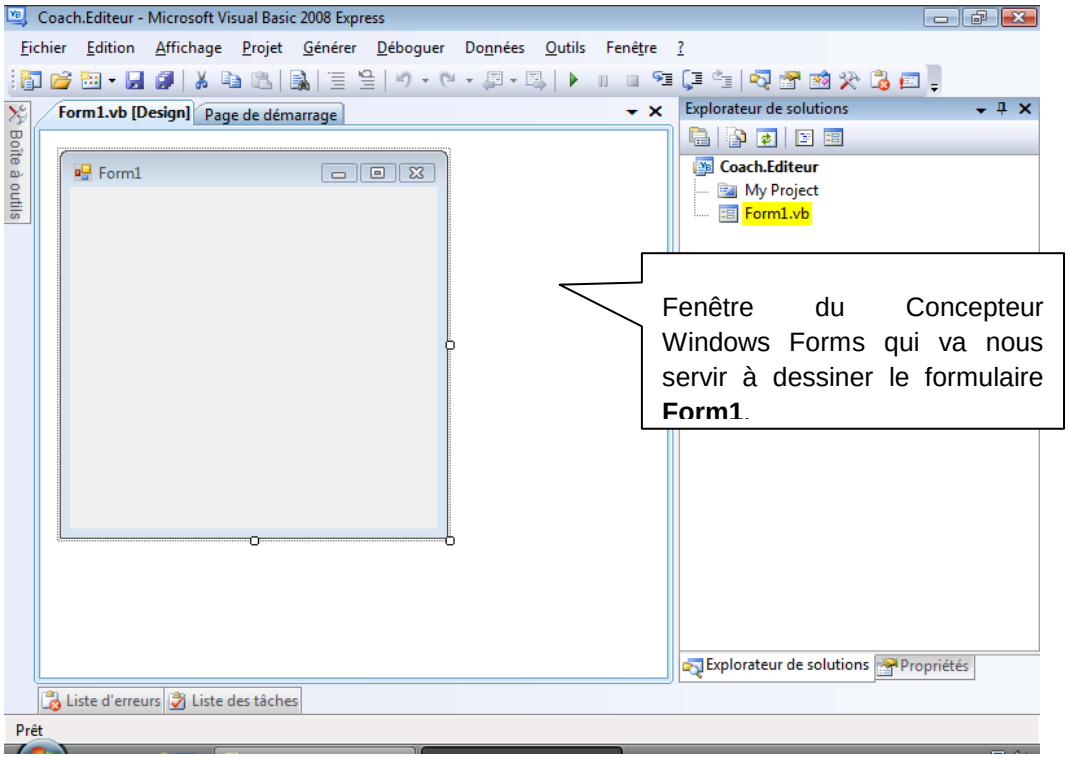
A ne pas confondre avec les Web Forms qui permettent de développer un autre type de formulaire pour les applications web. Pour comparer les deux technologies, rendez-vous à l'adresse suivante :

http://msdn.microsoft.com/fr-fr/library/5t6z562c.aspx

- Validez par **OK**.

Que contient ce modèle de projet ?

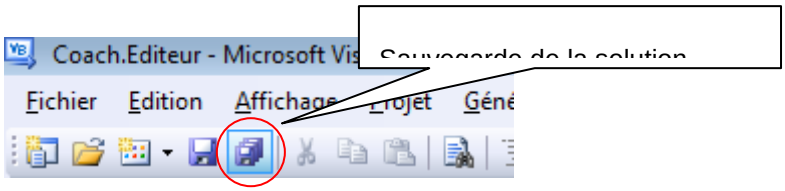
Un projet généré sur la base du modèle **Application Windows Forms** comprend un formulaire vide nommé **Form1** par défaut. Vous pouvez voir dans la surface de travail sa représentation visuelle dans l'onglet **Form1.vb [Design]** généré par le **Concepteur Windows Forms** de Visual Studio.



Pour en savoir plus sur l'outil de design de formulaires clients **Concepteur Windows Forms** :

http://msdn.microsoft.com/fr-fr/library/e06hs424.aspx

- Sauvegardez tout de suite le projet et créer une solution en cliquant sur l'icône dans la barre d'outils standard de Visual Studio.

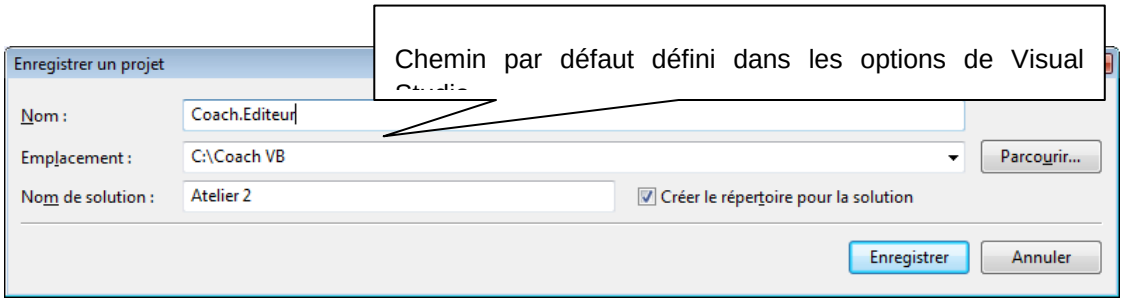


- Dans la boîte de dialogue **Enregistrer un projet**, indiquez votre répertoire de travail.

Par défaut vous devez retrouver le chemin que vous avez spécifié dans la boîte de dialogue d'options de Visual Studio dans l'exercice 3.1 de l'atelier 1 du tutorial.

C'est aussi à ce moment là que vous pouvez demander la création d'une solution en cochant la case **Créer le répertoire pour la solution**.

- Cochez la case **Créer le répertoire pour la solution** et saisissez un nom pour la solution par exemple : **Atelier 2**.



- Cliquez sur **Enregistrer**.

CONTRÔLER LE DÉMARRAGE DE L'APPLICATION

Dans ce type de projet, où est l'indispensable procédure *Main* qui constitue le point d'entrée du programme ?

C'est vrai que nous avons dit à l'exercice 2 de l'atelier 1 de ce tutorial que le Framework .NET appelle la procédure **Main** du programme lorsqu'il a chargé l'application. Elle constitue le point de départ de l'exécution.

Dans le cas d'une application basée sur des formulaires, en général il est surtout intéressant de démarrer l'exécution directement par l'affichage d'un formulaire du projet. En théorie, il faudrait donc écrire une procédure **Main** qui créé une instance du formulaire puis l'affiche.

Pour simplifier, le compilateur Visual Basic génère automatiquement cette procédure pour vous si bien que vous n'avez pas à vous soucier de l'écrire. En revanche, c'est à vous d'indiquer au compilateur quel est le formulaire du projet sur lequel vous souhaiter

démarrer.

Déroulement de l'exercice :

1. Configurez le formulaire de démarrage de l'application :

    - Dans l'**Explorateur de Solutions**, double cliquez **My Project** pour faire apparaître le Concepteur de projets de Visual Studio.

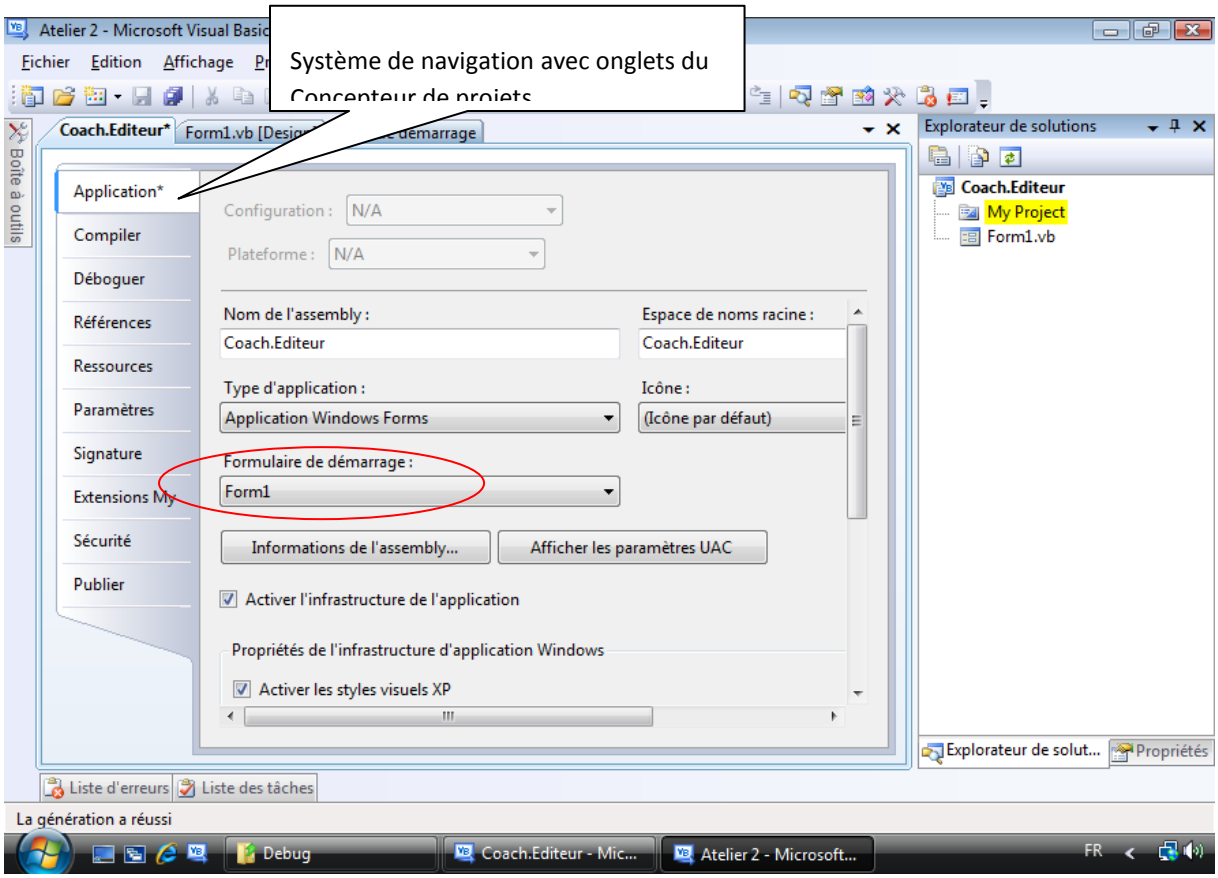
Pour rappel, cette fenêtre que nous avons déjà eu l'occasion d'afficher dans l'atelier précédent de ce tutorial, centralise l'ensemble des propriétés et paramètres de votre projet. Dans l'atelier 1, nous l'avons utilisé pour ajouter au projet Console Windows une référence à la dll du Calculateur ou pour configurer des options de compilation.

Vous pouvez également afficher le Concepteur de projets en faisant un clic droit à la racine du projet dans l'Explorateur de solutions > **Propriétés**.

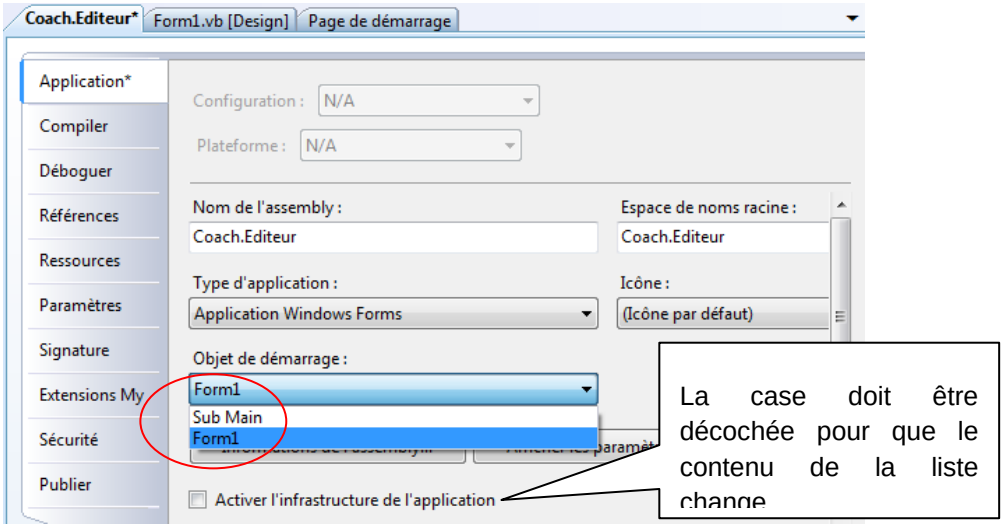
Pour en savoir plus sur le **Concepteur de projets** de Visual Studio :

http://msdn.microsoft.com/fr-fr/library/bb1aa8f1.aspx

    - Dans l'onglet **Application**, vérifiez que le formulaire de démarrage est correctement configuré sur **Form1**.



Notez que dans l'état actuel du projet qui ne comporte qu'un seul formulaire, il n'y a pas d'autre possibilité dans la liste des formulaires de démarrage. Mais si vous désactivez la case **Activer l'infrastructure de l'application**, une nouvelle option est disponible dans cette même liste qui s'intitule alors **Objet de démarrage**.



L'option supplémentaire **Sub Main** serait utile pour écrire une fonction de démarrage personnalisée à votre convenance. Celle-ci pourrait être définie dans une classe (moyennant le mot clé **Shared**) ou dans un module. Pour afficher le formulaire **Form1** à partir d'une telle méthode utiliser la méthode **Run()** de la classe **Application** :

Code VB

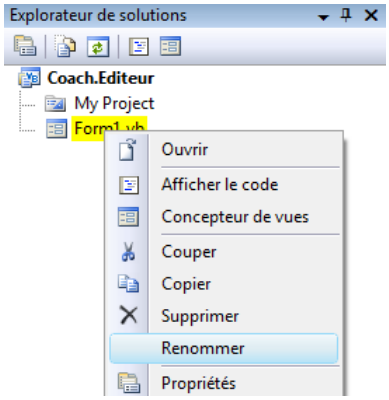
```
Sub Main()
    Application.Run(New Form1())
    'Coder ici les autres actions à exécuter au lancement de l'application
    …
End Sub
```

Pour en savoir plus sur la méthode **Run** de la classe **Application** :

http://msdn.microsoft.com/fr-fr/library/ms157902(VS.85).aspx

2. Renommez la fenêtre principale du projet pour qu'elle soit clairement reconnaissable comme étant le point d'entrée de l'application :

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

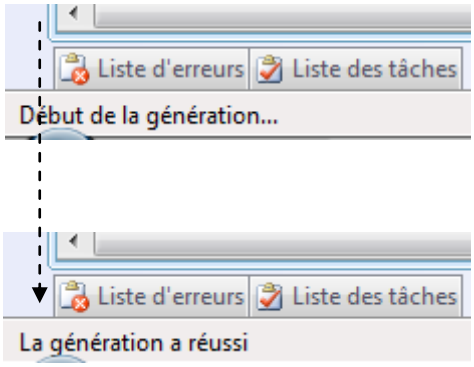


    - Changez le nom de **Form1.vb** en **Main.vb**.
    - Appuyez sur la touche **Entrée**.

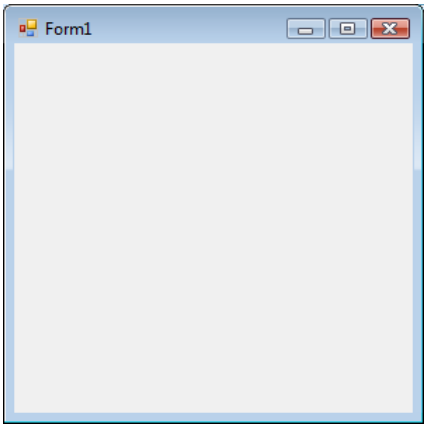
3. Testez le comportement de l'application en mode d'exécution :

    - Lancez l'application en utilisant le menu **Débogage**, ou la flèche dans la barre d'outils standard ou via le raccourci **F5**.

Visual Studio commence par générer l'application. Vous pouvez observer en bas à gauche de l'écran les messages relatifs au processus de génération :



Puis le formulaire **Main** s'affiche :



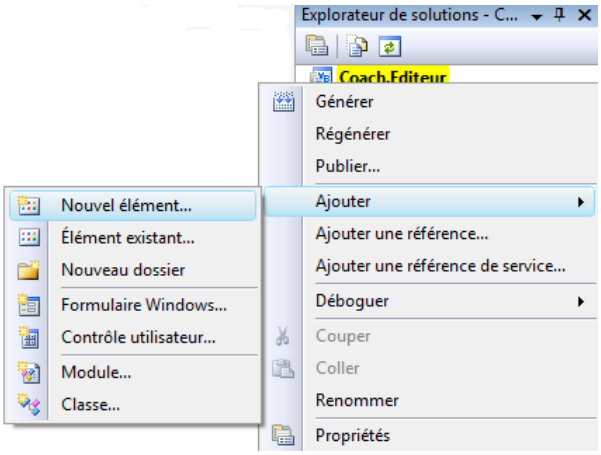
- Arrêter l'exécution de l'application en cliquant l'icône du formulaire.

Une alternative à la configuration de démarrage que nous venons de mettre en place consiste à afficher un petit écran de démarrage l'espace de quelques instants *avant* d'afficher le formulaire principal de l'application. C'est d'ailleurs comme cela que démarre Visual Studio.

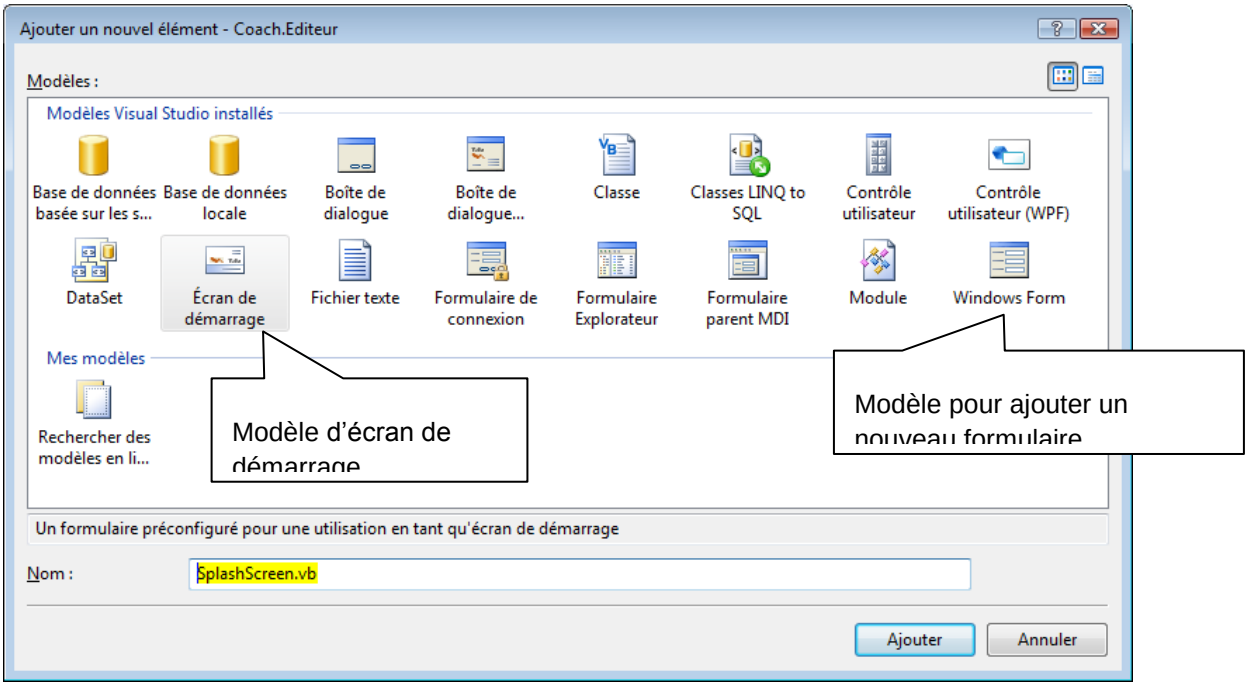
Le temps d'affichage de cet écran intermédiaire peut servir à couvrir le chargement ou l'initialisation de données, à ouvrir des fichiers etc…

4. Ajoutez un écran de démarrage au projet Coach.Editeur :

   - Dans l'**Explorateur de Solutions**, faites un **clic-droit** sur la racine du projet **Coach.Editeur** > **Ajouter** > **Nouvel élément…**.



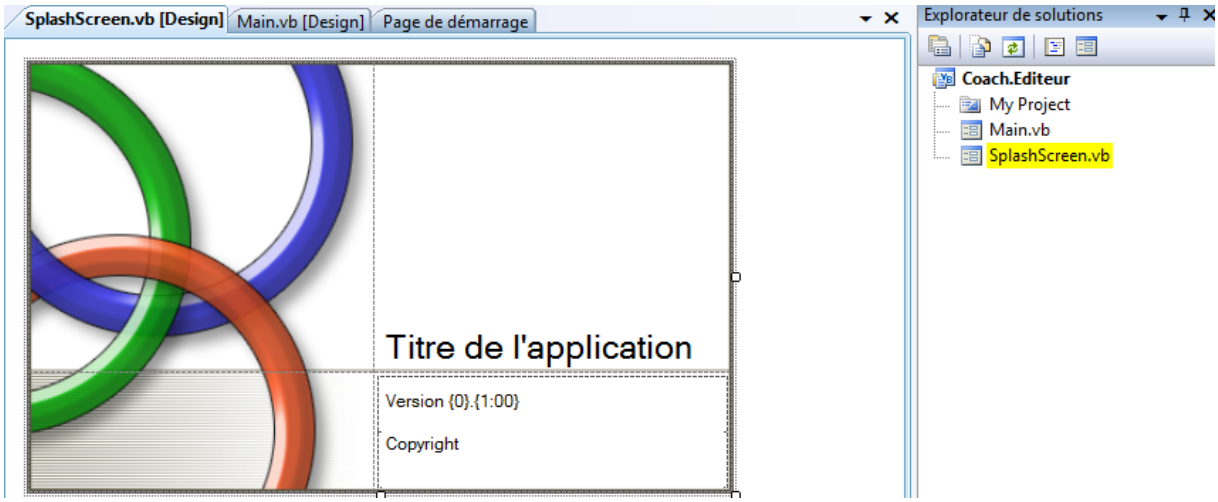
- Dans la fenêtre **Ajouter un nouvel élément**, sélectionnez le modèle **Ecran de démarrage**.
- Nommez le formulaire **SplashScreen.vb**.



Notez que le modèle **Windows Form** sur lequel est basé votre formulaire **Main** est proposé toute à droite de la fenêtre par Visual Studio. Mais ne perdez pas de vue qu'un projet Windows Forms peut contenir aussi beaucoup d'autres types d'éléments, ce qui explique le nombre de modèles que propose cette fenêtre !

Vous y trouvez par exemple un modèle de création de classe, dont nous verrons l'utilité par la suite dans ce tutorial ; et pourquoi pas aussi un modèle d'**Ecran de démarrage**, basé lui-même sur le modèle Windows Form… Il s'agit d'un formulaire dessiné par défaut avec les dimensions et les zones qui caractérisent ce type de fenêtre.

- Validez en cliquant **Ajouter**.



Vous obtenez en effet un écran constitué par défaut d'une image, d'une zone de titre de l'application et des informations de version et de copyright.

Comment faire pour afficher ce formulaire pendant une durée limitée au démarrage de l'application ?

Vous avez deux possibilités :

- La première, est de vous reposer sur Visual Studio qui, ne l'oublions pas,

s'attache à nous simplifier la vie le plus possible. Puisqu'il sait afficher un formulaire au démarrage via un simple paramétrage dans le Concepteur de projets, pourquoi ne serait-il pas capable de nous afficher aussi un écran intermédiaire l'espace de quelques secondes ?

Cette première approche est dite *déclarative* dans la mesure où nous n'avons qu'à *déclarer (*ou *paramétrer*) les objets et le comportement attendu à Visual Studio pour qu'il sache ce qu'il faut faire. C'est cette option que nous allons mettre en œuvre.

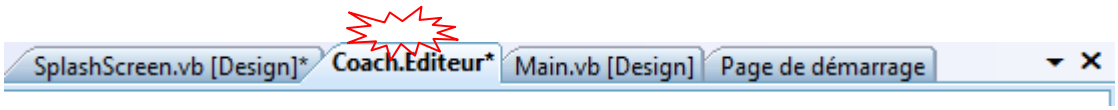
- Mais gardez toujours à l'esprit que tout ce que vous faites avec l'aide de Visual Studio, peut évidemment être fait sans lui, tout simplement en programmant directement les ordres équivalents par code.

  Cette autre approche *par code* est généralement un peu plus longue mais permet d'affiner bien davantage le résultat. Et grâce aux innombrables classes fournies par le .NET Framework, il n'est pas utile de tout réécrire de zéro ☺.

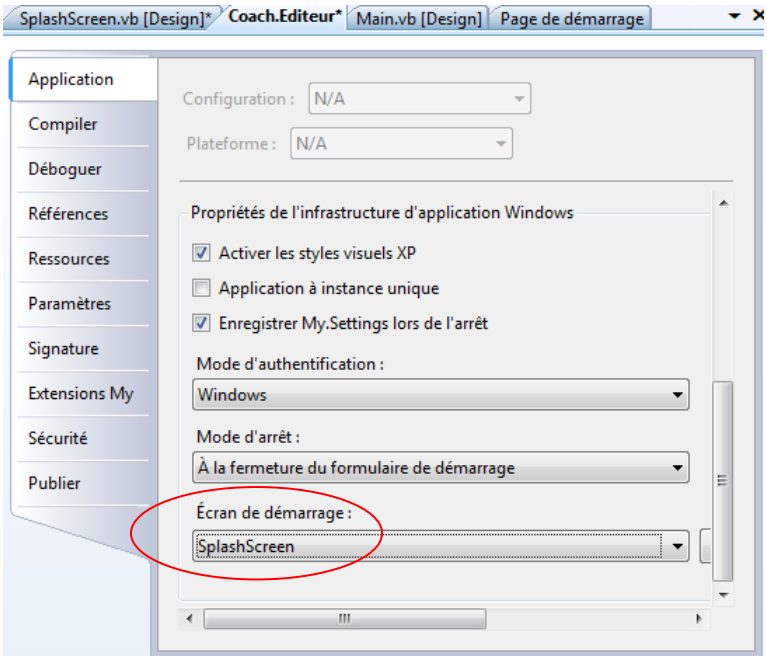
5. Associez le formulaire SplashScreen en tant qu'écran de démarrage du projet Coach.Editeur à l'aide du Concepteur de projets de Visual Studio :

   - Dans l'**Explorateur de Solutions**, double cliquez sur **My Project** pour afficher à nouveau le **Concepteur de projets**.

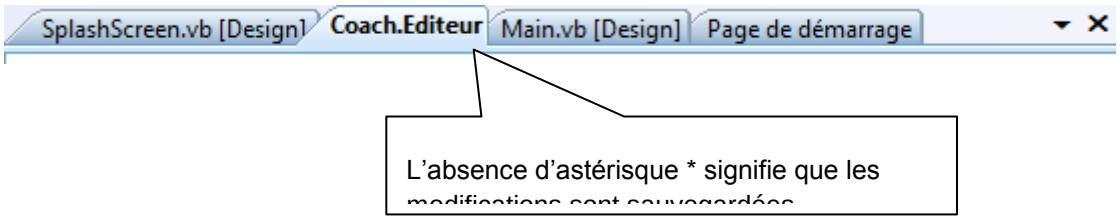
Vous pouvez aussi cliquer sur l'onglet **Coach.Editeur** sur la surface de travail si vous n'avez pas fermé le **Concepteur de projets** auparavant :



   - Dans l'onglet **Application**, sélectionnez le formulaire **SplashScreen** dans la liste de la propriété **Ecran de démarrage**.



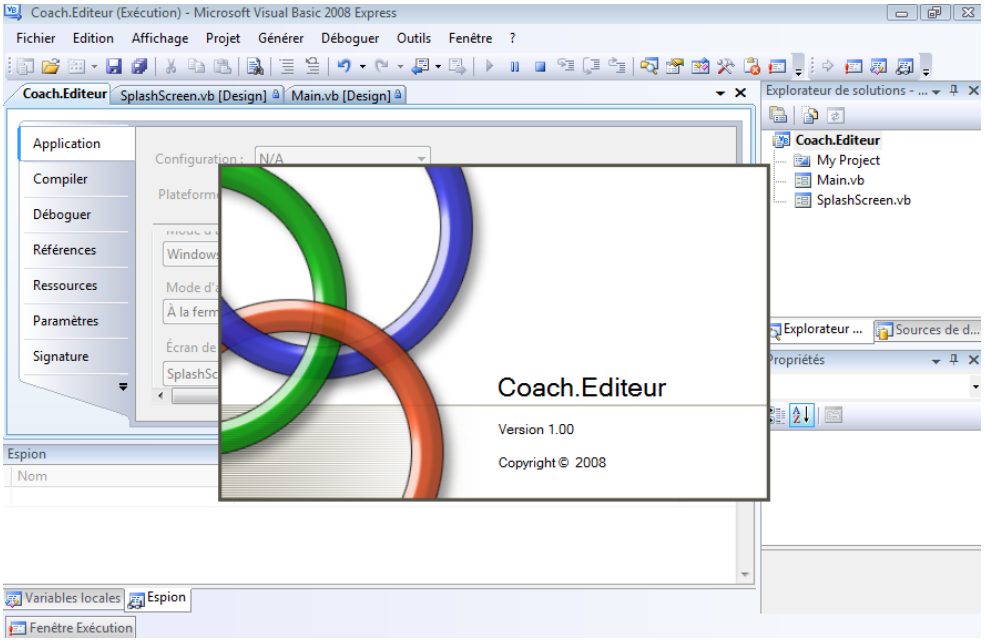
- Enregistrez vos changements en cliquant sur l'icône dans la barre d'outils standard de Visual Studio.



6. Testez le comportement de l'application en mode d'exécution :

   - Relancez l'application (F5).

Visual Studio affiche pendant quelques secondes notre formulaire de démarrage. C'est cool !

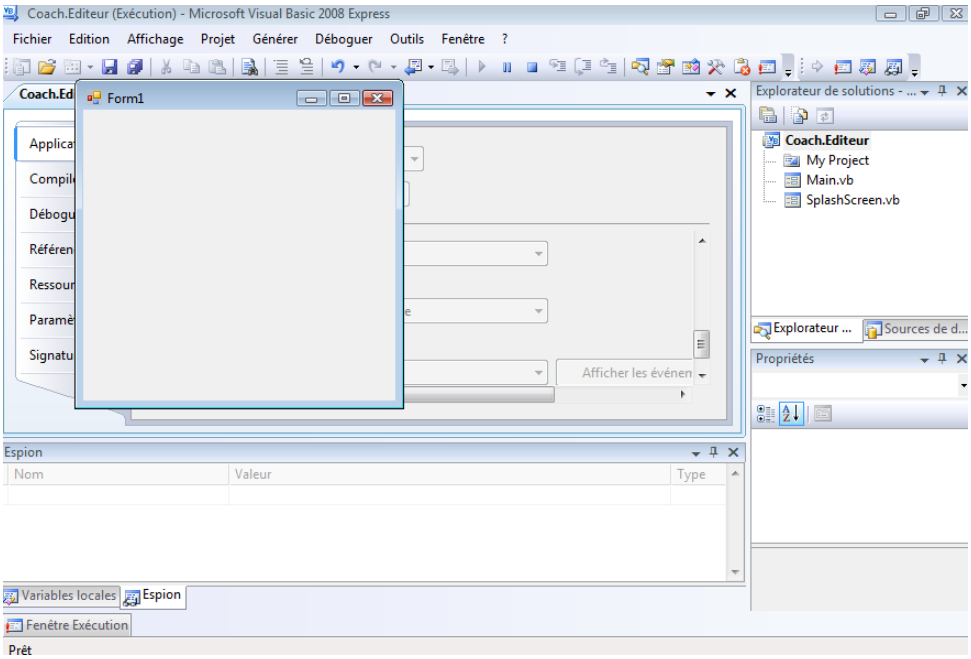


Mais vous avez pu constater que si le paramétrage de l'écran de démarrage ne nous a demandé que quelques minutes, il ne nous est pas possible en revanche de régler par exemple la durée d'affichage de l'écran. La solution est de la programmer en vous aidant du Framework .NET. Nous aurons bien sûr l'occasion d'utiliser l'approche par code tout au long de ce tutorial.

Pour apprendre à contrôler le temps d'affichage de l'écran de démarrage, rendez-vous sur :

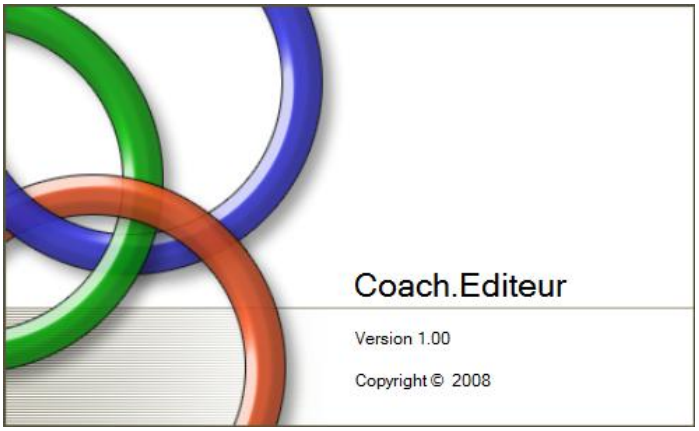
http://msdn.microsoft.com/fr-fr/library/ms234874(en-us,VS.85).aspx

Enfin, au bout de quelques secondes, c'est au tour du formulaire principal de notre projet de s'afficher :



- Arrêter l'exécution de l'application en cliquant l'icône du formulaire Form1.

Juste une petite question : est-ce que vous êtes certain que c'était bien votre formulaire **SplashScreen** qui s'est affiché au démarrage ?



Comment se fait-il que le titre de l'application soit **Coach.Editeur**, que l'écran indique un numéro de version **1.00** et un copyright avec l'année **2008** ?

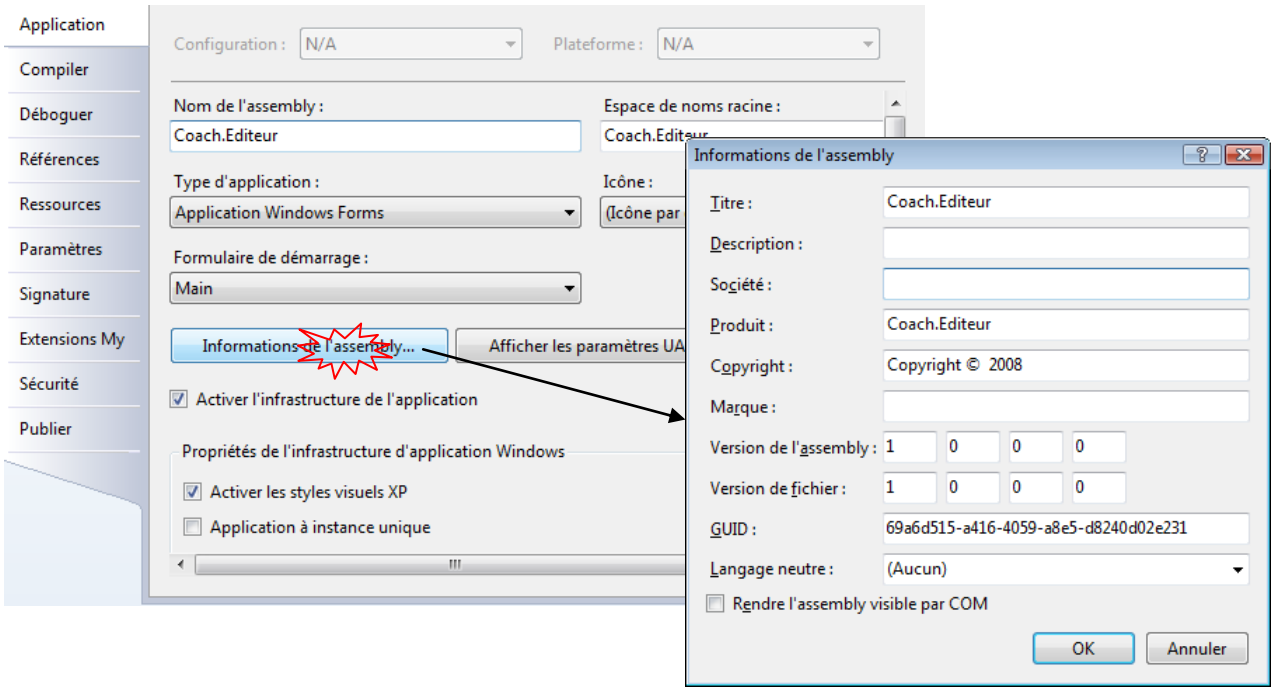
Pour rappel, votre formulaire ressemblait à ceci lorsque que vous l'avez dessiné :

L'explication est simple : le modèle d'élément que nous avons utilisé pour construire l'écran de démarrage ne s'est pas contenté de *dessiner* un formulaire de démarrage type. Il vous a fourni en même temps le *code de traitement* qui contrôle le comportement du formulaire. Ce dernier consiste à afficher les informations de l'application dans les zones correspondantes de l'écran.

Où le code trouve-t-il les informations du programme ?

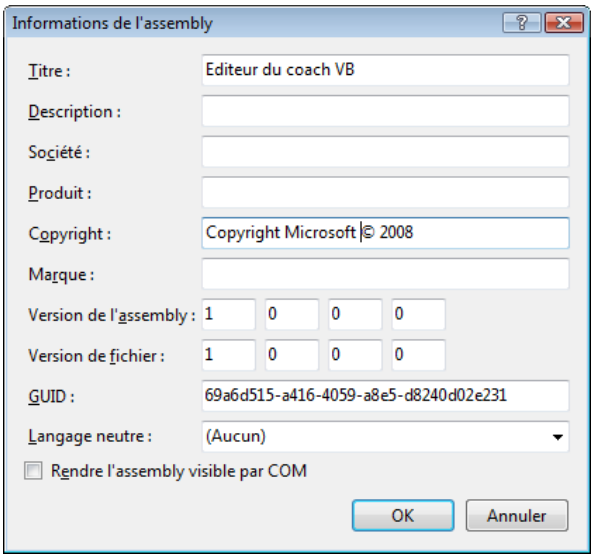
Celles-ci constituent des propriétés du projet configurables directement dans le Concepteur de projets.

- Dans l'**Explorateur de Solutions**, double cliquez sur **My Project** pour afficher à nouveau le **Concepteur de projets** ou basculez sur l'onglet **Coach.Editeur** s'il est encore ouvert.
- Dans l'onglet **Application**, cliquez sur le bouton **Informations de l'assembly…**

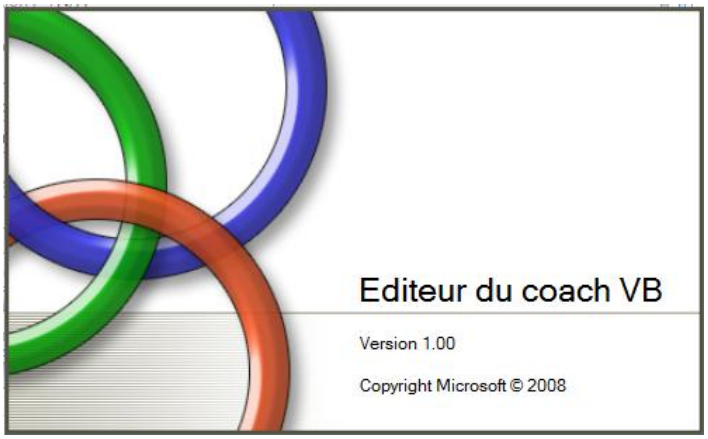


Les informations de titre, de version et de copyright récupérées par le formulaire de démarrage sont remplies ici par défaut en s'inspirant du nom de votre projet.

- Remplacez par exemple le titre de l'application par **Editeur du Coach VB**.
- Rajoutez le nom de votre société dans le Copyright.



- Validez par **OK**.
- Enregistrez vos modifications dans le Concepteur de projets.
- Validez le nouvel écran de démarrage en relançant l'exécution de l'application (F5) :



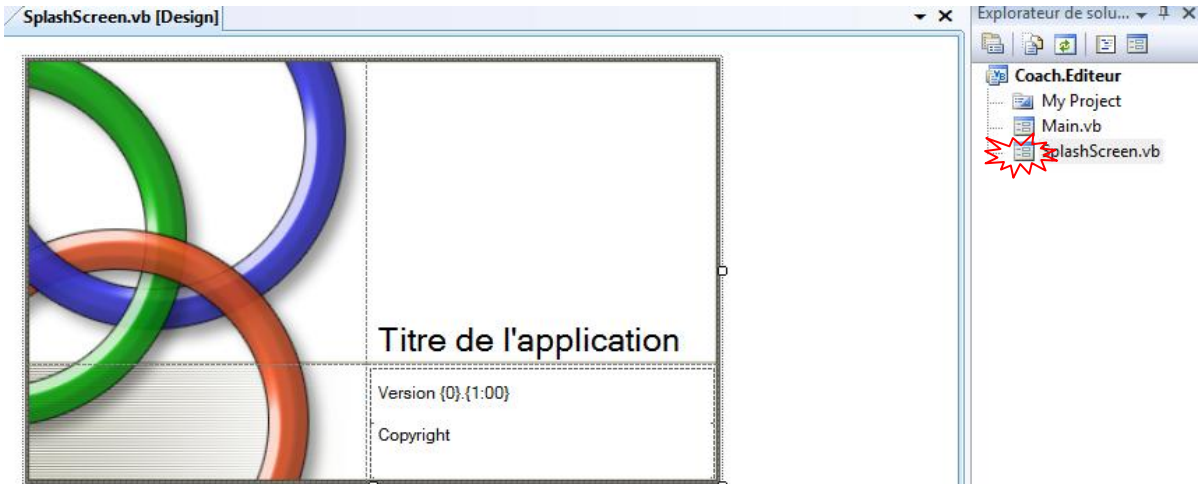
COMPRENDRE LE FONCTIONNEMENT D'UN FORMULAIRE

En fait, un formulaire Windows n'est rien d'autre qu'une classe d'objet. Nous allons voir de quelle manière nous pouvons travailler sur cette classe pour l'enrichir.

Déroulement de l'exercice :

1. Observez la structure du formulaire de l'écran de démarrage SplashScreen :

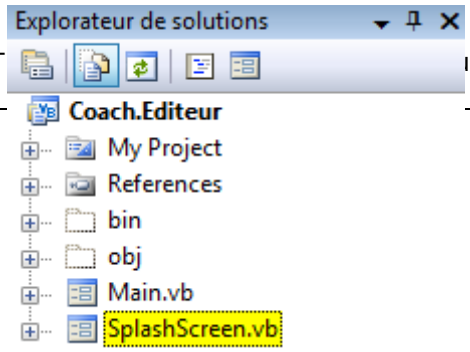
   - Dans l'**Explorateur de solutions**, double cliquez sur le fichier **SplashScreen.vb**.

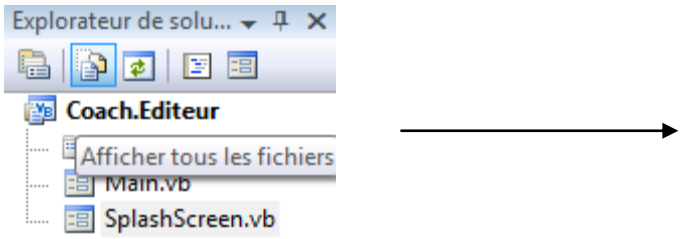


Que se passe-t-il ?

Le Concepteur de formulaire (ou concepteur de vues) de Visual Studio vous affiche l'interprétation graphique du code de définition de la classe. Cette représentation WISIWIG est très utile pour dessiner rapidement le formulaire mais dans les coulisses, tout ce que vous dessinez est automatiquement transcrit au niveau du fichier de code de la classe.

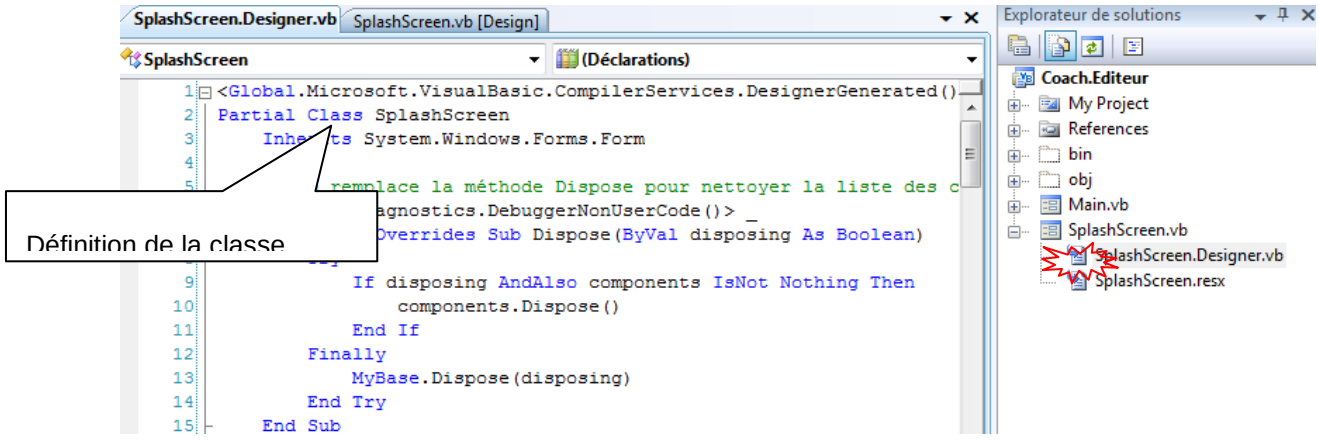
- Pour afficher le fichier de code contenant la définition design du formulaire, cliquez sur l'icône **Afficher tous les fichiers** dans la barre d'outils de l'**Explorateur de solutions**.





Pour éviter de se disperser, l'**Explorateur de solutions** de Visual Studio ne vous affiche pas par défaut l'intégralité de tous les fichiers dans la structure du projet. Par exemple, les dossiers **bin** et **obj** ne sont pas nécessairement très utiles au moment du développement donc sont cachés par défaut. Il en est de même pour les fichiers de code des formulaires générés automatiquement par le Concepteur de formulaire.

- Etendez le nœud ⊞ du fichier **SplashScreen.vb** et double cliquez sur le fichier **SplashScreen.Designer.vb**.



Dans ce fichier de code, vous trouvez la définition de la classe du formulaire nommée **SplashScreen** et toutes les informations nécessaires pour construire le panneau contenant l'image, les zones d'affichage du titre de l'application, du numéro de version et

du Copyright.

Exemple de la définition des propriétés de la zone d'affichage du numéro de version

```
72          '
73          'Version
74          '
75          Me.Version.Anchor = System.Windows.Forms.AnchorStyles.None
76          Me.Version.BackColor = System.Drawing.Color.Transparent
77          Me.Version.Font = New System.Drawing.Font("Microsoft Sans Serif", 9.0!, Sy
78          Me.Version.Location = New System.Drawing.Point(3, 3)
79          Me.Version.Name = "Version"
80          Me.Version.Size = New System.Drawing.Size(241, 20)
81          Me.Version.TabIndex = 1
82          Me.Version.Text = "Version {0}.{1:00}"
```

Le code généré ici est exactement celui que nous devrions développer à la main si nous voulions faire l'équivalent de ce que fait le générateur du Concepteur de formulaire. Attention, il n'est pas interdit d'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.

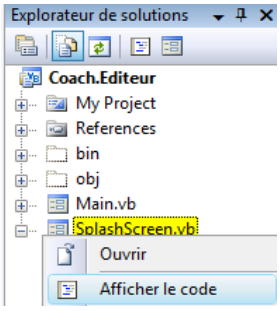
Pour éviter de nous embrouiller, Visual Studio nous demande de coder le comportement du formulaire dans un autre fichier de code séparé de celui généré par le Concepteur de formulaire. Où se trouve cet autre fichier ?

Il s'agit du fichier **SplashScreen.vb**. Pourtant, nous avons vu que lorsque l'on double

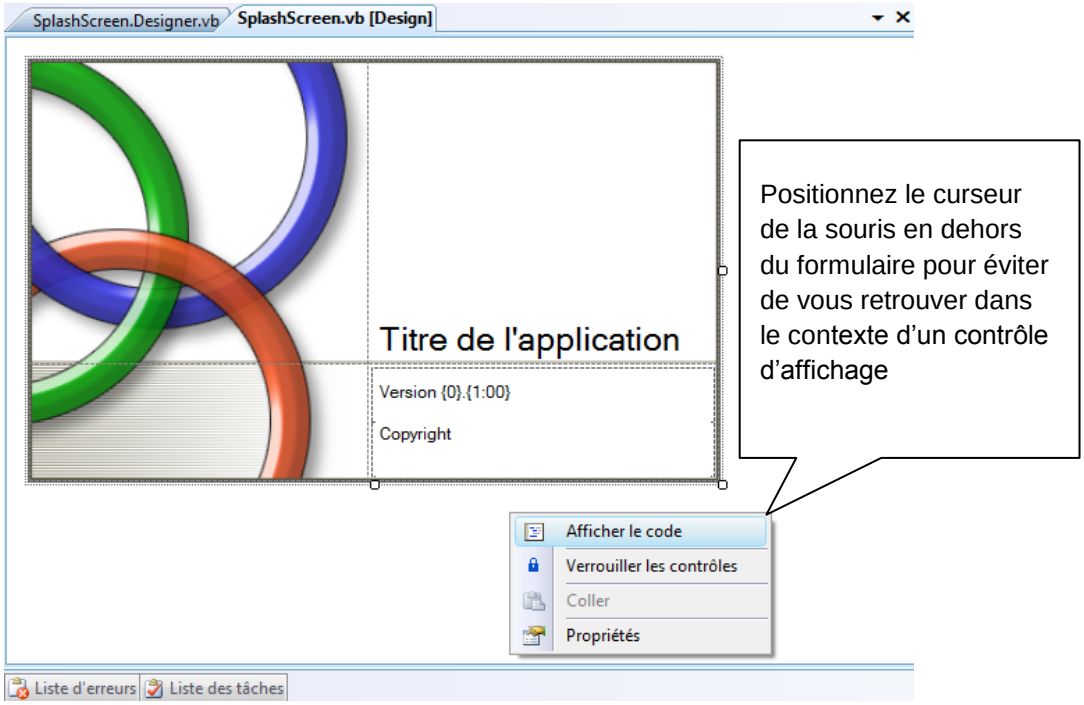
clique sur ce fichier, Visual Studio nous affiche par défaut la représentation visuelle du formulaire. En fait, c'est l'option par défaut parce qu'en général il faut bien commencer par dessiner le formulaire avant d'écrire son comportement ☺.

Pour voir le code du formulaire, vous pouvez suivre plusieurs chemins :

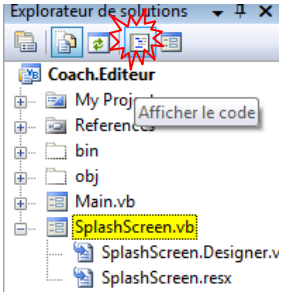
- Faites un clic droit sur le fichier **SplashScreen.vb** dans l'**Explorateur de solutions** > **Afficher le code**.



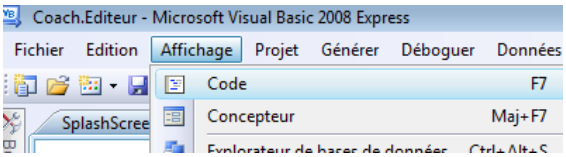
- Faites un clic droit n'importe où sur le Concepteur de formulaire dans l'onglet **SplashScreen.vb [Design]**, puis > **Afficher le code**.



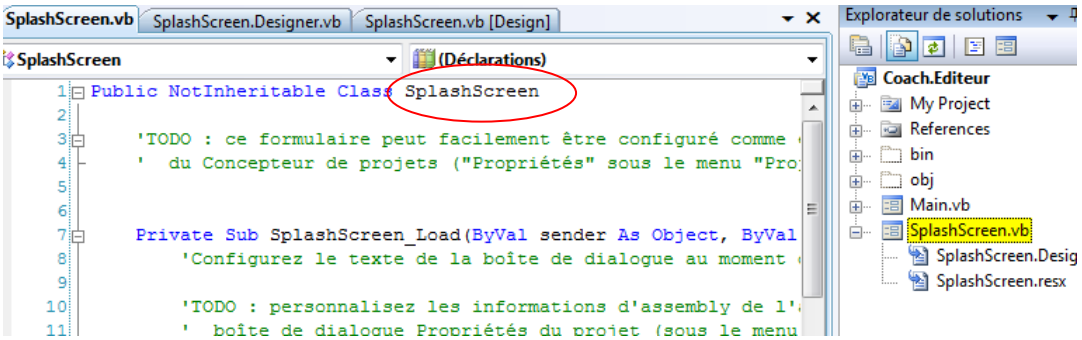
- Sélectionnez le fichier **SplashScreen.vb** dans l'**Explorateur de solutions** puis cliquez l'icône (Afficher le code) de la barre d'outils. Inversement il suffit de cliquer sur l'icône (Afficher le concepteur de vues) pour afficher la représentation graphique du formulaire.



- Ou encore Visual Studio vous propose d'afficher le fichier sélectionné par son menu **Affichage** > **Code**.

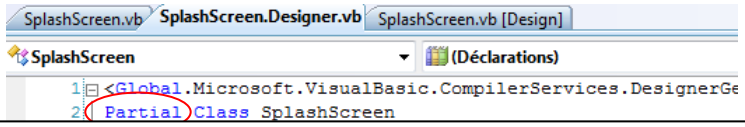


Dans ce fichier de code, vous retrouvez la définition de la classe du même nom que dans le fichier **SplashScreen.Designer.vb**.

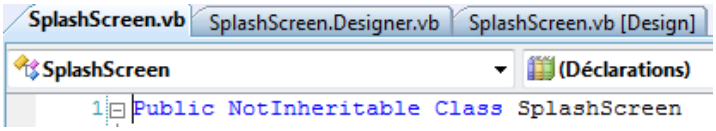


Comment fait le compilateur pour s'y retrouver dans la définition de la classe puisqu'elle est éclatée dans plusieurs fichiers distincts ?

Il s'agit là d'une propriété du langage Visual Basic proprement dit, qui autorise la définition d'une classe découpée en plusieurs fichiers physiques. Pour indiquer au compilateur qu'il va devoir recoller les morceaux et fusionner l'ensemble des déclarations pour n'en faire qu'une, on ajoute à la directive de classe le mot clé **Partial** :



Dans le second fichier, le mot **Partial** est omis mais si l'on voulait être rigoureux il devrait apparaître. VB n'autorise d'ailleurs pas plus d'une déclaration à omettre le mot clé.



La classe **SplashScreen** est une classe dite **partielle**, c'est-à-dire que la définition de la classe est divisée en plusieurs déclarations.

Pour tout savoir sur les classes partielles en langage VB, cliquez sur :

http://msdn.microsoft.com/fr-fr/library/yfzd5350.aspx

2. Observez le code du comportement du formulaire SplashScreen : il consiste à récupérer les informations de l'assembly pour les afficher dans les zones de l'écran de démarrage correspondant :

   - Affichez le code du fichier fichier **SplashScreen.vb**.
   - Retrouvez la procédure **SplashScreen_Load** qui est exécutée au moment de l'affichage de l'écran de démarrage.
   - Recherchez par exemple les lignes de code qui récupère le titre de l'application :

```
13          'Titre de l'application
14          If My.Application.Info.Title <> "" Then
15              ApplicationTitle.Text = My.Application.Info.Title
16          Else
17              'Si le titre de l'application est absent, utilisez le nom de l'applicatior
18              ApplicationTitle.Text = System.IO.Path.GetFileNameWithoutExtension _
19                                      (My.Application.Info.AssemblyName)
20          End If
```

Que signifie le mot **My** écrit en bleu ?

Comme le Framework .NET contient une multitude de classes et qu'il n'est parfois par simple de s'y retrouver, Visual Basic se propose de vous fournir des raccourcis vers quelques fonctionnalités les plus couramment utilisées du Framework. Comment ? Grâce à un espace de nom qui lui est propre appelé **My**, qui contient un certain nombre de classes organisées hiérarchiquement vous donnant un accès rapide aux informations concernant l'application et son environnement d'exécution. Vous y trouvez par exemple les classes **My.Application**, **My.Computer**, **My.User** ou encore **My.Resources** qui donnent un accès aux ressources de l'application.

D'un point de vue programmation objet, les classes de l'espace de nom **My** sont vraiment très faciles à utiliser car elles n'ont pas besoin d'être instanciées. En quelque sorte, elles vous fournissent des objets immédiatement opérationnels sur lesquels vous pouvez directement travailler. Nous reparlerons de cette notion « d'instanciation » lorsque nous aborderons les principes objet.

Ainsi pour connaître le titre de votre application, il suffit de récupérer la valeur de la propriété **Title** de l'objet **My.Application.Info** dans l'espace de noms **My**. Si celui-ci n'est pas renseigné, le code recherche le nom du fichier de sortie de l'application (auquel on soustrait l'extension) via la popriété **AssemblyName** de l'objet **My.Application.Info**.

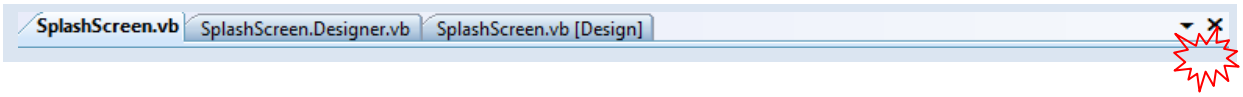
Pour tout savoir sur le développement avec **My** :

http://msdn.microsoft.com/fr-fr/library/5btzf5yk.aspx

Pour en savoir plus spécifiquement sur l'objet **My.Application.Info** :

http://msdn.microsoft.com/fr-fr/library/0f1ec0yf.aspx

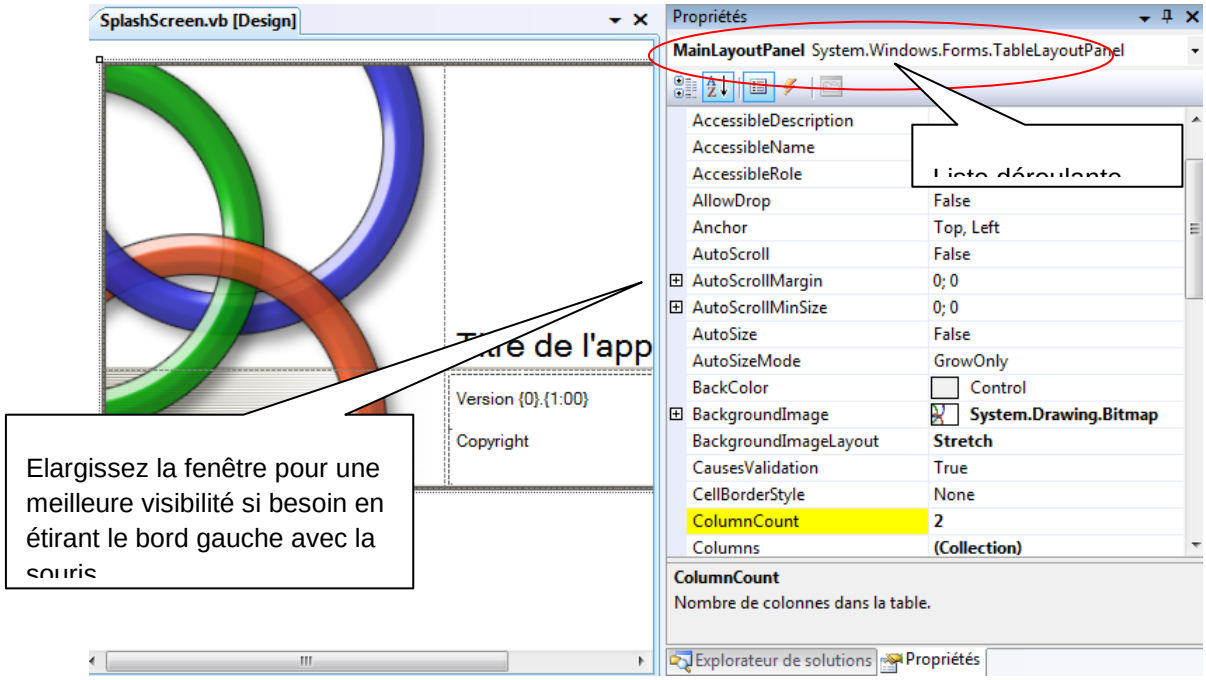
- Fermez le fichier **SplashScreen.vb** en sélectionnant le menu **Fichier** > **Fermer** ou en cliquant la croix à droite de la fenêtre ouverte sur la surface de travail.



D'une manière générale, c'est une bonne pratique de fermer au fur et à mesure les fichiers ouverts sur la surface de travail sur lesquels vous ne travaillez plus.

3. Pour terminer, personnalisez l'image de l'écran de démarrage :

   - Affichez le Concepteur de formulaire en double cliquant sur le fichier **SplashScreen.vb** dans l'**Explorateur de solutions**.
   - Cliquez sur l'image à gauche de l'écran de démarrage pour la sélectionner.
   - Faites un clic droit sur la zone > **Propriétés** ou appuyez la touche **F4** pour faire apparaître la fenêtre de propriétés de l'élément d'interface que vous avez sélectionné.
   - Dans la fenêtre **Propriétés** qui apparaît 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.



Dans le cas d'un formulaire complexe contenant de nombreux éléments d'interface, cette liste déroulante est très utile pour sélectionner directement le bon élément sans passer par le Concepteur de formulaire. D'une manière générale, c'est une bonne pratique que de vérifier le nom du contrôle d'affichage dont vous modifiez les propriétés.

Notez que la fenêtre **Propriétés** est dotée d'une barre d'outils dont les boutons sont les suivants :

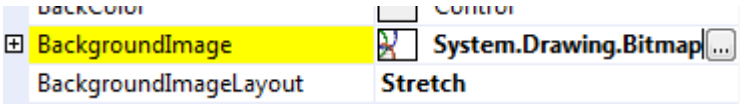
- Le bouton affiche les propriétés en les triant par catégorie ;
- Le bouton affiche les propriétés en les triant par nom ;
- Le bouton affiche les propriétés de l'objet sélectionné ;
- Le bouton affiche les événements de l'objet sélectionné – cette vue sert à ajouter facilement des méthodes de réponses aux événements (nous reviendrons sur ce point plus tard) ;

- Le bouton affiche une page de propriétés complémentaires de l'objet (si toutefois il en existe une bien sûr).

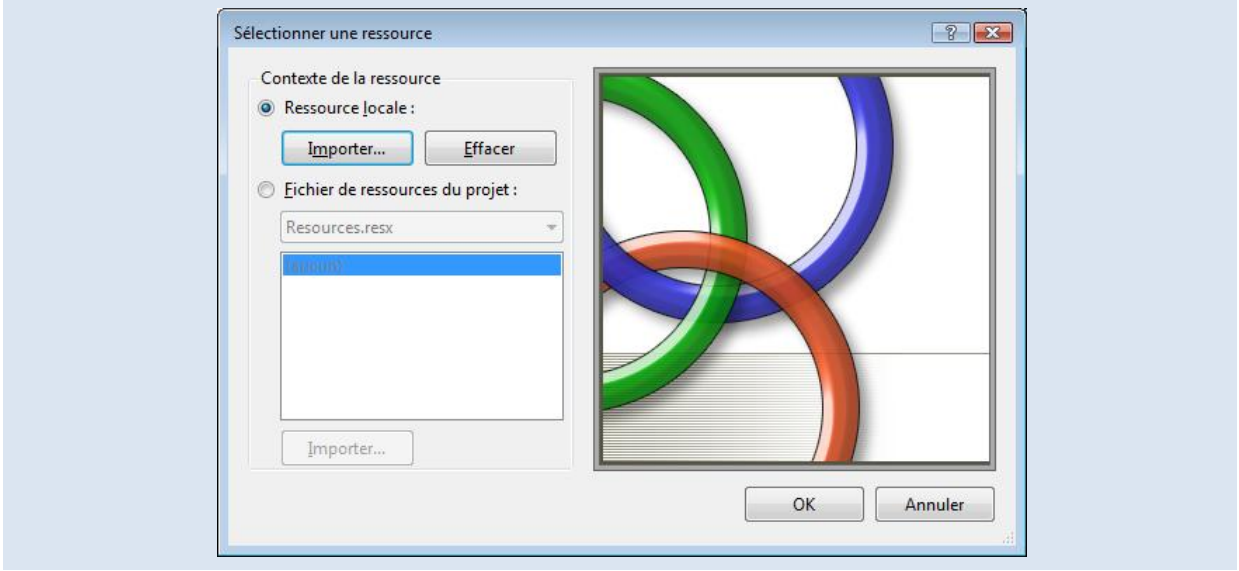
Dans la fenêtre de **Propriétés**, les propriétés affichées en **gras** sont celles dont la valeur a été modifiée par rapport à la valeur fournie par défaut par l'objet d'interface. Du coup, pour chacune de ces propriétés dont la valeur indiquée est différente de la valeur par défaut, le Concepteur de formulaire génère une (ou plusieurs) ligne(s) dans le fichier **SpashScreen.Designer.vb** pour coder le paramétrage correspondant. Merci Visual Studio !

Nous, ce qui nous intéresse est de changer l'image de fond de la zone d'affichage. Pour cela, vous disposez de la propriété **BackgroundImage** qui apparait en gras puisqu'elle a été renseignée avec l'image que vous voyez sur l'écran de démarrage.

- Sélectionnez la ligne de la propriété **BackgroundImage**.

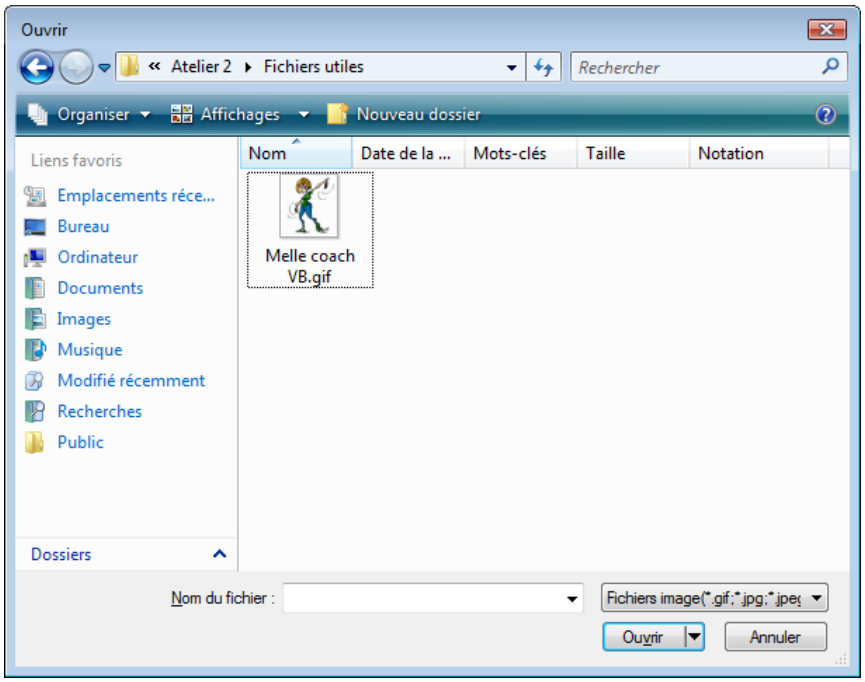


- Sélectionnez le bouton ... à droite de la zone de saisie de la valeur de la propriété.

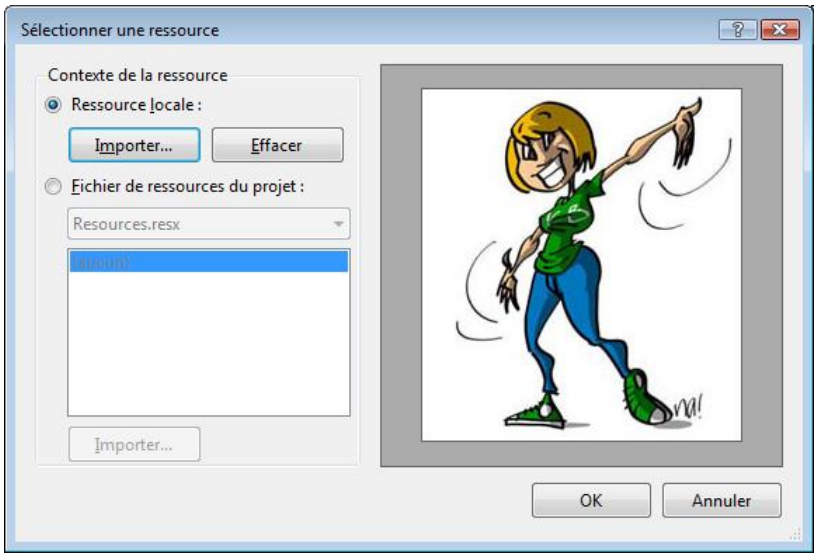


Visual Studio nous assiste dans la sélection de l'image qui se comporte comme une ressource embarquée du projet et est exposée au moment de l'exécution en tant qu'objet **System.Drawing.Bitmap**.

- Cliquez le bouton **Importer…**.
- Retrouvez le fichier **Melle coach VB.gif** représentant Melle coach VB dans le dossier **..\Atelier 2\Fichiers utiles** fourni avec l'atelier.



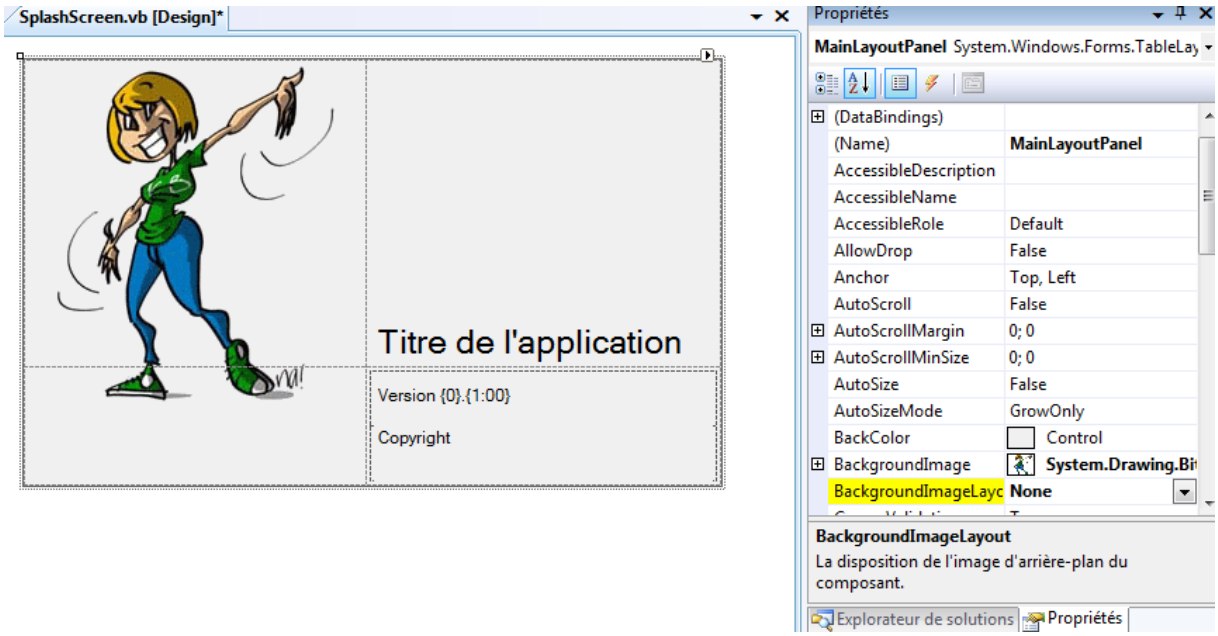
- Cliquez **Ouvrir**.



- Cliquez **OK**.

- Configurez la propriété **BackgroundImageLayout** à la valeur **None** pour supprimer l'ajustement (stretch) de l'image sur l'entière surface du contrôle d'affichage.

- Enregistrez vos modifications. Vous devez obtenir :



4. Exécutez l'application pour tester l'écran de démarrage :

   - Lancez l'application (touche F5).
   - L'écran de démarrage suivant s'affiche pendant quelques secondes avant le formulaire principal de l'application :

CONTROLER L'AFFICHAGE ET L'ARRET DE L'APPLICATION

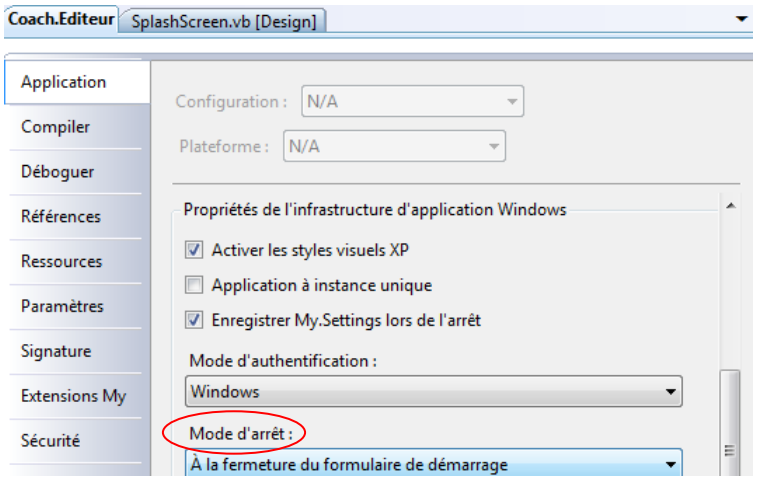
Maintenant que vous savez contrôler le démarrage de l'application, la question est : quand l'application s'arrête-t-elle ?

Comme précédemment, deux possibilités s'offrent à vous : soit vous laissez faire Visual Studio en lui donnant un minimum d'information pour qu'il gère les choses à votre convenance ; soit vous contrôlez par code l'arrêt de l'application.

Dans cet exercice, nous allons voir comment contrôler de manière déclarative l'arrêt du programme.

Déroulement de l'exercice :

1. Configurez Visual Studio pour que l'application s'arrête lorsque l'utilisateur décide de fermer le formulaire principal de l'application :
    - Editez le Concepteur de projets en cliquant sur **My Project** dans l'**Explorateur de solutions**.
    - Dans l'onglet **Application**, vérifiez que l'option **A la fermeture du formulaire de démarrage** est sélectionnée dans la liste de la propriété **Mode d'arrêt**.



Voilà pourquoi l'application se fermait précédemment lorsque vous cliquiez l'icône en haut à droite du formulaire **Main**.

Une seconde valeur **A la fermeture du dernier formulaire** vous est proposée dans cette liste pour le cas où l'application comprendrait plusieurs formulaires. Dans ce cas, il serait certainement plus logique d'interrompre le programme lorsque tous les formulaires sont fermés.

Nous verrons dans la suite de cet atelier comment contrôler l'arrêt de l'application par code. Nous pourrons ainsi proposer à l'utilisateur de quitter l'application de deux manières :

- en cliquant l'option **Fichier** > **Quitter** du menu principal de l'application
- ou en utilisant le menu contextuel associé à l'icône de notification que nous allons programmer dans la barre de notification d'état de Windows.

Mais pour mettre au point ces deux scénarios, il faut que nous enrichissions le formulaire avec ce qu'on appelle des **contrôles Windows Forms**. C'est l'objet de l'exercice suivant.

TRAVAILLER A BASE DE CONTROLES (COMPOSANTS)

De la même manière que le Framework .NET fournit toute une palette de classes pour nous aider à programmer plus vite le code de l'application, il nous fournit une boîte à outils de contrôles d'affichage pour nous aider à dessiner plus vite l'interface d'un formulaire.

Qu'est ce qu'un *contrôle Windows Forms* ?

Ce n'est ni plus ni moins qu'une classe du Framework .NET ayant une représentation graphique, que l'on peut donc ajouter au design d'un formulaire.

Par exemple : un contrôle de type **TextBox** est un objet basé sur la classe System.Windows.Forms.TextBox du Framework .NET représentant une zone de texte sur l'écran.

Peut-être avez-vous déjà entendu parler également de *composant Windows Forms* ?

Ce sont des objets similaires aux contrôles Windows Forms à ceci près qu'ils n'ont pas d'équivalence graphique, mais sont néanmoins très utiles au moment du design d'un formulaire.

Par exemple : un composant de type **BindingSource** s'occupe de gérer la source de données liée à un formulaire et s'appuie sur la classe System.Windows.Forms.BindingSource du Framework .NET.

Pour avoir une vue d'ensemble des contrôles et composants Windows Forms, rendez-vous à l'adresse suivante :

http://msdn.microsoft.com/fr-fr/library/ettb6e2a.aspx

Enfin, sachez que vous pouvez bien évidemment développer vos propres contrôles personnalisés dans l'objectif de les partager ou de les réutiliser d'une application à une autre. Pour explorer ce sujet, rendez-vous sur le lien : http://msdn.microsoft.com/fr-fr/library/6hws6h2t.aspx

Dans cet exercice, vous allez apprendre à :

- Utiliser les contrôles Windows Form MenuStrip et NotifyIcon,
- Utiliser le composant Windows Form ContextMenuStrip,
- Développer un gestionnaire d'évènement,
- Définir et utiliser une ressource liée au projet.

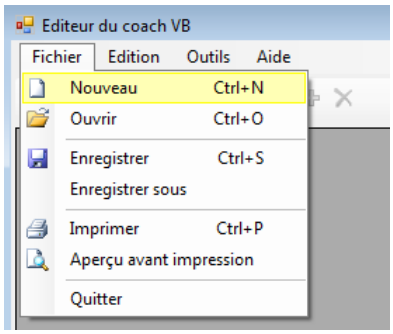
Objectif

Dans cet exercice, nous vous proposons de dessiner le formulaire principal de l'application en utilisant quelques contrôles standards du Framework .NET.

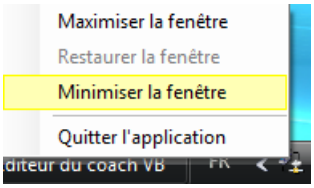
Contexte fonctionnel

L'objectif de cet exercice est de modifier le titre et les dimensions de la fenêtre principale de l'application et d'ajouter à celle-ci 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 une icône s'affichant dans la zone de notification (en bas à droite de l'écran) :



CONFIGURER LES CARACTERISTIQUES DE LA FENETRE PRINCIPALE

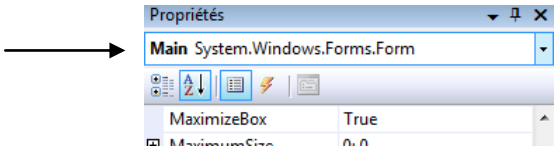
Le formulaire principal que nous avons appelé **Main** dans notre application **Coach.Editeur** est en réalité l'objet conteneur global dans lequel nous allons positionner tous les contrôles d'affichage qui vont servir à dessiner l'écran.

Au même titre qu'un contrôle, un formulaire est donc un objet possédant son propre ensemble de propriétés, de méthodes et d'évènements.

Dans cet exercice, nous allons modifier deux propriétés du formulaire que sont la taille et le titre de la fenêtre.

Déroulement de l'exercice :

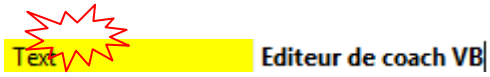
1. Affichez la fenêtre de propriétés du formulaire :
    - Affichez le fichier **Main.vb** dans le Concepteur de formulaire.
    - Faites un clic droit n'importe où sur le formulaire > **Propriétés** ou appuyez la touche **F4**.
    - Dans la fenêtre **Propriétés** qui apparaît sur la droite, vérifiez dans la liste d'objets que vous êtes sur l'objet **Main** de type **System.Windows.Forms.Form**.



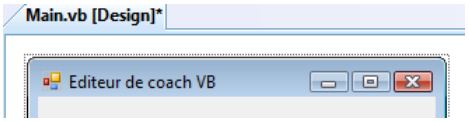
1. Modifiez le texte dans la barre de titre du formulaire :

   - Dans la liste des propriétés de l'objet **Main**, sélectionnez la propriété **Text**.
   - Tapez par exemple la valeur suivante : **Editeur de coach VB**.

Pour saisir la valeur d'une propriété, il suffit de cliquer sur l'intitulé de la propriété et vous pouvez démarrer aussitôt la saisie. En effet, pour gagner du temps, il n'est pas nécessaire de positionner le curseur dans la zone de texte de saisie de la valeur.

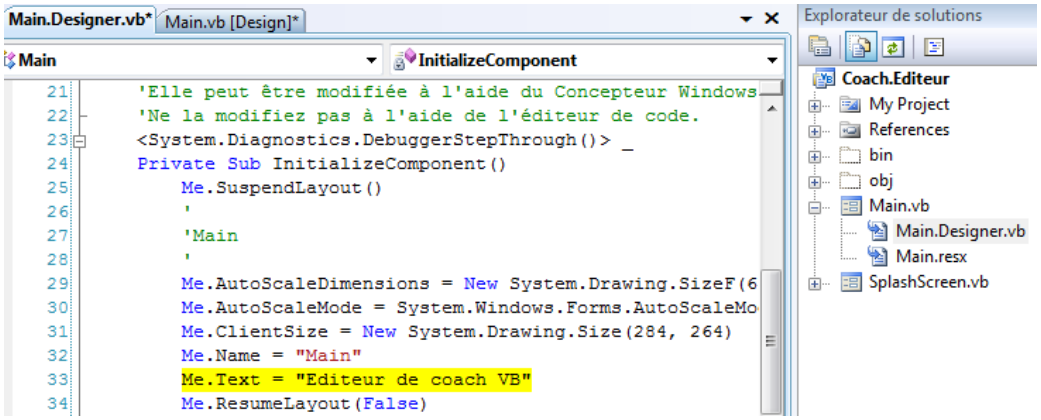


   - Validez par la touche **Entrée** ou cliquez ailleurs dans Visual Studio. Notez que le titre du formulaire reproduit immédiatement le nouvel intitulé.



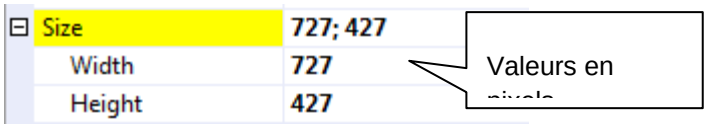
Nous allons retrouver cette propriété **Text** au niveau de nombreux contrôles. Elle référence toujours ce qui est affiché par le contrôle à l'écran, que ce soit le texte d'un libellé, le titre d'un bouton etc…

Vous pouvez éditer le fichier **Main.Designer.vb** à partir de l'Explorateur de solutions pour observer la nouvelle ligne de code générée par le Concepteur de formulaire suite à votre modification.

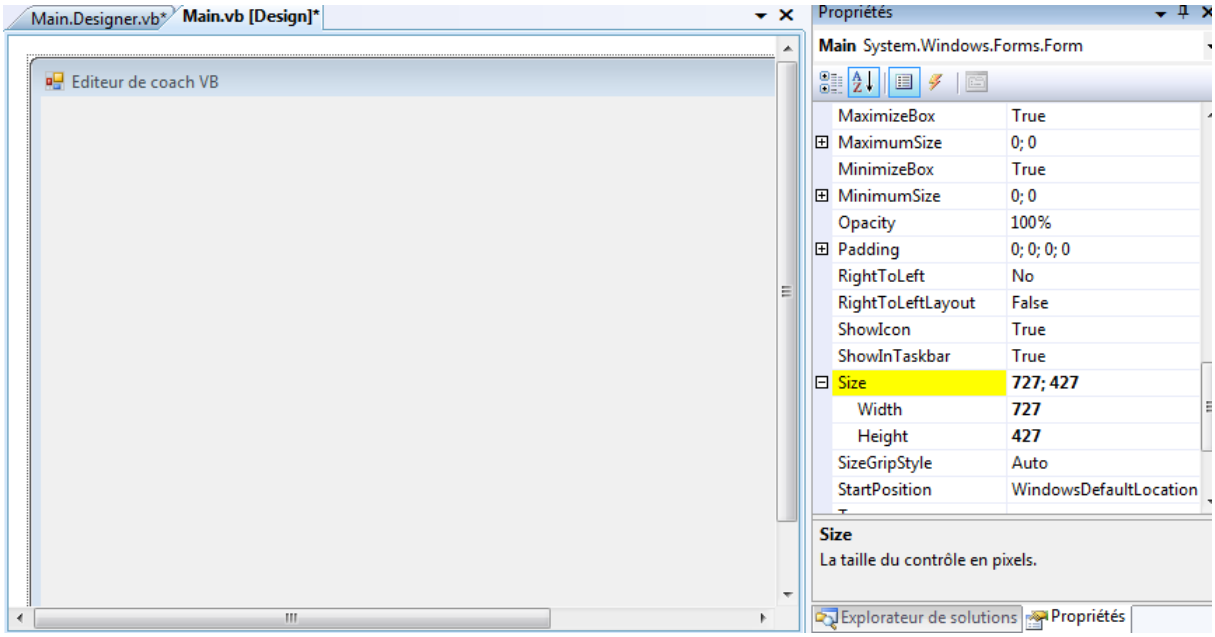


2. Modifiez la taille du formulaire :

   - Toujours dans la fenêtre **Propriétés** de l'objet **Main**, sélectionnez la propriété **Size** (dimension).
   - Ouvrez-la en cliquant sur ⊞, et indiquez la valeur **727** pour **Width** (largeur) et **427** pour **Height** (hauteur).

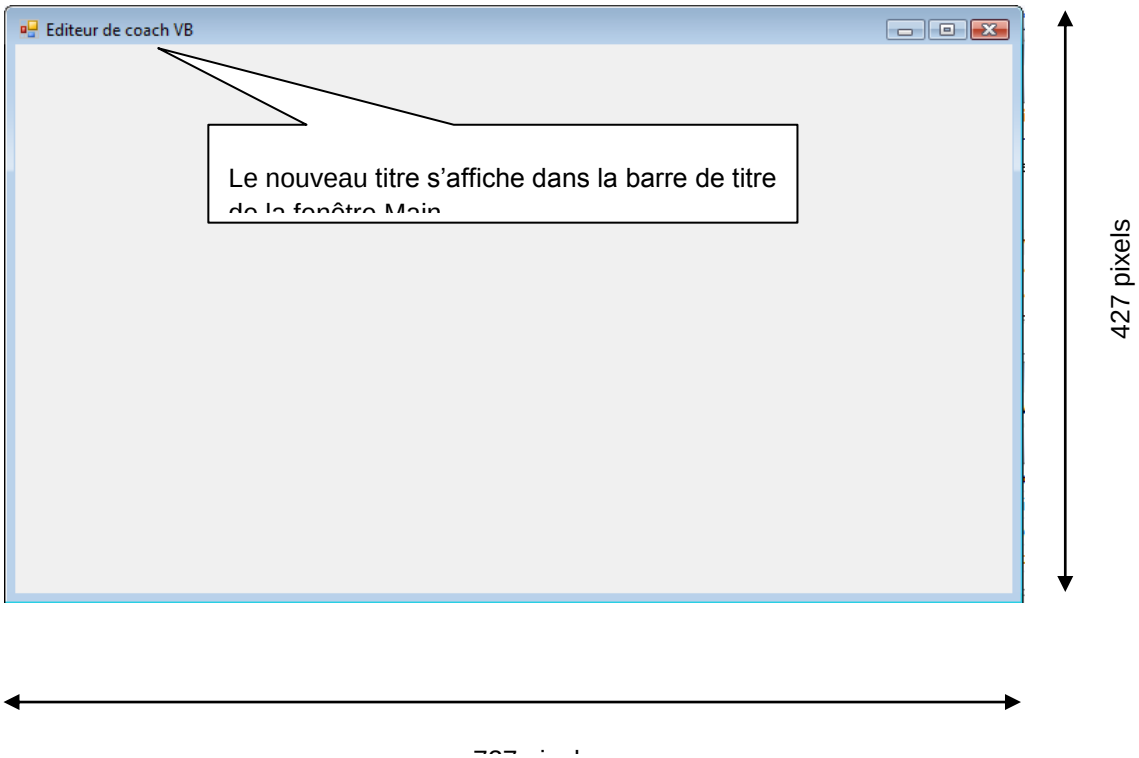


   - Le Concepteur de formulaire réajuste automatiquement les dimensions du formulaire en conséquence :



- Enregistrez vos changements.

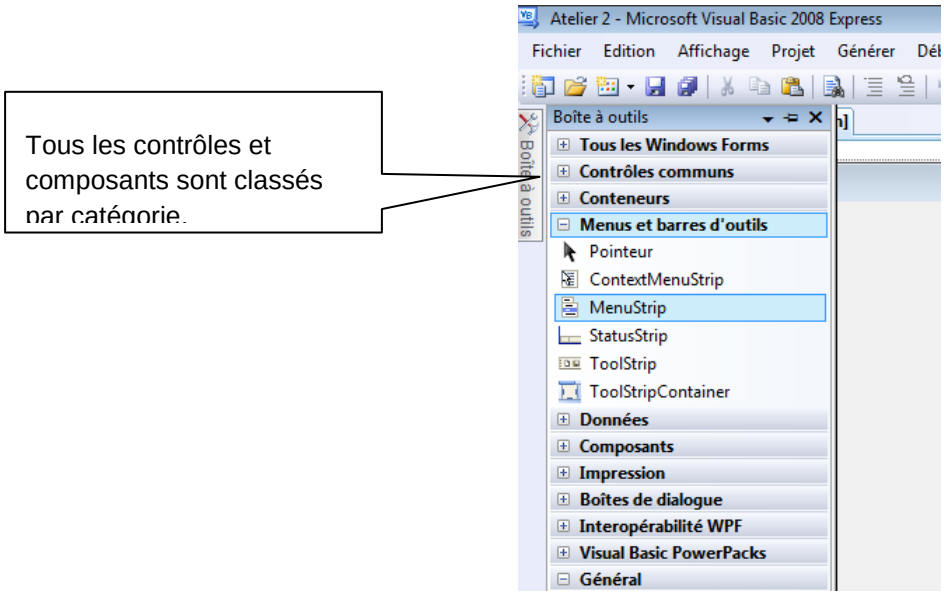
3. Exécutez l'application pour tester le formulaire :
   - Lancez l'application (touche F5).
   - Le formulaire s'affiche après l'écran de démarrage :



CONSTRUIRE LE MENU DE L'APPLICATION

Dans cet exercice nous allons ajouter notre premier contrôle au formulaire.

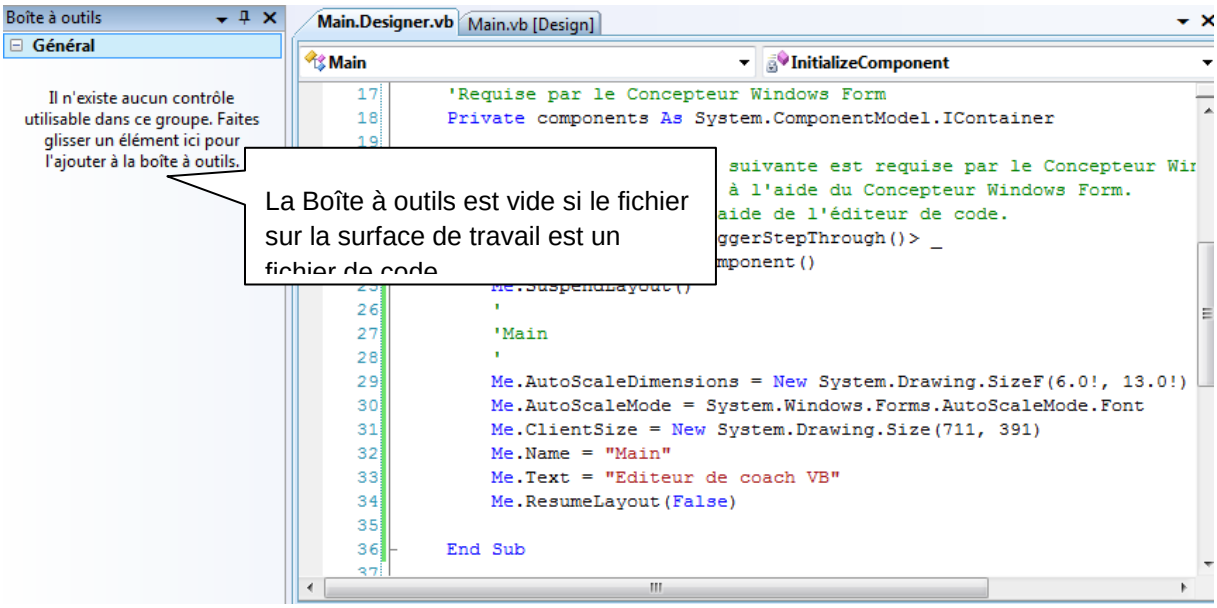
Tous les contrôles Windows Forms sont disponibles dans la **Boite à outils** de Visual Studio qui s'affiche en standard sur la gauche de votre écran. Si elle n'est pas visible, vous pouvez l'afficher en cliquant le menu **Affichage > Boite à outils** ou avec la combinaison de touches **Ctrl+Alt+X**.



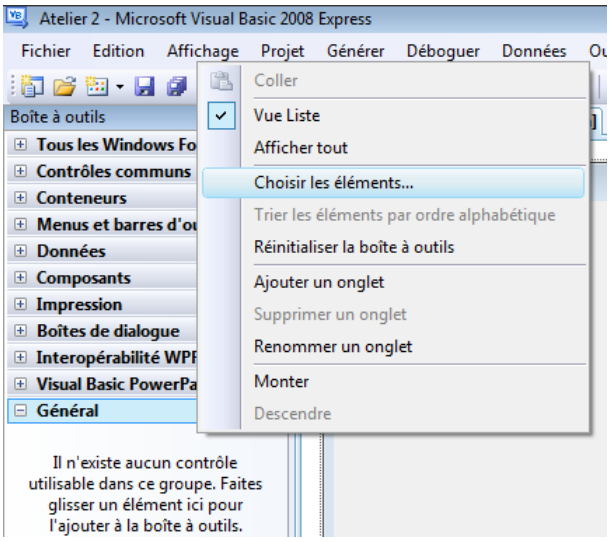
Déroulement de l'exercice :

1. Dessinez une barre de menu sur le formulaire Main à l'aide d'un contrôle MenuStrip :
    - Commencez par afficher le formulaire **Main** en mode Design.

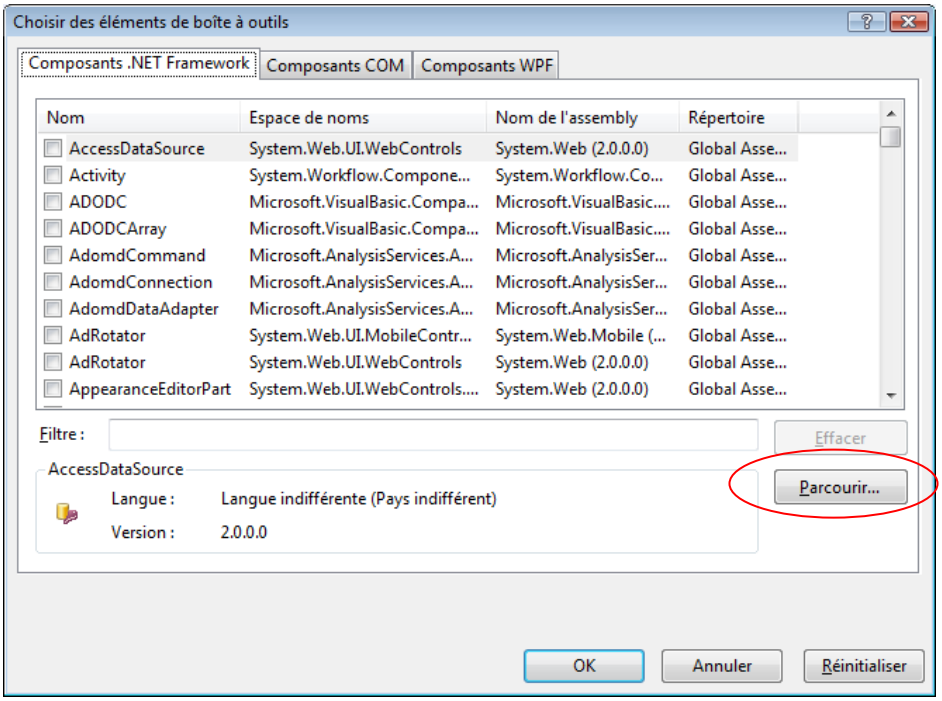
La **Boite à outils** affiche uniquement les composants qui sont disponibles compte tenu du fichier en cours dans la surface de travail. Par exemple, si la page active est une page de code, la **Boite à outils** est vide. Voilà pourquoi il est important de commencer par afficher le Concepteur de formulaire de la fenêtre sur laquelle vous voulez travailler.



Si vous ne trouvez pas le contrôle qui vous intéresse dans la liste de la boîte à outils, vous pouvez l'ajouter (ou à l'inverse l'en retirer) en faisant un **clic-droit** dans une rubrique quelconque de la **Boite à outils** > **Choisir les éléments…** :

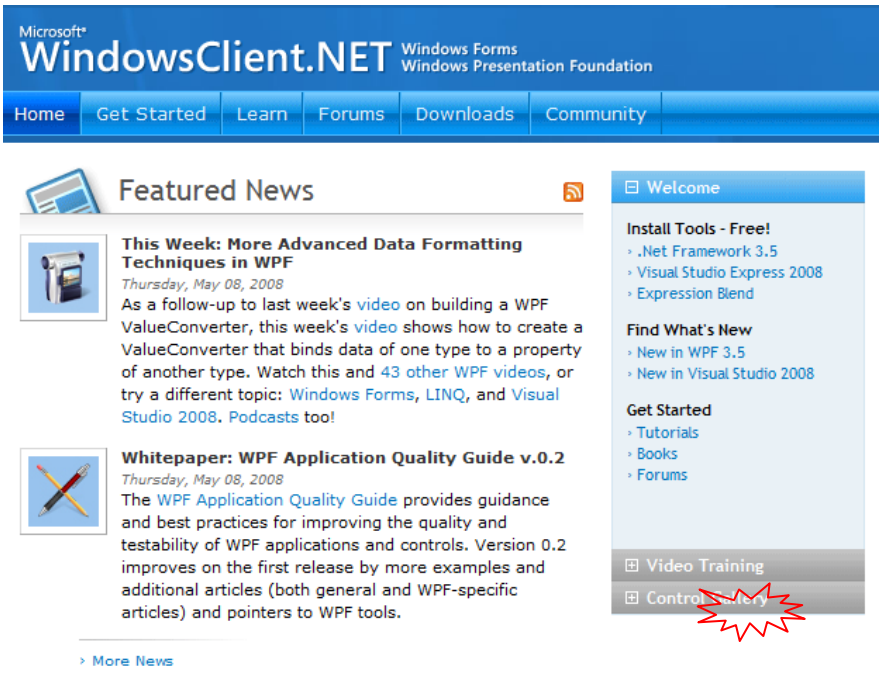


La boîte de dialogue **Choisir des éléments de boîte à outils** vous aide à sélectionner les composants ou contrôles dont vous avez besoin. La boîte peut prendre un certain temps à s'afficher du fait de la multitude de contrôles à charger dans la liste.

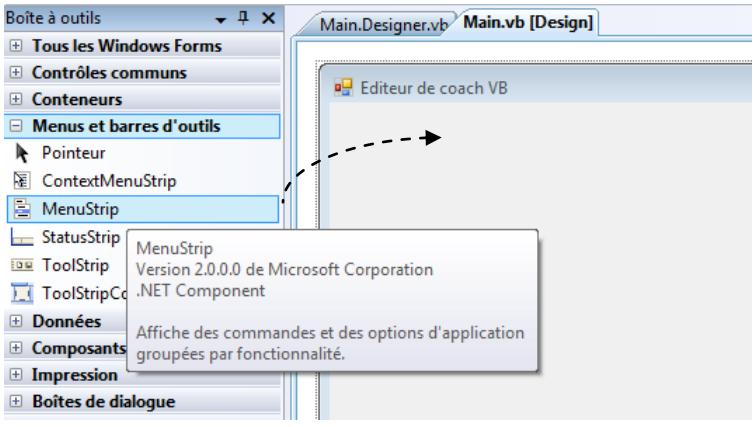


Le bouton **Parcourir** permet de sélectionner directement un assembly (.dll) qui contiendrait les contrôles voulus, à partir d'un emplacement sur disque.

Vous pouvez ajouter des composants issus d'une bibliothèque de contrôles que vous auriez vous-même développer, de nombreuses sociétés tierces à Microsoft, ou de sites communautaires comme http://windowsclient.net/. Sur la page d'accueil de ce dernier, sélectionnez la galerie de contrôles **Control Gallery** pour consulter la liste de contrôles et d'exemples téléchargeables.

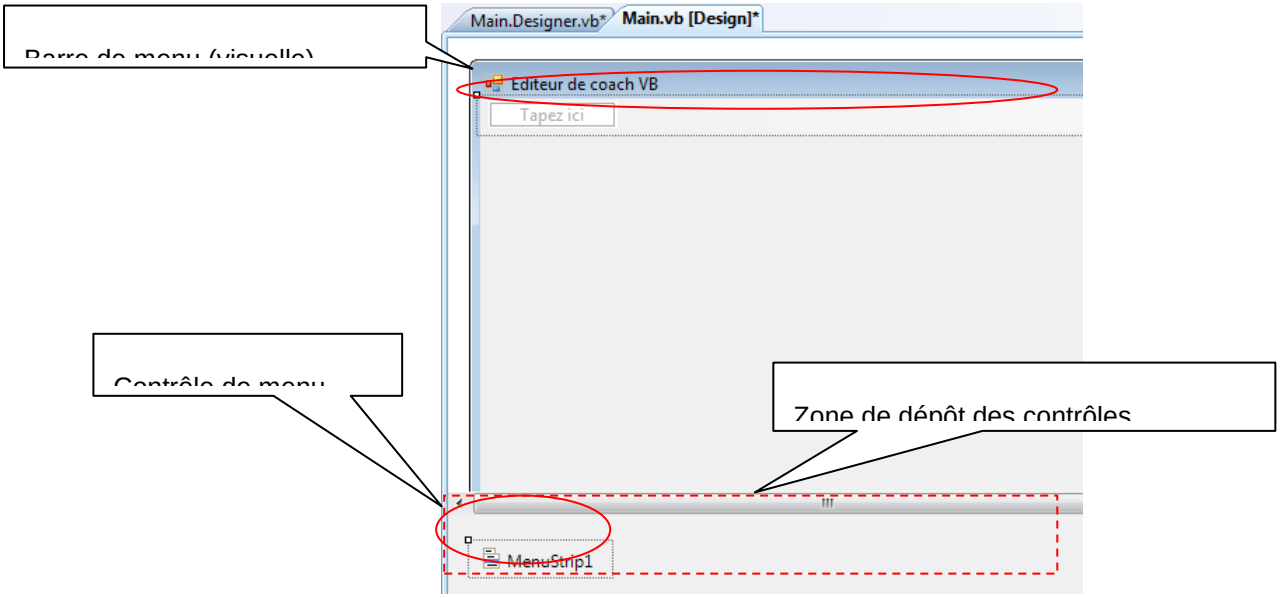


- Ouvrez la Boîte à outils de Visual Studio et fixez-la en cliquant sur l'icône (sauf si elle est déjà figée et que l'icône affiche au contraire la punaise verticale).
- Faites un glisser déplacer de la catégorie **Menus et barres d'outils** du contrôle **MenuStrip** n'importe où sur la surface du formulaire **Main**.

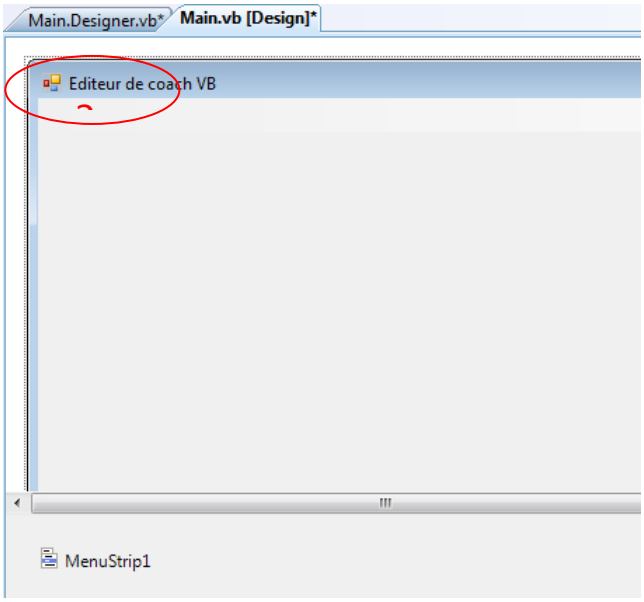


Deux nouveaux éléments apparaissent sur le formulaire :

- Une barre de menu vide sous la barre de titre de la fenêtre,
- Et un composant nommé **menuStrip1** dans une nouvelle zone au bas de la surface de travail.



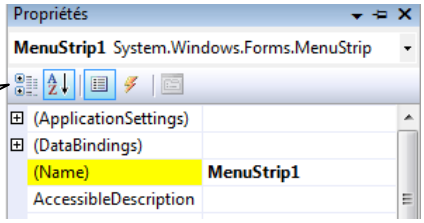
Si vous cliquez maintenant n'importe où sur le formulaire, la barre de menu sous la barre de titre disparait.



Ceci s'explique par le fait que le menu ne comporte pour l'instant aucune option donc Visual Studio ne sait pas comment le dessiner. D'où l'intérêt de la zone de dépôt au bas du formulaire, qui permet de retrouver quoiqu'il arrive le contrôle, de façon notamment à travailler sur les propriétés de celui-ci.

- Sélectionnez le contrôle **menuStrip1** dans la zone de dépôt. La ligne de menu doit réapparaître en dessous de la barre de titre du formulaire.
- Faites un **clic-droit** sur le contrôle **menuStrip1** > **Propriétés** ou appuyer sur **F4**.
- Dans la fenêtre de propriétés du contrôle, retrouvez le nom du contrôle donné par la propriété **(Name)**. Cette propriété apparaît parmi les premières si vous êtes en classement alphabétique (bouton de la barre d'outils des propriétés) :

L'icône classe les propriétés par catégorie selon leur fonction



Le nom des contrôles est très important puisqu'il sert à identifier l'objet correspondant dans le code. Soyez donc vigilant quant à la façon dont vous nommez vos contrôles et composants pour les retrouver facilement au moment du codage. Une bonne pratique consiste à établir un plan de nommage précis que vous pouvez suivre dès que vous avez à définir un nom de variable.

- Modifiez le nom du contrôle comme suit :

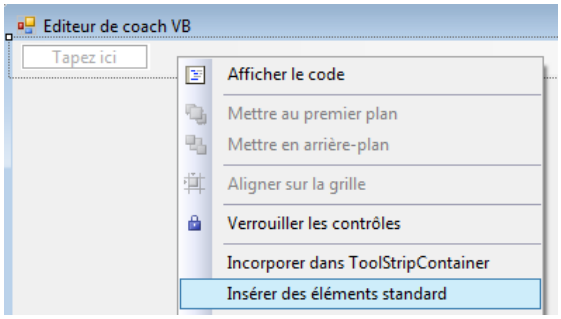
De quoi avez-vous besoin dans la barre de menu ?

L'application que nous développons va servir assez classiquement à manipuler des données. Nous avons donc besoin d'un menu **Fichier** pour manipuler des fichiers de données, d'un menu **Edition** pour les actions standards de copier/coller, d'un menu d'**Aide** etc… Ce ne sont là ni plus ni moins que les menus habituels que vous rencontrez dans toute application Windows. C'est pourquoi le Concepteur de formulaire se propose de vous donner un petit coup de pouce pour dessiner la barre de menu avec les éléments standards d'une application professionnelle.

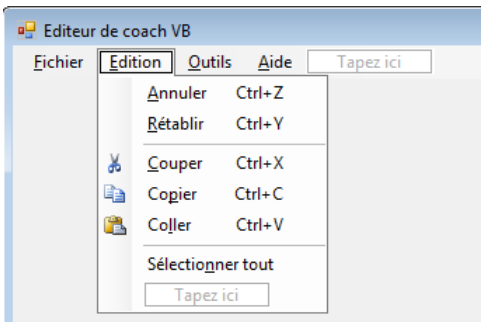
2. Ajoutez les éléments de menu standard de Windows sur la barre de menu mainMenuStrip :

   - Dans le Concepteur de formulaire, sélectionnez le contrôle **mainMenuStrip** pour faire apparaître la barre de menu sous le titre de la fenêtre.

   - Faites un clic droit sur la barre de menu > **Insérer des éléments standard**.



Vous obtenez :

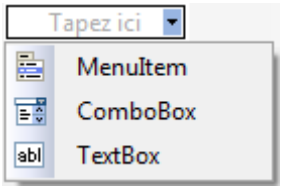


C'est carrément magique non ☺ ?

Par contre, la magie s'arrête au design car bien évidemment Visual Studio vous laisse écrire le code qui doit s'exécuter sur le clic de chacune des options de menu. Mais c'est toujours ça de pris et comme on dit, ce qui est fait n'est plus à faire…

Pour ajouter de nouvelles options de menu, il suffit de cliquez sur Tapez ici à la suite des autres options de menu ou de sous-menu.

Comme vous pouvez le constater avec les éléments standards ajoutés par Visual Studio, le contrôle MenuStrip permet de construire des menus dynamiques très riches contenant des images, des informations de raccourcis clavier, des barres d'espacement etc… Si vous cliquez par exemple la flèche qui apparaît à droite de la zone Tapez ici lorsque vous la sélectionner, une liste déroulante montre qu'il est possible de créer une zone de saisie (basée sur le contrôle standard **TextBox**) ou une liste de valeurs (basée sur le contrôle standard **ComboBox**) en tant qu'option de menu. L'option **MenuItem** créé une option de menu classique matérialisée par un libellé.

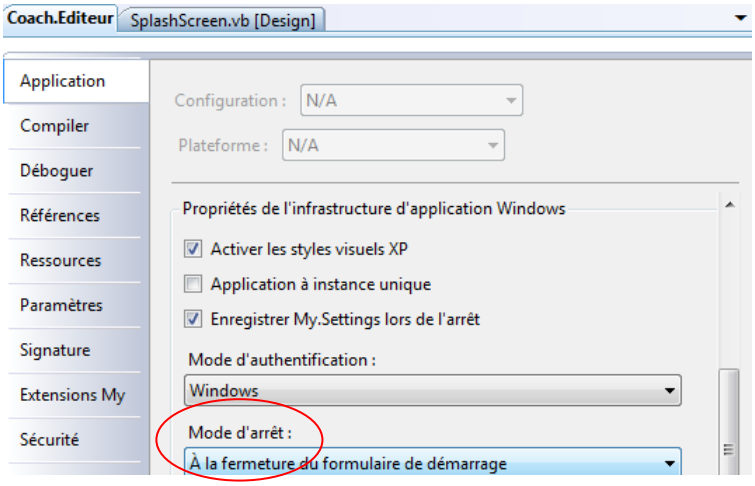


- Avant de poursuivre, enregistrez toutes vos modifications.

CODER LA FERMETURE DE L'APPLICATION

Maintenant que la barre de menu de la fenêtre principale de l'application est dessinée, il faut coder les instructions en réponse au clic de l'utilisateur sur toutes les options de celle-ci.

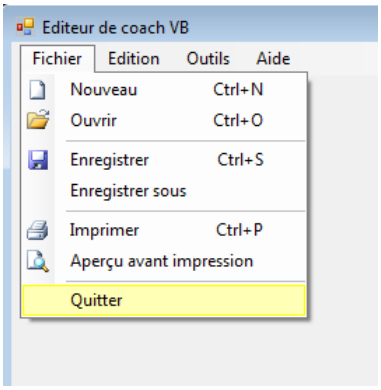
Vous vous souvenez que nous avons configuré précédemment le mode d'arrêt de l'application de manière déclarative dans le Concepteur de projets pour que l'application s'arrête à la fermeture du formulaire de démarrage.



Coder la fermeture de l'application revient donc à coder la fermeture du formulaire **Main**.

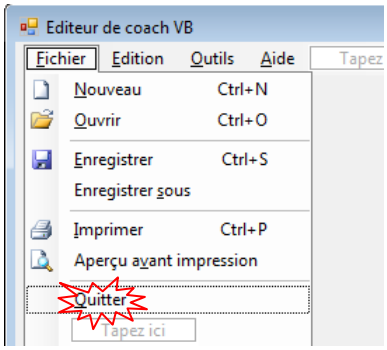
Contexte fonctionnel

Dans cet exercice, nous allons écrire le code de fermeture de l'application associé au clic sur l'option de menu **Fichier** > **Quitter** de la barre de menu du formulaire **Main**.



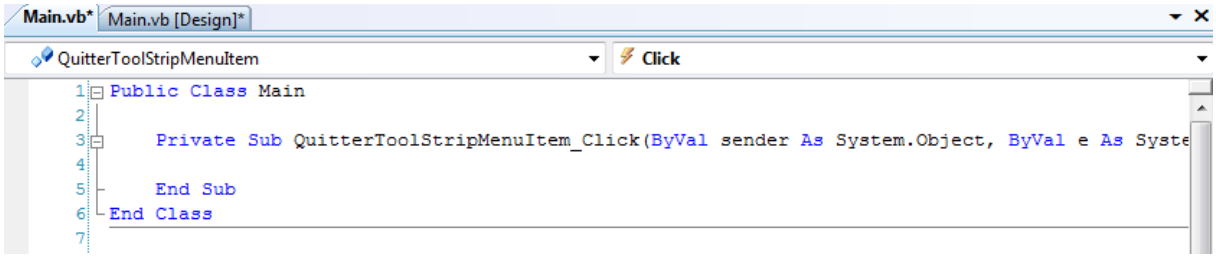
Déroulement de l'exercice :

1. Générer une procédure associée au clic sur l'option de menu Quitter :
    - Dans le **Concepteur de formulaire**, sélectionner le menu **Fichier** de la barre de menu du contrôle **mainMenuStrip** puis double cliquez sur l'option **Quitter**.



Visual Studio vous bascule automatiquement dans le fichier de code **Main.vb** et créé une procédure **QuitterToolStripMenuItem_Click** dans laquelle vous pouvez programmer le code de fermeture du formulaire :

Le design du formulaire est accessible par l'onglet correspondant resté ouvert



Cette procédure s'appelle un *gestionnaire d'évènement* (*event handler* en anglais) parce qu'elle est exécutée automatiquement lorsqu'un évènement est déclenché.

Pourquoi écrire des gestionnaires d'évènement ?

Le principe de la programmation des Windows Forms repose sur l'approche dite *évènementielle* c'est-à-dire que le code de l'application s'exécute dans un ordre non déterminé à l'avance (par opposition à un code qui s'exécuterait en suivant une séquence prédéfinie), en réponse à une action de l'utilisateur, telle que le clic de la souris ou l'utilisation des touches du clavier sur l'interface de l'application. En somme, si l'utilisateur ne clique jamais sur l'option de menu **Quitter**, le code de cette procédure destinée à fermer le formulaire ne s'exécutera jamais.

Pour toute action de l'utilisateur, le runtime d'exécution déclenche un **évènement** caractérisant cette action. Il s'offre même le luxe de déclencher d'autres évènements pas nécessairement en rapport avec une action de l'utilisateur, en fait chaque fois qu'il considère qu'il pourrait être opportun de brancher un traitement pour l'application. Par exemple, au chargement du formulaire, se produit l'évènement **Load** pour vous prévenir que c'est le moment idéal pour initialiser certaines données ou contrôles.

Du coup, coder l'application revient à écrire des gestionnaires d'évènement pour s'abonner (comme on s'abonne à un flux RSS) aux évènements percutants pour la logique de votre application et à y brancher le code de traitement approprié.

Comment écrire un gestionnaire d'évènement en réponse à un évènement ? Et oui ! Comment le système sait-il que la procédure **QuitterToolStripMenuItem_Click** est associée au clic sur le bouton **Quitter** du menu de l'application ?

Rappelez vous il y a deux approches possibles, déclarative et par code. En fait, dans notre cas c'est le Concepteur de formulaire qui a automatiquement paramétré la procédure pour nous, et ce en utilisant la méthode déclarative.

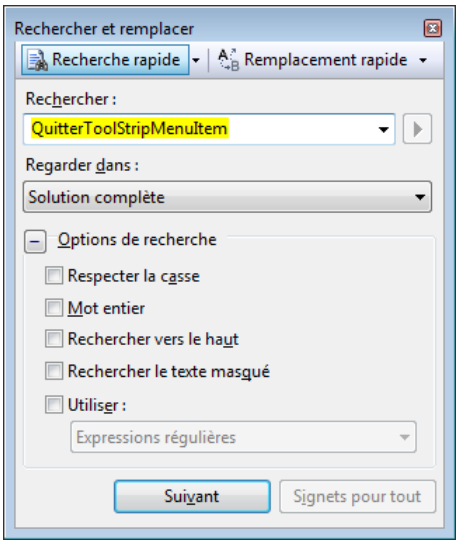
Regardons ensemble ce qui a été généré.

Contrairement à ce qu'on pourrait croire, le nom *QuitterToolStripMenuItem_Click* de la procédure n'est pas en cause !

Pour comprendre comment ça marche, il faut d'abord retrouver la déclaration de l'option de menu **Quitter** dans le fichier **Main.Designer.vb**.

2. Retrouvez la déclaration de l'objet associé à l'élément de menu Quitter :

   - Ouvrez le fichier **Main.Designer.vb** à partir de l'**Explorateur de solutions**.
   - Dans le menu de Visual Studio, cliquez sur **Edition** > **Recherche rapide** et rentrez le nom de la variable **QuitterToolStripMenuItem** dont on souhaite retrouver la définition.



- Cliquez sur **Suivant** jusqu'à retrouver la ligne suivante :

```
289     Friend WithEvents toolStripSeparator2 As System.Windows.Forms.ToolStripSeparator
290     Friend WithEvents QuitterToolStripMenuItem As System.Windows.Forms.ToolStripMenuItem
291     Friend WithEvents EditionToolStripMenuItem As System.Windows.Forms.ToolStripMenuItem
```

Notez tout d'abord le type de la variable **System.Forms.ToolStripMenuItem**. Il va nous servir dans quelques instants.

La clause **WithEvents** est le premier élément nécessaire à la construction d'un gestionnaire d'évènement. En définissant la variable **QuitterToolStripMenuItem** à l'aide de **WithEvents**, cela signifie que vous autoriser l'écriture de procédures en réponse aux évènements de cette variable.

Notez que la définition de **QuitterToolStripMenuItem** a été générée automatiquement lorsque vous avez dessiné le menu sur le formulaire (via l'option d'insertion des éléments standards). Elle correspond à la définition de la variable associée à l'option de menu dans votre programme.

Si vous cliquez plusieurs fois sur le bouton **Suivant** de la fenêtre de recherche, vous constaterez qu'il y a également d'autres références à cette même variable, par exemple pour reproduire dans le code, l'intitulé (**Text**) de l'option de menu, son positionnement (**Size**) et son identifiant (**Name**).

```
171         'QuitterToolStripMenuItem
172         '
173         Me.QuitterToolStripMenuItem.Name = "QuitterToolStripMenuItem"
174         Me.QuitterToolStripMenuItem.Size = New System.Drawing.Size(205, 22)
175         Me.QuitterToolStripMenuItem.Text = "&Quitter"
```

Le caractère & devant la lettre Q définit un raccourci clavier pour ce menu

- Fermez le fichier **Main.Designer.vb**.

Le second élément pour écrire un gestionnaire d'évènement se trouve au niveau de la définition de la procédure elle-même. Il s'agit du mot clé **Handles** qui fonctionne de paire avec la clause **WithEvents**.

3. Observez la signature du gestionnaire d'évènement généré dans le fichier de code :

   - Revenez sur la définition de la procédure **QuitterToolStripMenuItem_Click** dans le fichier **Main.vb**.

Code VB

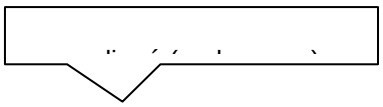
```
Public Class Main


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

```
    End Sub
End Class
```

Le mot clé **Handles** est le second élément nécessaire à la construction d'un gestionnaire d'évènement. Pensez à la littérature anglaise qui parle d'event *handler* là où nous parlons de gestionnaire d'évènement. Il signifie que la procédure **QuitterToolStripMenuItem** *gère (handles)* l'évènement **Click** de la variable **QuitterToolStripMenuItem**.

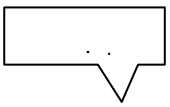
Attention une fois encore à la terminologie utilisée !

- Visual Studio nomme la procédure par défaut avec le format :



<nom de la variable>_<nom de l'évènement>

- Alors que le nom de l'évènement est noté :



<nom de la variable>.<nom de l'évènement>

En fait le nom de la procédure importe peu. Vous pourriez par exemple remplacer sans hésiter *QuitterToolStripMenuItem_Click* par *Quitter*.

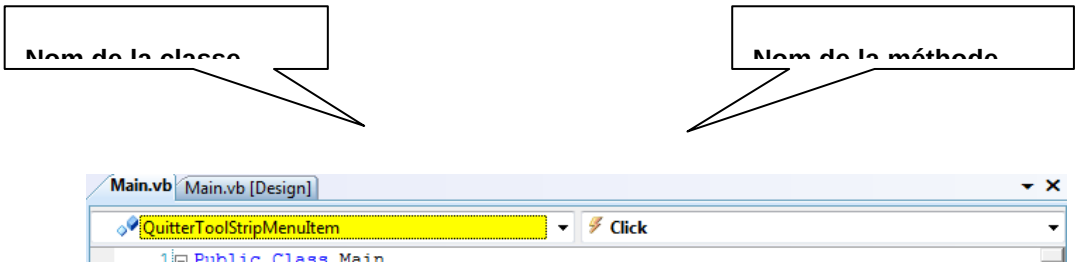
En revanche, le nom de l'évènement auquel vous l'accrochez est très important, de même que la signature de la procédure.

Lorsque vous double cliquez sur un objet en mode Design, Visual Studio génère automatiquement la procédure de réponse à *l'évènement par défaut* associé à un objet, ici l'évènement **Click** sur un objet de type **System.Forms.ToolStripMenuItem**. Mais ce type d'objet dispose de bien d'autres évènements qui vous permettent d'interagir sur son fonctionnement.

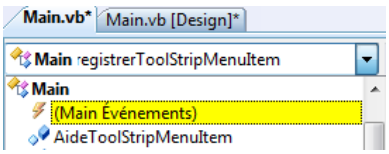
Où trouver le nom de l'évènement auquel s'abonner ?

En haut de l'éditeur de code de Visual Studio, vous trouverez une barre contenant deux listes déroulantes. Elle donne une liste des évènements disponibles pour chaque objet du formulaire.

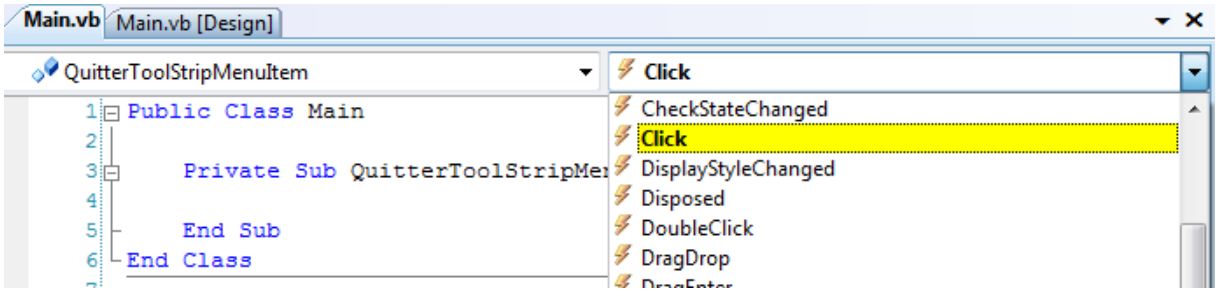
- Dans la liste de gauche **Nom de la classe**, sélectionnez l'objet **QuitterToolStripMenuItem** sur lequel porte l'évènement attendu.



Tous les autres éléments de la liste sont des objets qui font partie de votre formulaire, pour la plupart générés par l'action d'ajout des éléments standards au menu **mainMenuStrip**. Le premier élément de la liste, **(Main Evènements)** correspond à l'objet formulaire lui-même pour lequel le Framework .NET fournit toute une batterie d'évènements caractérisant le cycle de vie du formulaire.



- Dans la liste de droite **Nom de la méthode**, sélectionnez l'évènement auquel vous voulez vous abonner pour l'objet sélectionné.



L'évènement **Click** apparait en caractères gras car Visual Studio détecte que vous avez déjà un gestionnaire pour cet évènement. Si vous cliquez sur cette option, Visual Studio amène le curseur à l'emplacement du gestionnaire dans le code.

Tous les autres évènements ne sont pas en gras car aucun gestionnaire n'existe encore pour ceux-ci dans l'application. Si vous cliquez sur un évènement quelconque, Visual Studio génère la procédure de réponse à l'évènement et l'ajoute à votre code avec la signature adéquate ☺.

Au fait, qu'est ce qu'on entend par *signature* et pourquoi est-elle importante pour définir un gestionnaire d'évènement ?

La signature d'une procédure est sa ligne de déclaration comprenant :

- Le type de la méthode (procédure sans valeur de retour ou fonction avec une valeur de retour),
- La définition de tous les paramètres de la méthode avec leur type de données,
- Et le type de données de la valeur de retour de la méthode s'il y en a une (ce qui n'est pas le cas dans une procédure de réponse à un évènement).

Pour les gestionnaires d'évènement, c'est très facile, car la signature du gestionnaire

d'évènement est souvent la suivante :

Code VB

```
Private Sub NomDuGestionnaire(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) _
    Handles NomDeLEvenement


End Sub
```

où :

- **Sender** de type **System.Object**, est l'élément à l'origine du déclenchement de l'évènement (dans notre cas, ce serait donc l'objet **QuitterToolStripMenutItem**),
- Et où **e** sert à transmettre toutes informations utiles au gestionnaire pour l'aider à répondre à l'évènement. En fait, le type **System.EventArgs** représente un jeu d'information vide. Lorsqu'un évènement transmet des informations au gestionnaire, la signature de l'évènement est légèrement différente et **e** est d'un autre type.

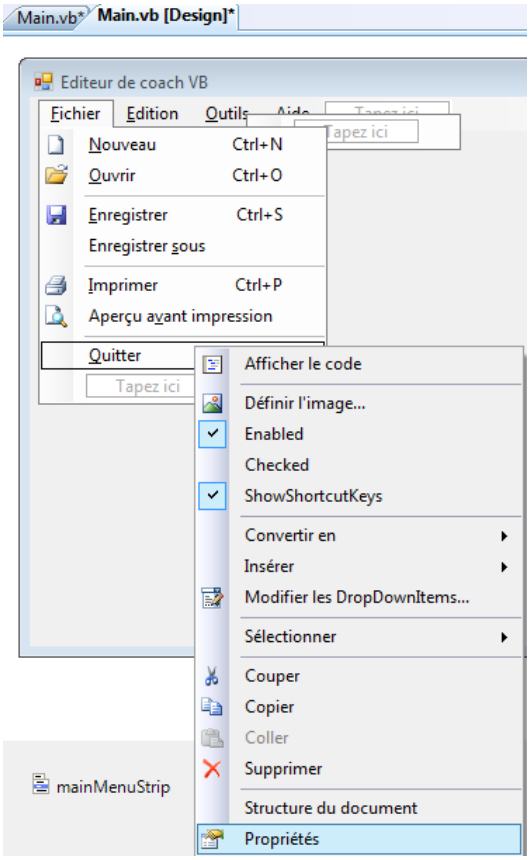
Par exemple, l'évènement **DragDrop** de l'objet **QuitterToolStripMenuItem** envoie des informations de positionnement très utile pour gérer le glisser déplacer de l'élément. La signature d'un gestionnaire pour cet évènement fait donc appel à un argument **e** de type plus complexe : **System.DragEventArgs**.

```
7   Private Sub QuitterToolStripMenuItem_DragDrop(ByVal sender As Object, _
8                                                 ByVal e As System.Windows.Forms.DragEventArgs) _
9                       Handles QuitterToolStripMenuItem.DragDrop
10
11  End Sub
```

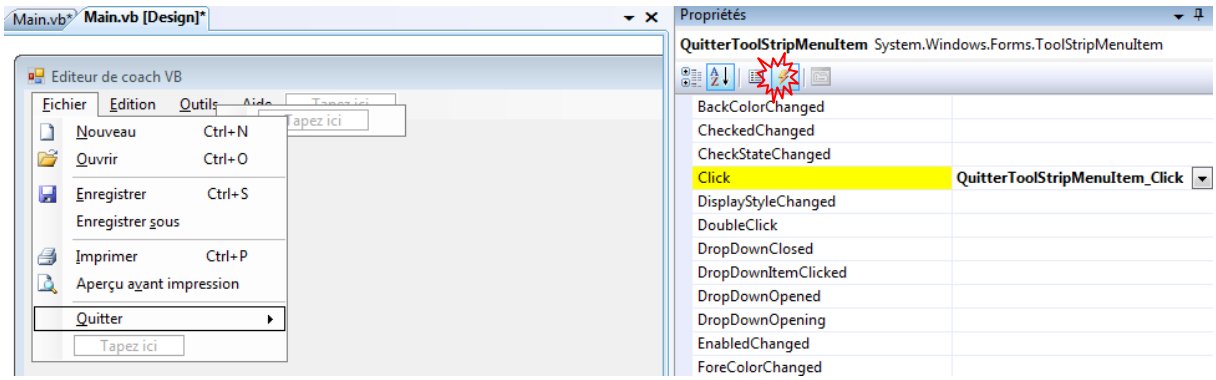
La signature d'une procédure en réponse à un évènement ne peut donc pas être inventée. Lorsque le système exécute les procédures en réponse à un évènement, il ne peut invoquer que des méthodes respectant rigoureusement la signature attendue. C'est sa manière de communiquer avec vous pour vous transmettre des informations pertinentes sur ce qui se passe dans l'application.

Vous retrouvez également tous les évènements disponibles sur un objet à partir de la fenêtre de Propriétés de l'objet en mode Design.

- Afficher le Concepteur de formulaire du fichier **Main.vb**.
- Sélectionnez le menu **Quitter** puis faites un clic droit > **Propriétés** pour afficher la fenêtre de propriétés pour cet objet.



- Dans la fenêtre **Propriétés**, cliquez le bouton de la barre d'outils pour afficher l'ensemble des événements correspondant. Vous devez retrouver le nom de la procédure **quitterToolStripMenuItem_Click** en face de l'événement **Click** indiquant que l'évènement possède un gestionnaire d'évènement :



Il suffit de cliquer sur le bouton de la barre d'outils de la fenêtre **Propriétés** pour revenir à la liste des propriétés de l'objet.

Pour générer automatiquement un gestionnaire en réponse à un évènement, repérez l'évènement dans la liste puis double cliquez dans la zone de texte à droite du nom de l'évènement. Visual Studio bascule dans la fenêtre de code et génère une procédure sur la base du format de nom que nous avons étudié précédemment.

Pour tout savoir sur les évènements et gestionnaires d'évènements :

http://msdn.microsoft.com/fr-fr/library/2z7x8ys3(VS.80).aspx

Pour en savoir plus sur les clauses **WithEvents** et **Handles** :

http://msdn.microsoft.com/fr-fr/library/stf7ebaz(VS.80).aspx

Et enfin, si vous voulez voir comment cela se passe avec l'approche par code, plus souple et dynamique :

http://msdn.microsoft.com/fr-fr/library/6yyk8z93(VS.80).aspx

4. Codez maintenant la fermeture du formulaire :
    - Revenez sur la définition de la procédure **QuitterToolStripMenuItem_Click** dans le fichier **Main.vb**.
    - Ajoutez le code suivant :

Code VB

```
Public Class Main
```

```
    Private Sub QuitterToolStripMenuItem_Click(ByVal sender As System.Object, _
                                               ByVal e As System.EventArgs) _
                Handles QuitterToolStripMenuItem.Click
        'Fermeture du formulaire
        Me.Close()
    End Sub
End Class
```

Que signifie le mot **Me** ?

Le mot clé **Me** référence l'objet dans lequel le code s'exécute au passage de la ligne courante. Dans notre cas, nous développons la classe **Main**. **Me** permet donc de retrouver l'instance de notre classe **Main** qui va être créée au moment de l'exécution du programme.

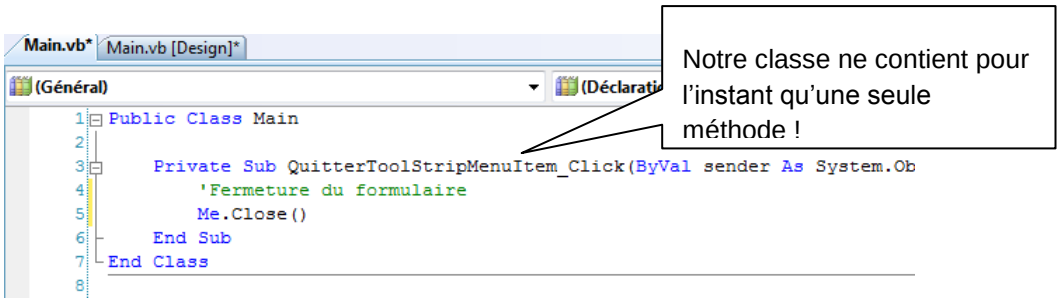
Voici un lien très intéressant pour ne pas confondre **Me** avec d'autres mots clé qui lui ressemblent et qui pourraient donc porter à confusion :

http://msdn.microsoft.com/fr-fr/library/20fy88e0.aspx

Et que signifie **Close()** ?

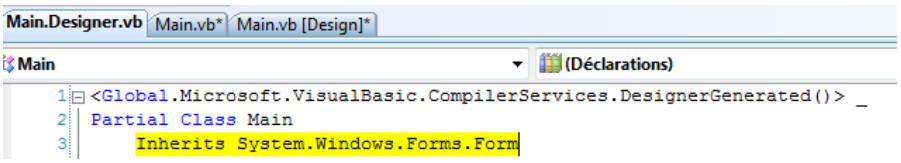
Il s'agit d'une méthode de l'objet référencé par **Me**, c'est-à-dire par l'instance en cours de notre classe **Main**. Et vous vous en doutez, c'est une méthode dont l'action est bien sûr de *fermer (close)* le formulaire.

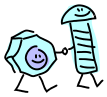
Mais notre classe **Main** n'a pas de méthode appelée **Close** ☹…Alors d'où sort-elle ?



Vous vous rappelez que **Main** est une classe partielle (mot clé *Partial*) c'est-à-dire qu'elle est définie en plusieurs morceaux répartis dans plusieurs fichiers. Peut-être que la méthode que nous cherchons est définie dans l'autre fichier, **Main.Designer.vb** ?

- A partir de l'**Explorateur de solutions**, ouvrez à nouveau le fichier **Main.Designer.vb**.
- Notez la définition de la classe **Main** tout en haut du fichier.





Si vous cherchez la méthode **Close** dans le fichier **Main.Designer.vb**, vous ne la trouverez pas, et pour cause puisque ce fichier ne fait que traduire ce que vous dessinez avec Visual Studio dans le formulaire et nous n'avons rien fait en rapport avec la fermeture du formulaire.

En revanche, la déclaration de la classe en haut du fichier est un peu plus complète que celle que nous avons dans le fichier **Main.vb**. Elle montre notamment que la classe **Main** *hérite* (Inherits) d'une classe du Framework .NET appelée **Form** dans l'espace de nom **System.Windows.Forms**.

Que signifie *l'héritage* ?

L'héritage est une association entre deux classes qui assure l'utilisation par une classe des fonctionnalités déjà définies dans l'autre classe, histoire de ne pas tout réécrire. Nous reviendrons sur ce concept de base de la programmation orienté objet plus tard dans ce tutorial.

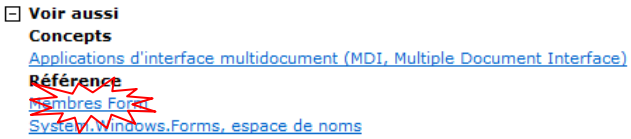
Dans notre cas, retenons que notre classe **Main hérite de la classe Form** fournie par le Framework .NET de façon à hériter de toutes les caractéristiques et du comportement standard d'un formulaire Windows Forms. Sans cet héritage, vous devriez construire le formulaire de A à Z en créant des propriétés pour définir sa taille, son titre etc… et en décrivant son comportement tel que l'ouverture ou la fermeture du formulaire.

Grâce au Framework .NET, nous pouvons donc utiliser une méthode **Close** qui vient de la classe **System.Windows.Form.Form** dans laquelle est codé l'ordre de fermeture du formulaire.

N'oubliez pas que la Library de MSDN vous permet de retrouver toutes les caractéristiques de n'importe quelle classe du Framework .NET. Par exemple, retrouvez la définition de la classe **Form** sur :

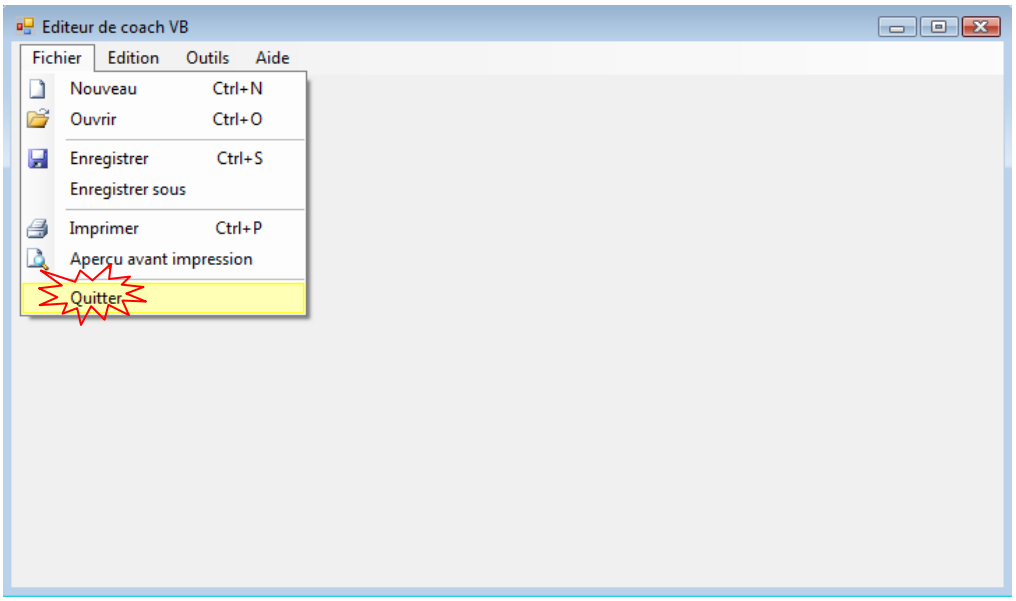
http://msdn.microsoft.com/fr-fr/library/system.windows.forms.form.aspx

Consulter la rubrique **Voir aussi** au bas de la page pour aller sur le lien listant tous les membres de la classe à savoir ses propriétés, ses méthodes et ses évènements. C'est là que vous trouverez la définition de la méthode **Close**.



5. Et si vous testiez votre première ligne de code ?

   - Enregistrez tous vos changements.
   - Lancez l'exécution de l'application (F5).
   - Cliquez le menu **Fichier** > **Quitter**. Le formulaire doit se fermer et dans la foulée, l'exécution de l'application doit s'arrêter.



Bravo ! Vous commencez à avoir une application qui roule ! Elle démarre avec un écran de démarrage et s'arrête proprement avec les menus de la barre de menu.

AFFICHER L'APPLICATION DANS LA ZONE DE NOTIFICATION

Dans cet exercice, vous allez maintenant manipuler un *composant* Windows Form. Il s'agit du composant **NotifyIcon** qui affiche une icône dans la zone de notification d'état de la barre des tâches de Windows. C'est un composant très utile par exemple pour contrôler des applications qui s'exécutent en arrière-plan et qui n'ont pas d'interface utilisateur.

Contexte fonctionnel

Dans cet exercice, nous allons rajouter une icône dans la zone de notification d'état de la barre des tâches de Windows en bas à droite du bureau, qui atteste que notre application est en cours d'exécution. Lorsque l'utilisateur clique sur l'icône, un menu contextuel lui propose des options standards lui permettant de quitter l'application et de redimensionner la fenêtre principale.

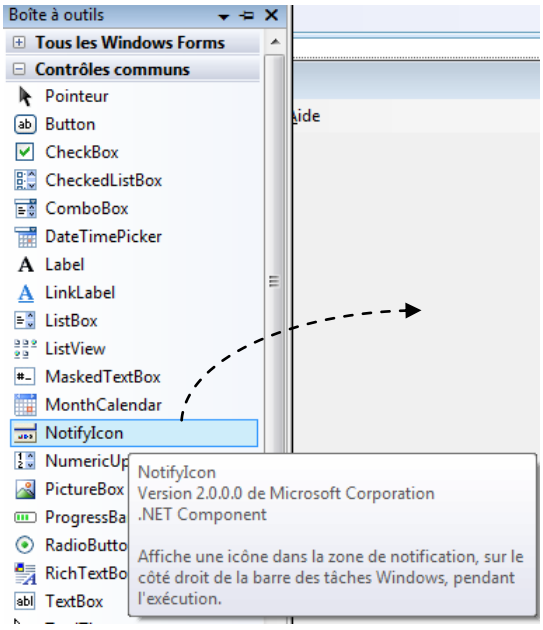
Le procédé s'articule en quatre étapes :

- Ajout d'une icône de notification au programme, qui s'affiche dans la zone de notification de Windows au moment de l'exécution.

- Ajout d'un menu contextuel au programme.
- Association du menu contextuel au clic sur l'icône de notification.
- Codage de l'option Quitter du menu contextuel pour fermer le formulaire.

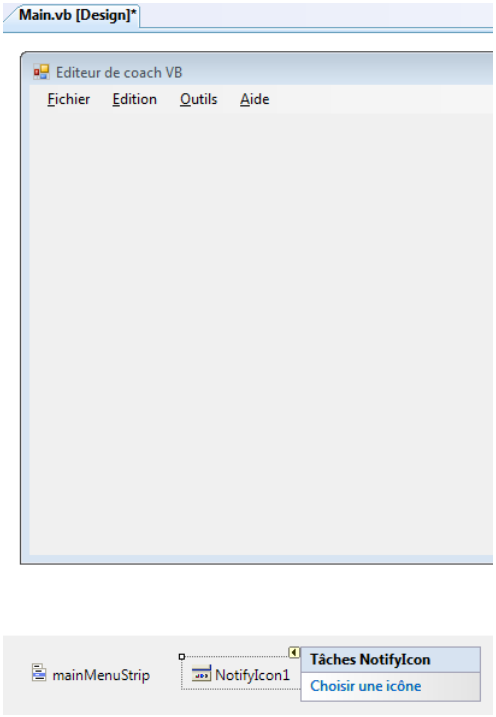
Déroulement de l'exercice :

1. *Etape 1* : ajout d'une icône de notification à l'application :
    - Ouvrez le formulaire **Main.vb** en mode Design en double-cliquant sur le fichier **Main.vb** dans l'**Explorateur de solutions**.
    - Faites un glisser déplacer de la Boite à outils, rubrique **Contrôles communs** > du composant **NotifyIcon** n'importe où sur la surface du formulaire.



Comme il s'agit d'un composant, c'est-à-dire qu'il n'a pas d'équivalence graphique directement sur le formulaire, Visual Studio nous l'affiche directement dans la zone de dépôt de contrôles en dessous du formulaire, sous la forme d'un objet nommé **NotifyIcon1**.

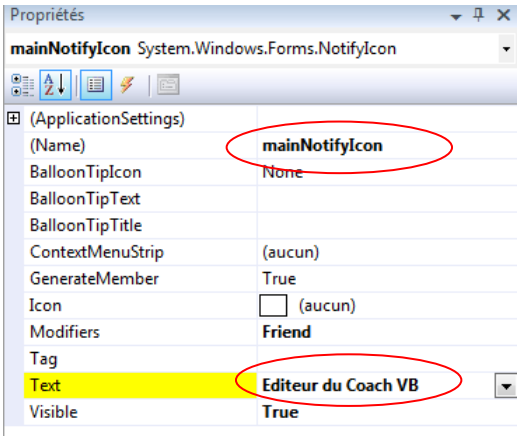
Au travers d'une balise active (smart tag) qui apparaît en haut à droite du contrôle lorsque celui-ci est sélectionné, Visual Studio nous propose des raccourcis vers les actions de configuration type pour paramétrer le plus rapidement possible ce contrôle. En l'occurrence ici, il faut principalement lui associer une icône pour l'afficher dans la zone de notification d'état de la barre des tâches de Windows. Mais une fois n'est pas coutume, nous n'allons pas utiliser cette approche déclarative et opter pour une configuration de l'icône de notification par code en utilisant les ressources du projet ☺.



Pour tout savoir sur le composant Windows Forms **NotifyIcon** :

http://msdn.microsoft.com/fr-fr/library/7yyz6s5c(VS.80).aspx

- Afficher les propriétés du composant par un clic droit > **Propriétés**.
- Modifiez le nom du composant 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 de la fenêtre Propriétés).
- Modifiez également la propriété **Text** avec la valeur **Editeur du Coach VB.** Ce texte apparaîtra en aide rapide (« tooltip ») lorsque le pointeur de souris sera au dessus de l'icône.



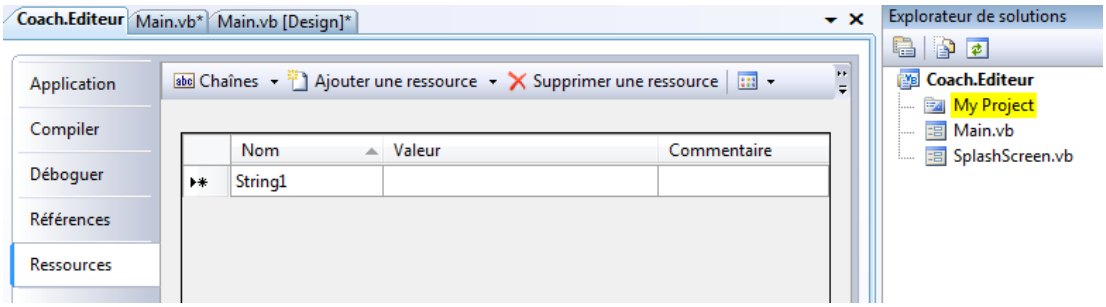
Notez au passage dans cette fenêtre **Propriétés** que la propriété **Icon** est celle qui va nous permettre d'associer une icône au composant, celle-là même qui apparaîtra dans la zone de notification.

Comment configurer l'icône par code ?

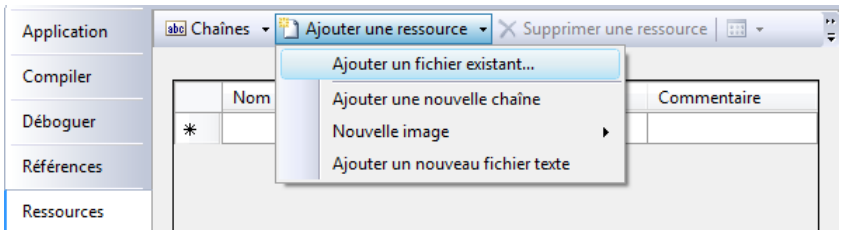
Les images, les icônes ou encore les fichiers audio sont ce qu'on appelle des *ressources* du programme. Nous allons récupérer une icône et la lier au projet en utilisant le **Concepteur de ressources** du projet puis nous paramétrerons la propriété **Icon** du contrôle **mainNotifyIcon** avec la ressource ajoutée.

2. Configurez l'icône du composant de notification avec un fichier icône fourni avec le code de l'atelier :

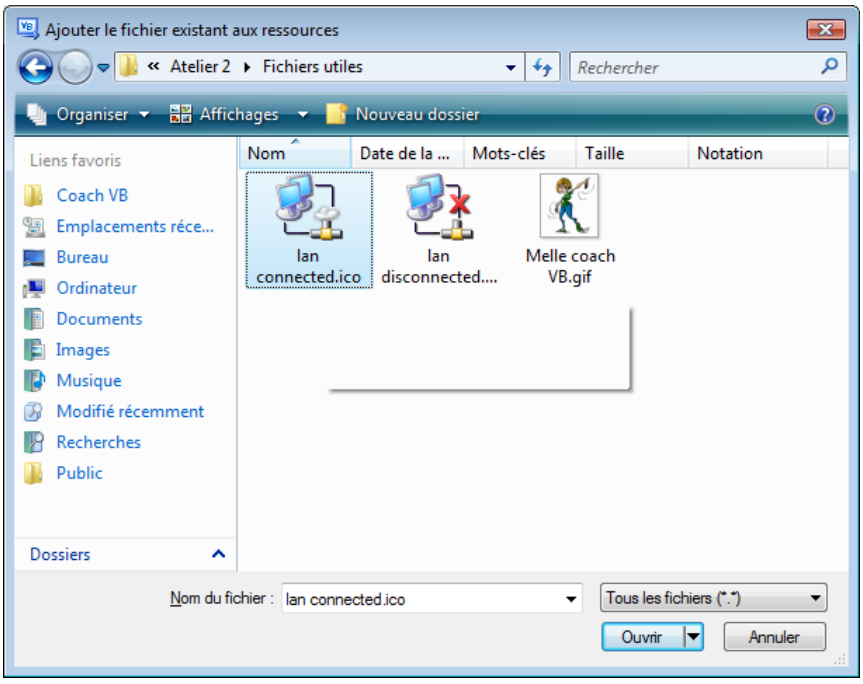
    - A partir de l'**Explorateur de solutions**, affichez le Concepteur de projets en cliquant sur **My Project**.
    - Sélectionnez l'onglet **Ressources**.



- Sur l'écran de gestion des ressources, cliquez le menu **Ajouter une ressource > ajouter un fichier existant…** :

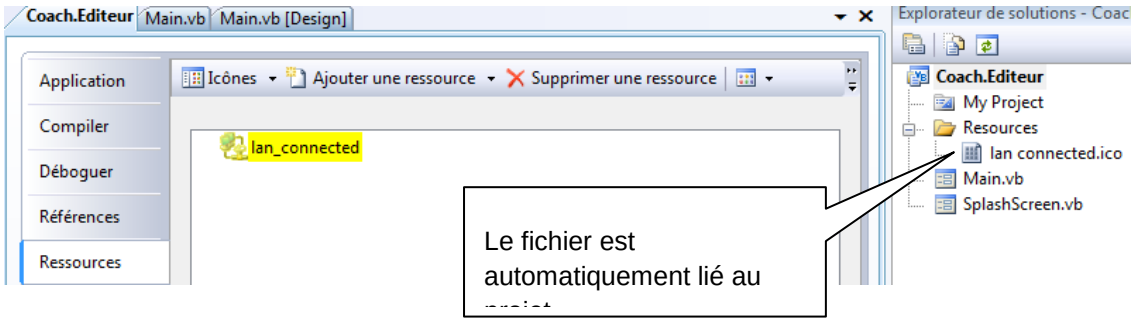


- Dans la boîte de dialogue **Ajouter le fichier existant aux ressources**, naviguez jusqu'au répertoire **..\Atelier 2\Fichiers utiles** fourni avec cet atelier.
- Sélectionnez le fichier **lan_connected.ico** puis cliquez le bouton **Ouvrir**.



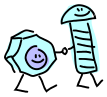
Nous avons choisi à titre d'exemple une icône représentant une connexion LAN . On pourrait en effet imaginer que votre application ait besoin à un moment ou un autre d'être connectée à Internet ou sur le réseau de l'entreprise. Dans ce cas, il pourrait être intéressant de détecter par programmation l'état de connexion de la machine de l'utilisateur et de modifier l'icône affichée par le composant **mainNotifyIcon** en conséquence, afin de donner à l'utilisateur une indication sur la connectivité de l'application en cours. L'icône correspondrait par exemple à l'état déconnecté.

Nous n'allons charger qu'une seule icône dans le projet à ce stade du tutorial.



- Fermez le Concepteur de ressources du projet.
- Enregistrez vos modifications sur le projet.

Il faut maintenant charger la propriété **Icon** du composant **mainNotifyIcon** avec la ressource que nous venons d'ajouter au programme.



Où doit-on brancher le code de configuration de la propriété **Icon** du composant ?

La question serait plutôt *quand* puisque nous développons avec une approche évènementielle. L'idéal serait d'agir au tout début de la vie du formulaire pour que l'icône apparaisse dans la zone de notification de Windows dès le démarrage de l'exécution.

Quel est le tout premier évènement disponible sur un objet ?

Il s'agit de son **constructeur**. Le constructeur est une méthode membre d'une classe qui est appelée par le système au moment de la *construction* de l'objet. On dit aussi que le nouvel objet est *instancié*. Du coup, cette méthode sert à créer convenablement une instance d'objet pour la classe, en le configurant dans un état valide.

Un constructeur en VB est une méthode publique nommée **New** sans aucune valeur de retour.

- Ouvrez le fichier de code **Main.vb**.
- Entrez directement sous la définition de la classe **Main** la ligne suivante :

Code VB

```
Public Class Main
```

```
    Public Sub New

    Private Sub QuitterToolStripMenuItem_Click(ByVal sender As System.Object, _
                                               ByVal e As System.EventArgs) _
                Handles QuitterToolStripMenuItem.Click
        …
    End Sub
End Class
```

- Validez par **Entrée**. Visual Studio rajoute la suite de la procédure pour vous ☺ !

Code VB

```
Public Class Main

    Public Sub New()

        ' Cet appel est requis par le Concepteur Windows Form.
        InitializeComponent()

        ' Ajoutez une initialisation quelconque après l'appel
        ' InitializeComponent().

    End Sub
```

```
    Private Sub QuitterToolStripMenuItem_Click(ByVal sender As System.Object, _
                                               ByVal e As System.EventArgs) _
                Handles QuitterToolStripMenuItem.Click
        ...
    End Sub
End Class
```

Notez entre autres l'appel à la méthode **InitializeComponent** que vous trouverez dans le fichier **Main.Designer.vb**. Il s'agit de la procédure qui initialise tous les éléments de votre interface et qui est donc appelée systématiquement à la construction du formulaire. D'habitude, on ne s'en préoccupe guère parce que Visual Studio créait le formulaire pour nous. Mais comme nous sommes en train de définir un constructeur spécifique pour le formulaire **Main**, c'est à nous maintenant d'appeler cette méthode !

- Dans le constructeur de la classe, ajoutez maintenant le code pour initialiser l'icône du contrôle **mainNotifyIcon** avec celui chargé dans les ressources du projet.

Code VB

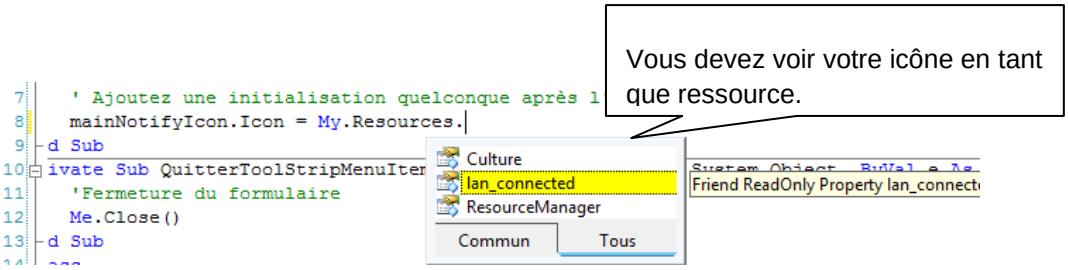
```
Public Class Main

    Public Sub New()

        ' Cet appel est requis par le Concepteur Windows Form.
        InitializeComponent()
```

```
        ' Ajoutez une initialisation quelconque après l'appel
        ' InitializeComponent().
        mainNotifyIcon.Icon = My.Resources.lan_connected

    End Sub
    …
End Class
```

Vous vous souvenez de l'objet **My** ?

C'est un espace de noms qui contient des classes donnant des raccourcis vers les fonctionnalités les plus couramment utilisées du Framework. **My.Resources** donne un accès simple et rapide aux ressources de l'application.

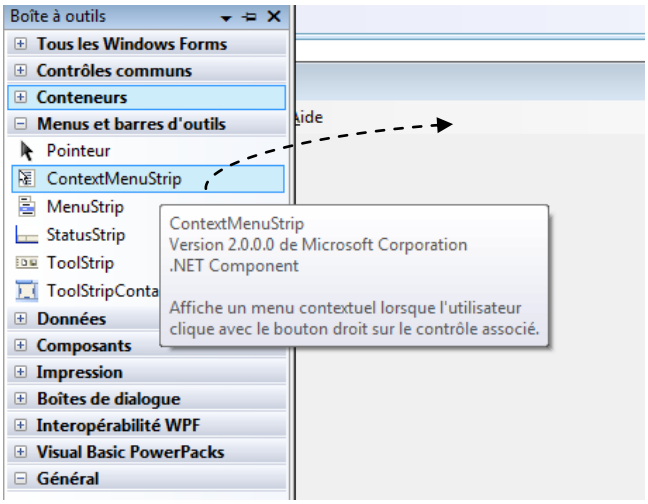
N'oubliez pas d'utiliser à fond l'aide de l'**IntelliSense** pour éviter les erreurs. Par exemple, à chaque fois que vous tapez un point, l'IntelliSense se manifeste et vous guide dans le choix des éléments possibles compte tenu du contexte.



3. Testez le fonctionnement de l'icône de notification de l'application :
    - Enregistrez tous les changements.

- Lancez l'exécution de l'application (F5) pour tester le fonctionnement de l'icône de notification.
- Vérifiez la présence de l'icône de connexion dans la zone de notification d'état de la barre des tâches de Windows.
- Arrêtez le curseur de la souris sur l'icône pour valider le texte d'aide (tooltip).

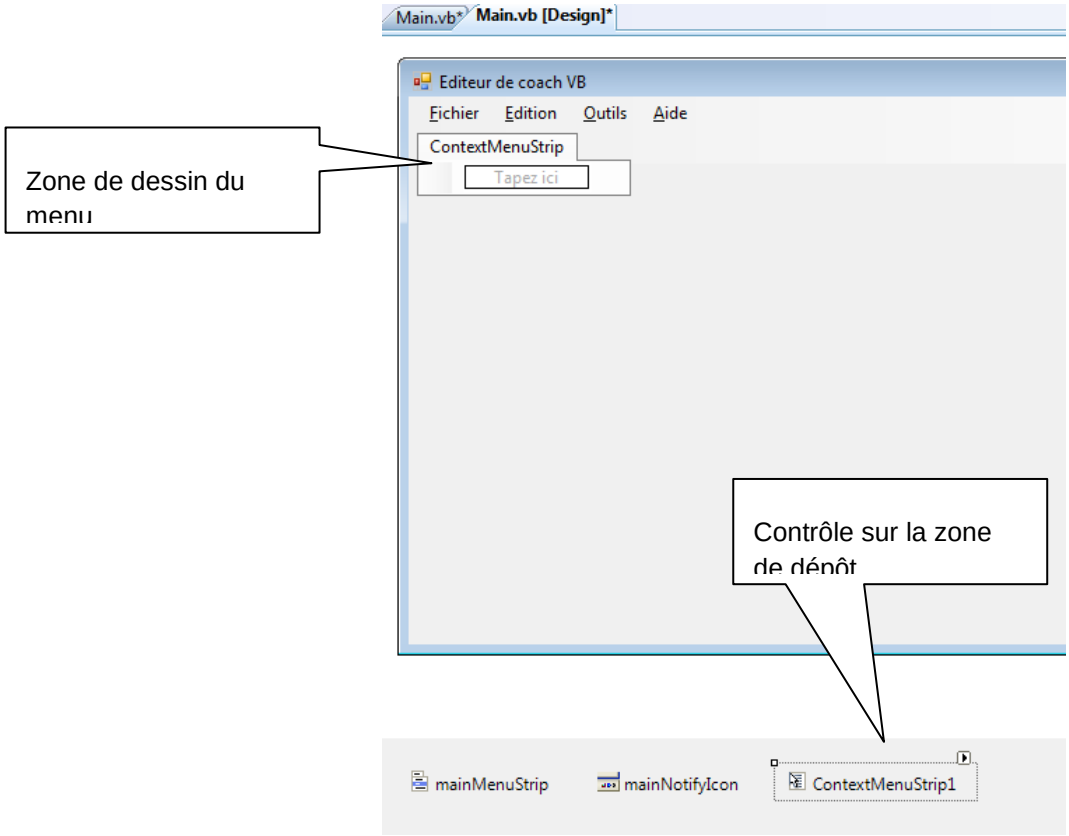
4. *Etape 2* : ajout d'un menu contextuel au programme :
   - Revenez sur le formulaire **Main.vb** en mode Design.
   - Faites un glisser déplacer de la **Boîte à outils** > rubrique **Menus et barre d'outils** > du contrôle **ContextMenuStrip** n'importe où sur la surface du formulaire :



Deux nouveaux éléments apparaissent sur le formulaire :

- Une barre de menu vide sous la barre de titre de la fenêtre,

- Et un composant nommé **contextMenuStrip1** dans la zone de dépôt de contrôles.



Vous observez un comportement rigoureusement identique à celui du contrôle **MenuStrip**, à ceci près qu'à l'exécution un menu contextuel de type **ContextMenuStrip** n'apparaît que dans le *contexte* pour lequel il est défini (d'où son nom). Donc le positionnement du menu juste en dessous du menu principal du formulaire ne fait pas foi. Mais il faut bien que Visual Studio vous l'affiche quelque part proprement pour vous permettre de le construire ☺.

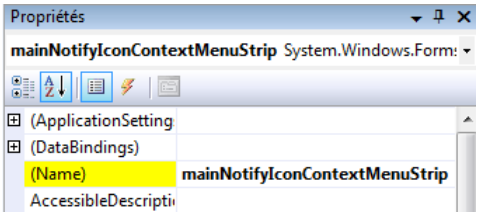
Dans notre cas, nous allons l'associer au composant **mainNotifyIcon** pour qu'il apparaisse sur le clic droit de l'icône dans la zone de notification d'état de la barre des tâches de Windows.

Pour tout savoir sur le contrôle Windows Forms **ContextMenuStrip** :

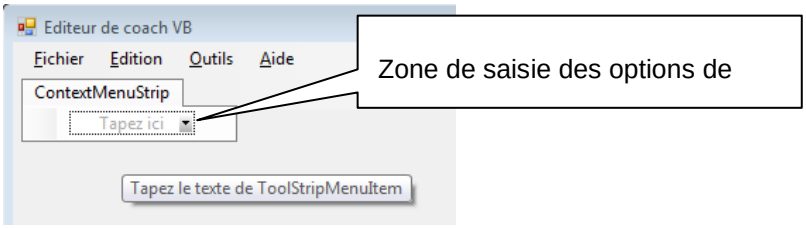
http://msdn.microsoft.com/fr-fr/library/ms229641(VS.80).aspx

5. Configurez les options du menu contextuel :

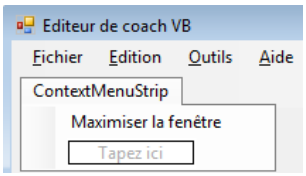
   - Sélectionnez le contrôle **contextMenuStrip1** dans la zone de dépôt de contrôles et faites un clic droit > **Propriétés** pour faire apparaître sa fenêtre de propriétés.
   - Dans les propriétés du contrôle, changez son nom par **mainNotifyIconContextMenuStrip** en changeant la valeur de la propriété **(Name)**.



   - Sur la zone de dessin du menu contextuel, cliquez sur **Tapez ici** pour saisir une première option de menu :

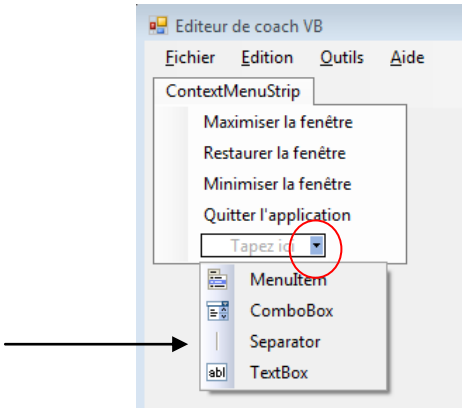


   - Saisissez **Maximiser la fenêtre** puis validez par **Entrée** :

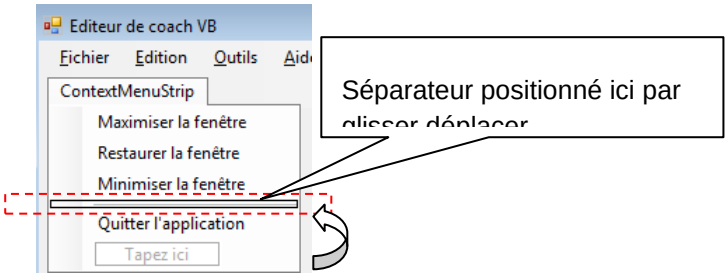


- Recommencez l'opération avec les options **Restaurer la fenêtre**, **Minimiser la fenêtre**, et **Quitter l'application** :

- Cliquez sur la flèche qui apparaît à droite de la zone Tapez ici (lorsque vous la sélectionner) en dessous de l'option **Quitter l'application**. Dans la liste déroulante, sélectionnez **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** :

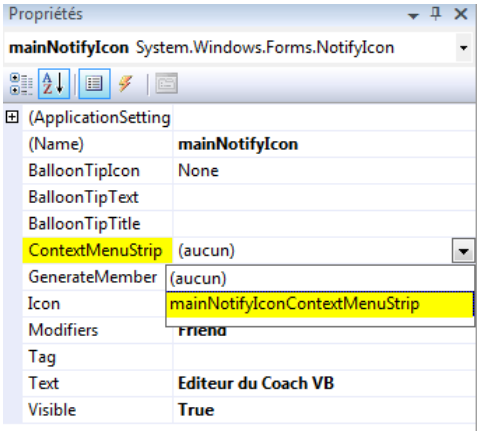


- Enregistrez vos changements.

6. *Etape 3* : association du menu contextuel au clic sur l'icône de notification :

   - Sélectionnez le contrôle **mainNotifyIcon** dans la zone de dépôt de contrôles, puis faites un clic droit > **Propriétés** ;

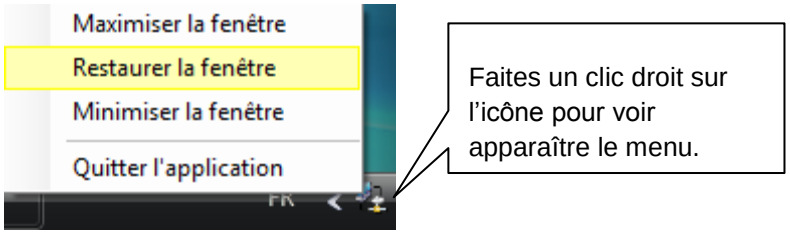
   - Dans les propriétés du contrôle, changez la valeur de **ContextMenuStrip** en utilisant la liste déroulante proposée, et selectionnez **mainNotifyIconContextMenuStrip** :



7. Testez le fonctionnement de l'icône de notification de l'application :

   - Enregistrez tous les changements.

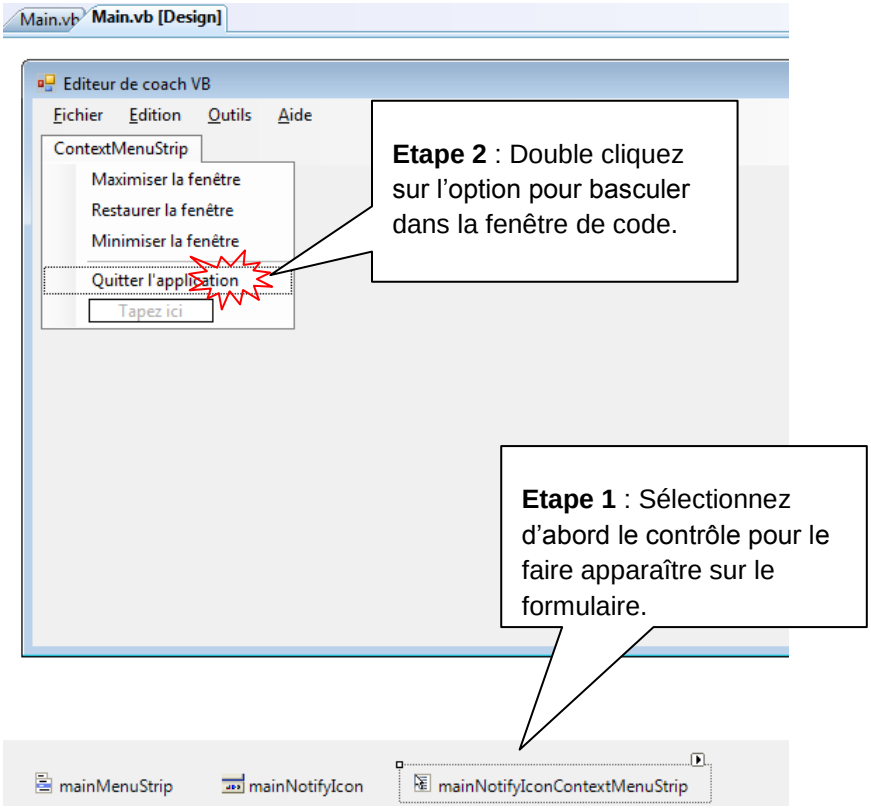
- Lancez l'exécution de l'application (F5).
- Vérifiez la présence de l'icône de connexion dans la zone de notification d'état de la barre des tâches de Windows.
- Faites un clic droit sur l'icône pour valider le déclenchement du menu contextuel :



L'étape suivante consiste à coder maintenant les actions associées au clic de l'utilisateur sur les différentes options du menu contextuel. Dans cet atelier, nous allons nous concentrer sur l'option **Quitter l'application** qui permet à l'utilisateur d'arrêter l'exécution du programme.

Le procédé de codage d'une option de menu contextuel est rigoureusement le même que celui que nous avons suivi précédemment pour coder l'option **Quitter** du menu principal du formulaire **Main**.

8. *Etape 4* : codage de l'option Quitter l'application :
   - Sélectionnez le contrôle **mainNotifyIconContextMenuStrip** dans la zone de dépôt de contrôles afin de le faire apparaître sur le formulaire.
   - Double cliquez sur l'option du menu contextuel **Quitter l'application** pour générer la procédure de réponse à l'évènement **Click** sur l'option de menu :



Visual Studio bascule dans la fenêtre de code et génère à la suite des membres de la classe **Main**, une nouvelle méthode nommée **QuitterLapplicationToolStripMenuItem_Click**.

```
15      Private Sub QuitterLapplicationToolStripMenuItem_Click(ByVal sender As System.Object, _
16                                                             ByVal e As System.EventArgs) _
17                                             Handles QuitterLapplicationToolStripMenuItem.Click
18
19      End Sub
20  End Class
```

Nom de l'évènement associé au gestionnaire d'évènement.

- Ajoutez le code de fermeture du formulaire :

Code VB

```
Private Sub QuitterLapplicationToolStripMenuItem_Click( _
                                        ByVal sender As System.Object, _
                                        ByVal e As System.EventArgs) _
                                Handles QuitterLapplicationToolStripMenuItem.Click
        'Fermeture du formulaire
        Me.Close()
End Sub
```

Ca ne vous rappelle rien ce code ? C'est évidemment rigoureusement le même que celui que nous avons programmé dans la procédure associée au clic de l'option **Quitter** du menu principal du formulaire.

Ce qui nous amène à la question suivante :

Est-ce qu'on ne pourrait pas récupérer le code du gestionnaire d'évènement **QuitterToolStripMenuItem_Click** pour l'appliquer à l'évènement **Click** sur l'option **Quitter l'application** du menu contextuel ?

La réponse est oui, vous vous en doutez. Mais comment ?

C'est le mot clé **Handles** qui détermine sur quel évènement s'abonne le gestionnaire d'évènement correspondant. Alors pourquoi ne pas ajouter un nouvel évènement au gestionnaire **QuitterToolStripMenuItem_Click** que nous avions écrit pour l'option **Quitter** du menu principal, pour qu'il prenne en charge aussi le clic de l'utilisateur sur l'option **Quitter l'application** du menu contextuel de l'icône de notification. Il suffit de le brancher également sur l'évènement **Click** de l'objet **QuitterLapplicationToolStripMenuItem**.

- Reprenez le gestionnaire d'évènement **QuitterToolStripMenuItem _Click** et ajoutez-lui le second évènement séparé par une virgule.
- Supprimez la méthode **QuitterLapplicationToolStripMenuItem_Click** qui ne sert plus à rien.

```
10    Private Sub QuitterToolStripMenuItem_Click(ByVal sender As System.Object, _
11                                               ByVal e As System.EventArgs) _
12            Handles QuitterToolStripMenuItem.Click, QuitterLapplicationToolStripMenuItem.Click
13        'Fermeture du formulaire
14        Me.Close()
15    End Sub
16
17    Private Sub QuitterLapplicationToolStripMenuItem_Click(ByVal sender As System.Object, _
18                                                           ByVal e As System.EventArgs) _
19                                       Handles QuitterLapplicationToolStripMenuItem.Click
20
21    End Sub
```

Une bonne pratique serait de renommer le gestionnaire d'évènement avec un nom plus générique qui refléterait son action.

- Renommez par exemple la méthode **QuitterToolStripMenuItem_Click** en **FormClose**.

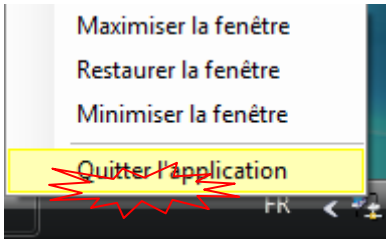
Main.vb | Main.vb [Design]

QuitterLapplicationToolStripMenuItem | Click

```
Public Class Main
    Public Sub New()

        ' Cet appel est requis par le Concepteur Windows Form.
        InitializeComponent()

        ' Ajoutez une initialisation quelconque après l'appel InitializeComponent().
        mainNotifyIcon.Icon = My.Resources.lan_connected
    End Sub
    Private Sub FormClose(ByVal sender As System.Object, _
                                        ByVal e As System.EventArgs) _
            Handles QuitterToolStripMenuItem.Click, QuitterLapplicationToolStripMenuItem.Click
        'Fermeture du formulaire
        Me.Close()
    End Sub


End Class
```

9. Testez le clic sur l'option de menu contextuel associé à l'icône de notification :

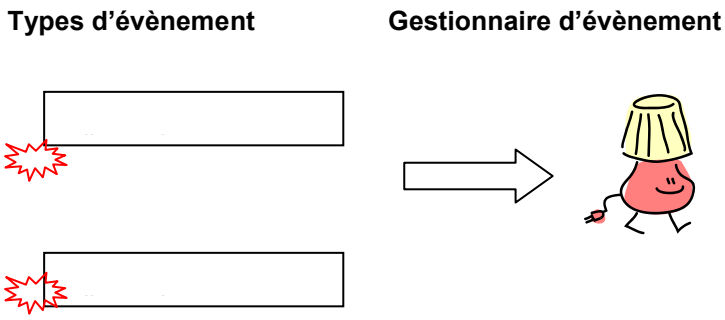
   - Enregistrez tous les changements.
   - Lancez l'exécution de l'application (F5).
   - Faites un clic droit sur l'icône pour faire apparaître le menu contextuel.
   - Vérifiez que l'application s'arrête proprement en cliquant l'option **Quitter l'application** du menu contextuel.



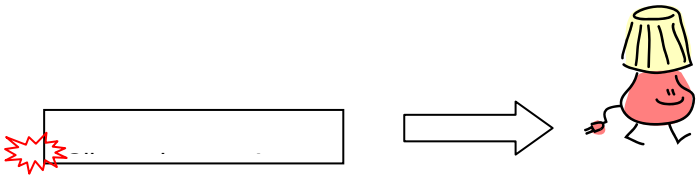
   - Relancez l'exécution de l'application (F5).
   - Vérifiez la fermeture du formulaire sur un clic de l'option **Quitter** du menu principal.

En conclusion, avec les clauses **WithEvents** et **Handles**, il est possible d'autoriser un gestionnaire d'évènements à gérer un ou plusieurs types d'évènements.

C'est un peu comme dans une maison, souvent dans une même pièce vous avez plusieurs boutons qui provoquent l'allumage d'un même jeu de plafonnier.



A l'inverse, un ou plusieurs gestionnaires d'évènements peuvent être configurés pour gérer un même type d'évènements. C'est-à-dire que le clic d'u uton de la pièce peut allumer le plafonnier de la pièce et celui également du hall voisi



Bravo ! Vous avez développé votre première application Visual Basic de type Windows. Dans les prochains ateliers, nous allons poursuivre sa construction tout en explorant tous les principes de programmation et caractéristiques liés au langage Visual Basic.

POUR ALLER PLUS LOIN…

Il existe d'autres alternatives aux différents procédés illustrés dans cet atelier. Par exemple, une autre alternative pour exécuter et fermer une application consiste à utiliser l'objet **Application** du Framework .NET et ses méthodes **Run** et **Exit** :

Pour Application.Run() :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.application.run(VS.80).aspx

Pour Application.Exit() :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.application.exit(VS.80).aspx

Mais attention, si deux procédés peuvent paraître identiques du point de vue de l'utilisateur, ils peuvent suivre un cheminement très différent d'un point de vue cycle d'exécution dans l'application. Par exemple, l'utilisation de la méthode **Close** d'un formulaire déroule le cycle de fermeture complet du dit formulaire c'est-à-dire déclenche une succession d'évènements caractéristiques de la fermeture d'une fenêtre Windows Form. En revanche, l'utilisation de la méthode **Exit** de l'objet **Application** a pour effet de fermer tous les formulaires ouverts dans l'application mais sans déclencher le cycle complet de fermeture de chacun d'eux.

Tout cela pour dire que le Framework recèle de nombreuses possibilités adaptées à chaque scénario. Aidez-vous de MSDN pour bien comprendre le domaine d'application de chacun des éléments que vous mettez en œuvre.

Utiliser les structures du langage et les types de base

http://msdn.microsoft.com/vbasic

SOMMAIRE

INTRODUCTION **3**

CONTEXTE FONCTIONNEL 3

UTILISER LES TYPES DE BASE **5**

CRÉER UNE BOÎTE DE DIALOGUE OPTIONS 6

UTILISER LA BOÎTE DE DIALOGUE STANDARD FOLDERBROWSERDIALOG 40

UTILISER LES TYPES DE DONNÉES COMMUNS 45

UTILISER LES STRUCTURES DU LANGAGE **60**

STRUCTURES DE DÉCISION 60

STRUCTURES DE CONTRÔLE 67

STRUCTURES DE BOUCLE 75

POUR ALLER PLUS LOIN… **106**

TRAITEMENT DU MENU CONTEXTUEL DE L'ICÔNE DE NOTIFICATION 106

ECRIRE DANS LE JOURNAL DE WINDOWS 111

INTRODUCTION

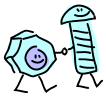
CONTEXTE FONCTIONNEL

Rappel du contexte fonctionnel du tutorial du coach VB

L'objectif du tutorial du Coach VB est d'accompagner les développeurs à la découverte et la prise en main du langage Visual Basic (VB) pour la construction d'applications avec une approche orientée objet.

Pour rappel, vous pouvez repérer facilement deux caractéristiques importantes du langage à l'aide des logos suivants en marge :

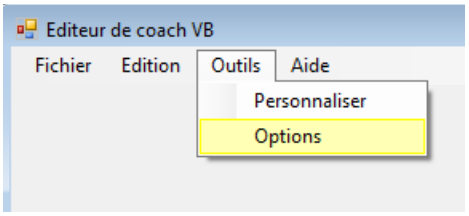
Ce logo marque une fonctionnalité de VB ou de Visual Studio qui permet de développer vite (et juste ☺).



Ce logo met en évidence une caractéristique de la programmation orientée objet.

Contexte fonctionnel du troisième atelier

Dans ce troisième atelier, vous allez ajouter au projet une boîte de dialogue permettant à l'utilisateur de configurer les options de l'application. Vous savez, c'est cette boîte du menu **Outils** > **Options** que l'on trouve dans tous les produits Microsoft.

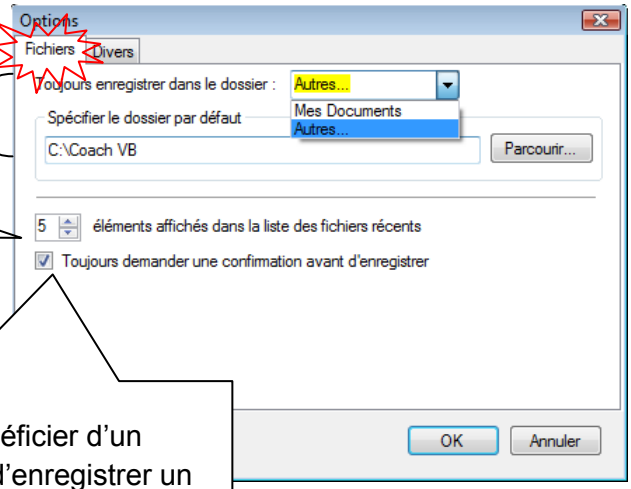


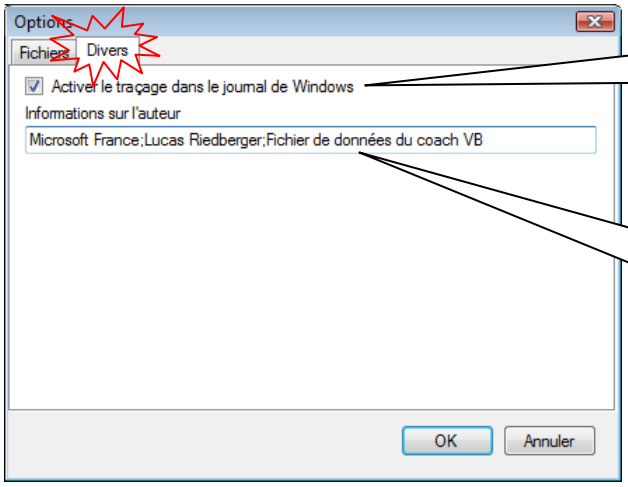
Il s'agit d'une boîte de dialogue proposant des options sur deux onglets **Fichiers** et **Divers** :

Option de configuration du dossier de sauvegarde par défaut proposé au moment de la sauvegarde des fichiers de données que nous

Option de configuration du nombre de fichiers dans la liste des fichiers récents proposés à

Option de configuration pour bénéficier d'un message de confirmation avant d'enregistrer un



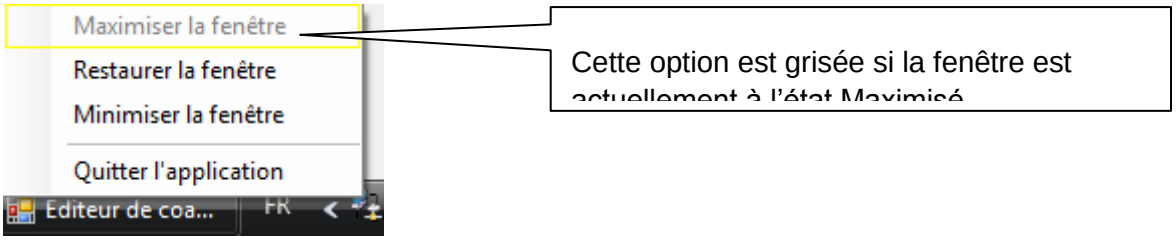


Option de configuration pour activer la fonction de traçage dans le journal des évènements de

Option de configuration des propriétés à enregistrer avec les fichiers de données donnant l'auteur du document, le titre etc…(chaque

Ces informations de configuration devront être préservées pendant la durée de l'exécution de l'application mais seront perdues à la fin de celle-ci.

Dans la dernière partie **Pour aller plus loin** de cet atelier, nous aborderons l'écriture d'un message dans le journal de Windows et nous verrons comment programmer les actions associées au menu contextuel de l'icône de notification de l'application que nous avons élaboré à l'atelier précédent. Elles permettent de retailler la fenêtre principale de l'application et sont grisées dynamiquement en fonction du contexte.



Contexte technique

Dans cet atelier nous allons nous concentrer sur l'apprentissage des principales structures et types de données du langage VB. C'est la première étape pour apprendre à développer avec le langage !

A la fin de cet atelier, vous saurez comment :

- Créer une énumération, une constante et utiliser les types de données élémentaires,
- Créer une variable en contrôlant sa portée et sa durée de vie,
- Ecrire une procédure,
- Utiliser les principales structures du langage telles que **If…Then…Else** (structure de décision) ou **For…Next** (structure de boucle), ou encore **With** (autre structure),
- Créer et afficher une boîte de dialogue personnalisée ou utiliser une boîte de dialogue standard de Windows,
- Ecrire dans le journal de Windows.

La solution de cet atelier est disponible dans le répertoire **..\Atelier 3\Solution**. Les fichiers utiles, auxquels font référence les exercices sont disponibles dans le répertoire **..Atelier 3\Fichiers utiles**.

UTILISER LES TYPES DE BASE

Dans cet exercice, vous allez apprendre à :

- Utiliser une boîte de dialogue standard de Windows,
- Créer une boîte de dialogue personnalisée,
- Dessiner des onglets,
- Utiliser les types de données élémentaires de Visual Basic,
- Utiliser une énumération et définir une énumération personnalisée,
- Définir et utiliser une constante,
- Définir une variable en contrôlant sa portée et sa durée de vie.

Objectif

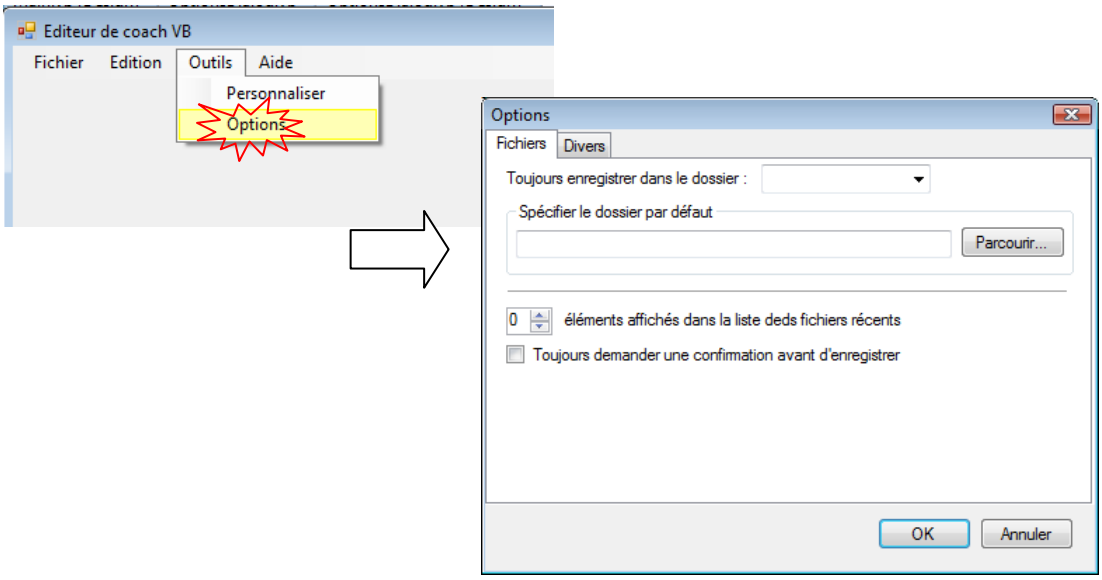
L'objectif de ce premier exercice est de se familiariser avec tous les types de données standard du langage Visual Basic. Au passage, nous en profiterons pour apprendre à manipuler les boîtes de dialogue.

CRÉER UNE BOÎTE DE DIALOGUE OPTIONS

Dans cet exercice, vous allez créer une boîte de dialogue personnalisée.

Contexte fonctionnel

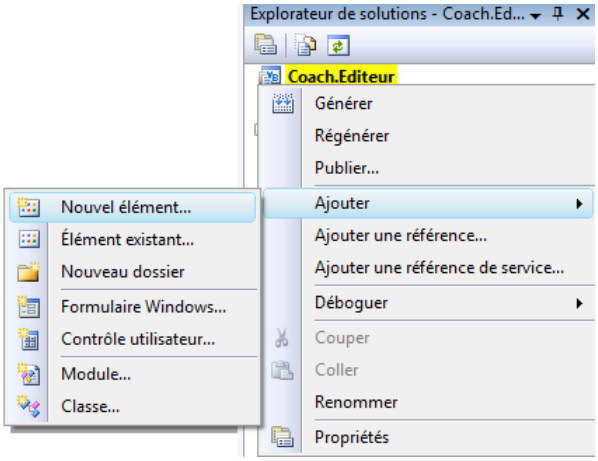
L'objectif est de créer la boîte de dialogue *Options* du programme qui s'affiche lorsque l'utilisateur sélectionne l'option de menu **Outils** > **Options** dans la barre de menu principal. Elle présentera toutes les options de l'application configurables par l'utilisateur.



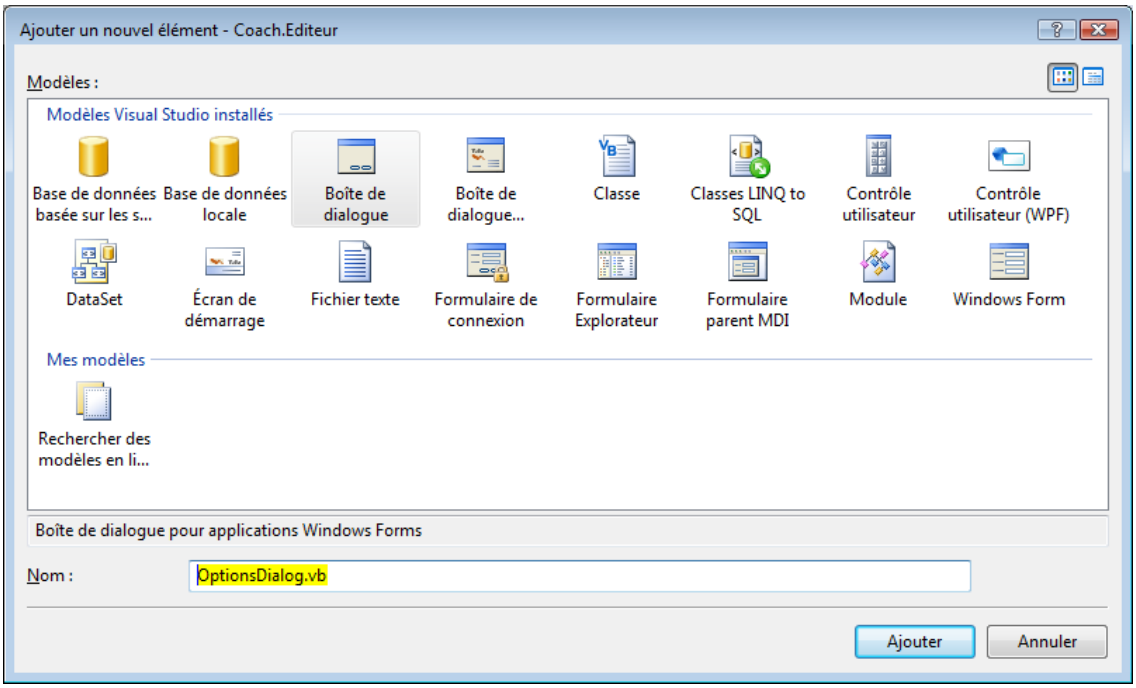
Déroulement de l'exercice :

1. Ouvrez le projet précédent réalisé lors de l'atelier 2 :
   - Lancez Visual Studio à partir du menu **Démarrer** > **Tous les programmes** > **Microsoft Visual Basic 2008 Express Edition**.
   - Menu **Fichier** > **Ouvrir un projet**.
   - Retrouvez le fichier **Atelier 2.sln** que vous avez créé lors de l'atelier 2 ou, si vous n'avez pas fait l'atelier précédent, récupérez le projet de solution dans le répertoire : **..\Atelier 3\Démarrage\Atelier 3.sln**.
2. Ajoutez une boîte de dialogue au projet :

- Dans l'**Explorateur de solutions**, faites un clic droit à la racine du projet > **Ajouter** > **Nouvel élément…**

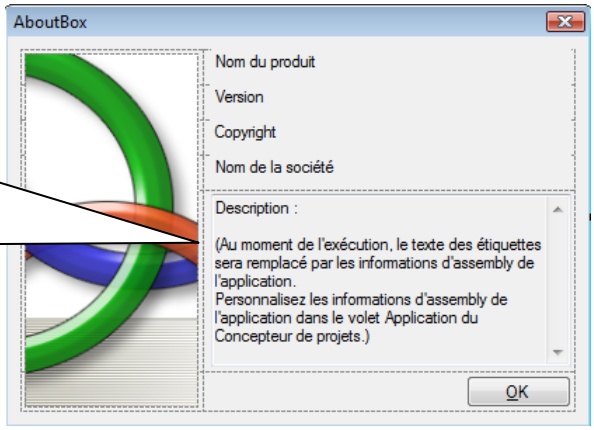


- Sélectionnez le modèle d'élément **Boîte de dialogue**.
- Nommez le fichier par exemple **OptionsDialog.vb**.



Notez qu'il y a un autre modèle de boîte de dialogue que celui que nous allons utiliser. Il s'agit du modèle intitulé **Boîte de dialogue A propos de**, qui (comme son nom l'indique) permet de créer rapidement une boîte **A propos de** dans le projet ressemblant à ceci :

Comme pour la fenêtre **SplashScreen** de l'atelier 2, les informations sont chargées à partir des propriétés d'assembly de l'application configurées avec le **Concepteur de projets**.



- Cliquez **Ajouter**.

En quoi consiste une boîte de dialogue ?

Une boîte de dialogue est une fenêtre qui, comme son nom l'indique, permet de *dialoguer* avec l'utilisateur :

- Elle peut se contenter d'afficher des informations à l'attention de l'utilisateur : par exemple c'est ce que fait une boîte de dialogue *A propos de*.
- Elle peut aussi demander des informations à l'utilisateur, comme cela va être le cas pour notre boîte *Options*.

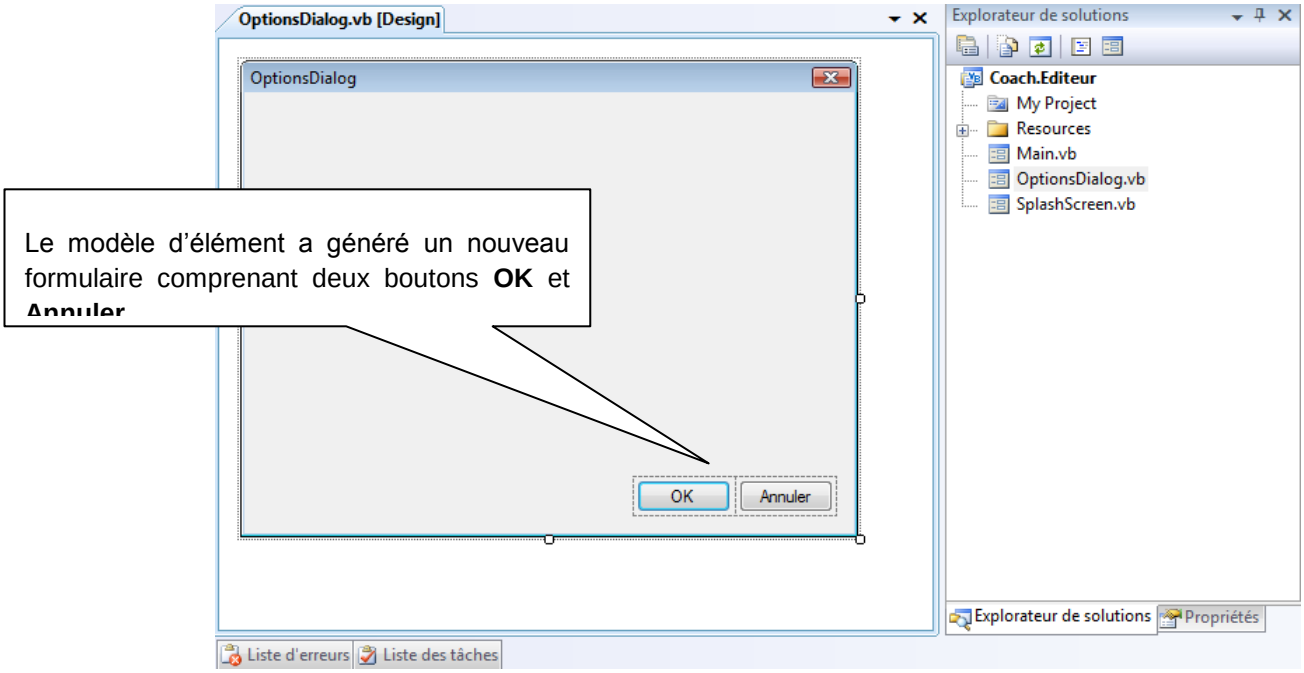
En réalité, ce n'est ni plus ni moins un formulaire Windows Form qu'il faut *aménager* pour répondre au besoin d'échange instantané avec l'utilisateur.

Par exemple, une boîte de dialogue est une fenêtre de taille fixe, non redimensionnable et comporte en général des boutons :

- typiquement un bouton **OK** pour quitter la fenêtre de manière à sauvegarder les

informations saisies par l'utilisateur,

- et un bouton **Annuler** pour au contraire ne rien enregistrer.



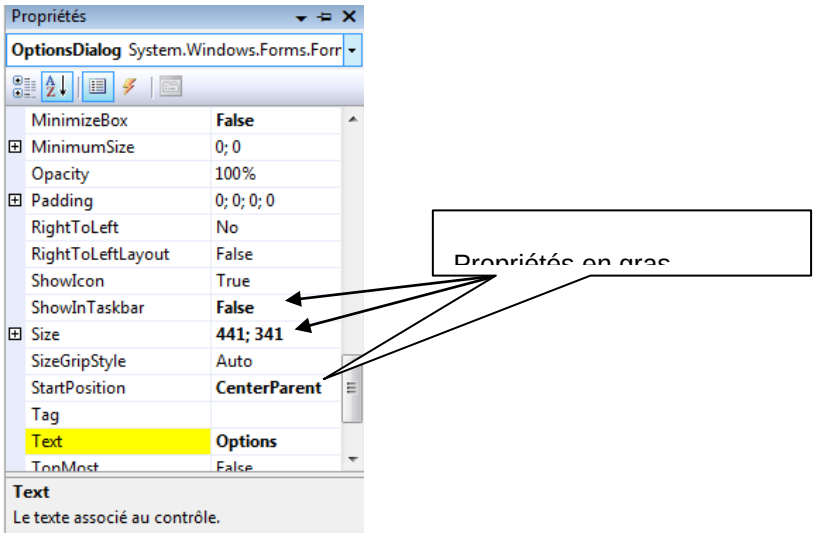
Comment configurer un formulaire pour qu'il se comporte comme une boîte de dialogue ?

Et bien profitons du fait que Visual Studio nous a généré une boîte de dialogue toute prête pour observer les paramètres qu'il aurait fallu configurer sur un formulaire Windows Form standard ☺ !

3. Affichez les propriétés du formulaire généré :

   - Sélectionnez le formulaire en mode **Design** puis faites un clic droit > **Propriétés** (ou cliquez la touche **F4**).
   - Observez toutes les propriétés dont la valeur est en caractères gras.

Vérifiez que c'est bien l'objet Form **OptionsDialog** qui est sélectionné dans la fenêtre de Propriétés



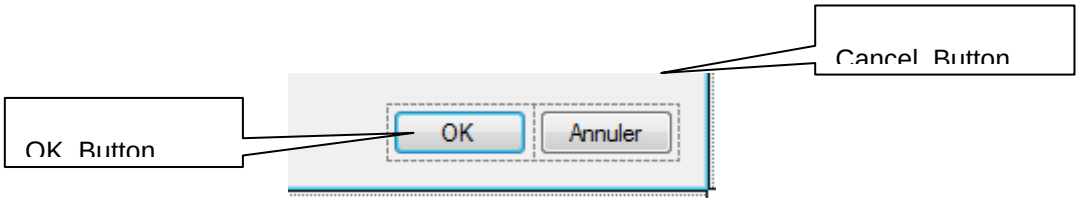
Quelles sont ces propriétés en gras ?

Ce sont celles dont la valeur est différente de la valeur par défaut, c'est-à-dire celles que Visual Studio a préconfigurées pour nous de façon à ce que le formulaire se comporte comme une boîte de dialogue.

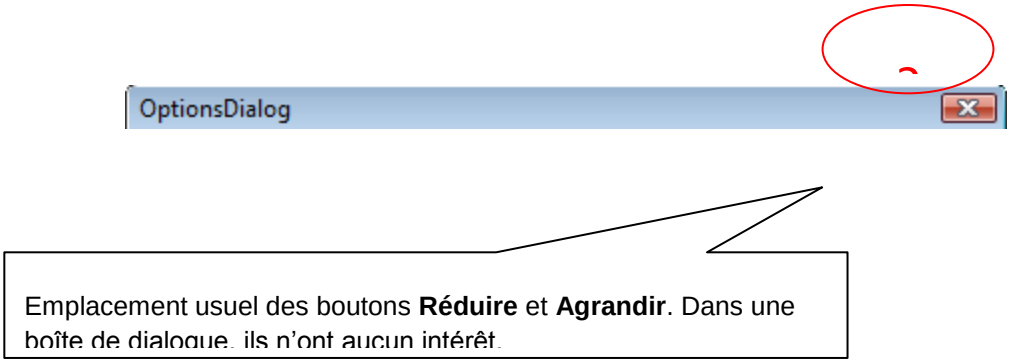
Hormis les propriétés **(Name)** et **Text** qui sont respectivement le nom du formulaire et son titre, voici les propriétés qu'il faut retenir :

Nom	Valeur	Explication
AcceptButton	**OK_Button**	Indique quel est le bouton sur lequel pourra cliquer l'utilisateur pour sauvegarder les changements du formulaire. On parle de *bouton d'acceptation* du formulaire. L'utilisateur pourra aussi appuyer la touche ***Entrée*** du clavier pour déclencher le clic du bouton. Bien sûr, c'est à vous de coder la fermeture du formulaire et d'enregistrer les informations saisies par l'utilisateur.
CancelButton	**Cancel_Button**	Indique le *bouton d'annulation* du formulaire c'est-à-dire au contraire celui qui ferme le formulaire sans

		sauvegarder les changements. Il est associé à la touche **ECHAP**.



FormBorderStyle	**FixedDialog**	Détermine l'apparence du bord extérieur du formulaire. Le style **FixedDialog** empêche l'utilisateur de redimensionner la fenêtre.
MaximizeBox	**False**	Indique que le formulaire ne doit pas présenter de bouton **Agrandir** dans la barre de légende du formulaire.
MinimizeBox	**False**	Indique que le formulaire ne doit pas présenter de bouton **Réduire** dans la barre de légende du formulaire.
Size	**441;341**	Taille fixe proposée à 441 pixels de largeur et 341 pixels de hauteur.



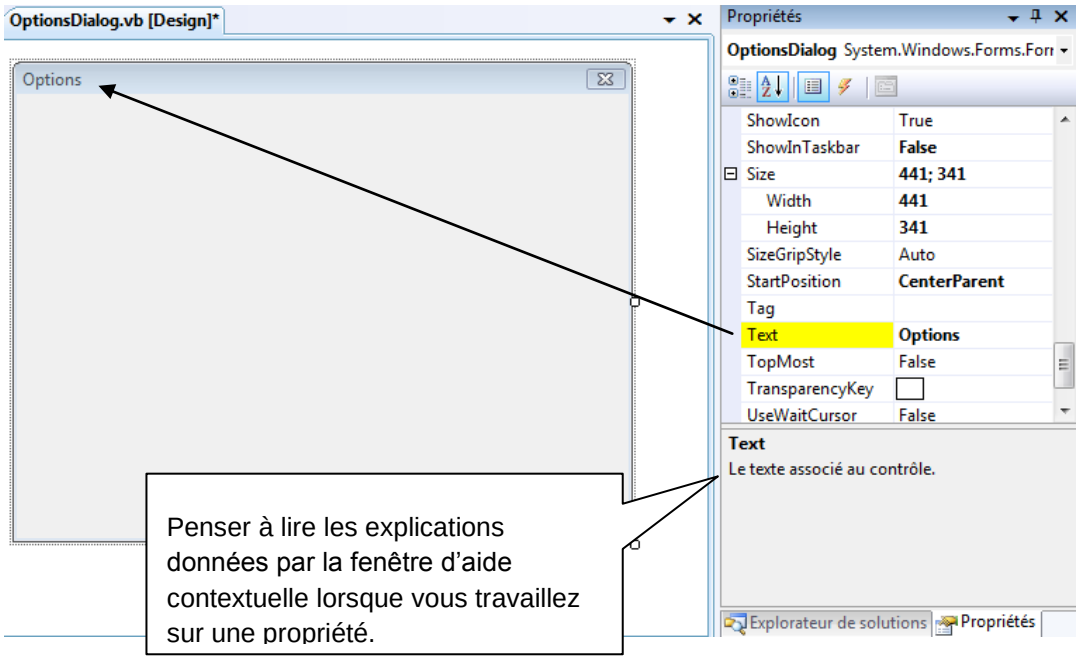
AutoScaleMode	**Font**	Désigne le mode de mise à l'échelle automatique du

		formulaire, en l'occurrence ici basé sur la police des caractères.
StartPosition	**CenterParent**	Détermine la position initiale de la boîte de dialogue au moment de l'affichage. **CenterParent** indique que le formulaire doit apparaitre au centre du formulaire parent qui a initié son affichage.
ShowInTaskbar	**False**	Indique qu'il n'est pas utile d'afficher le formulaire dans la barre de tâches Windows puisqu'il ne s'agit pas d'une application à part entière.

Pour en savoir plus sur la configuration de ces propriétés, consulter la rubrique **Exemples** de la page :

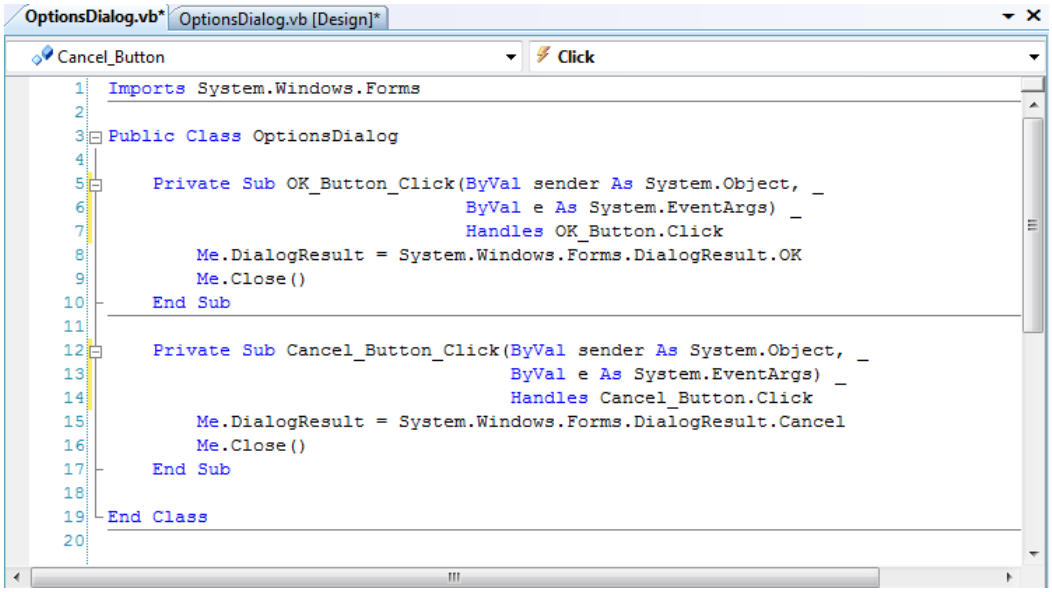
http://msdn.microsoft.com/fr-fr/library/system.windows.forms.form.aspx

- Modifiez maintenant le titre du formulaire pour que la boîte de dialogue s'appelle : **Options**.



4. Basculez dans le fichier de code du formulaire pour voir ce qui a été généré par le modèle d'élément Boîte de dialogue :

    - Sélectionnez le formulaire puis faites un clic droit > **Afficher le code**.



Le fichier de code contient une classe nommée **OptionsDialog** qui comprend deux gestionnaires d'évènements :

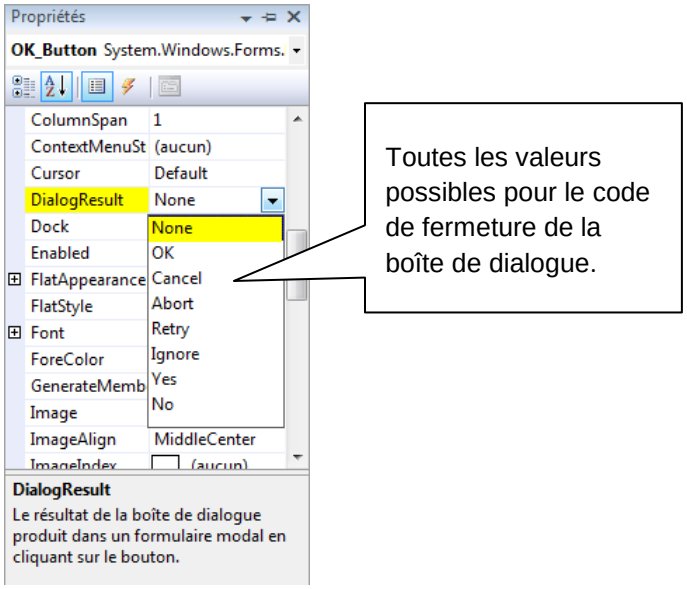
- **OK_Button_Click** qui gère (**Handles**) l'évènement **Click** sur le bouton **OK_Button**.
- **Cancel_Button_Click** qui gère (**Handles**) l'évènement **Click** sur le bouton **Cancel_Button**.

Ces deux gestionnaires gèrent la fermeture du formulaire. Pour cela ils invoquent tout simplement la méthode **Close** du formulaire comme nous l'avons vu dans l'atelier précédent.

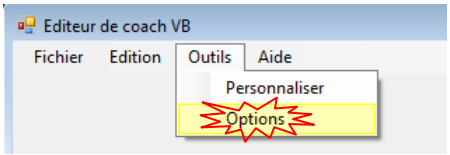
Mais comment l'application saura-t-elle que l'utilisateur a cliqué sur le bouton **OK** ou sur le bouton **Annuler** ?

Dans le premier cas, il faudra d'ailleurs gérer l'enregistrement des options configurées par l'utilisateur alors que dans l'autre, non. Du coup, juste avant de fermer la boîte de dialogue, les deux gestionnaires d'évènement configure une sorte de code de fermeture pour le formulaire parent par l'intermédiaire de la propriété **DialogResult** du formulaire. Nous reviendrons sur cette propriété juste après dans cet exercice en même temps que nous parlerons de la méthode d'affichage de la boîte de dialogue, à laquelle elle est directement liée.

Sachez que le code de fermeture peut être configuré de manière déclarative directement via la propriété **DialogResult** des boutons.



Maintenant que vous avez décortiqué la boîte de dialogue en long et en large, il s'agit de la connecter au reste de l'application pour qu'elle s'affiche lorsque l'utilisateur clique le menu **Options** dans le menu principal du formulaire **Main**.

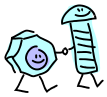


Où allez-vous connecter le code ?

Nous avons vu dans l'atelier précédent, que le principe consiste à écrire des gestionnaires d'évènements. Donc vous devez trouver quel est l'évènement déclencheur de l'affichage de la boîte de dialogue et à quel objet il correspond.

Pour cela, aidez-vous de l'expression textuelle de votre besoin :

« l'idée est de réagir **quand l'utilisateur clique** sur **Outils > Options dans le menu principal**. »



La conjonction *quand* est généralement suivie du verbe qui donne l'évènement déclencheur : **Click** . La suite de l'expression donne l'objet sur lequel l'évènement va porter : **OptionsToolStripMenuItem**.

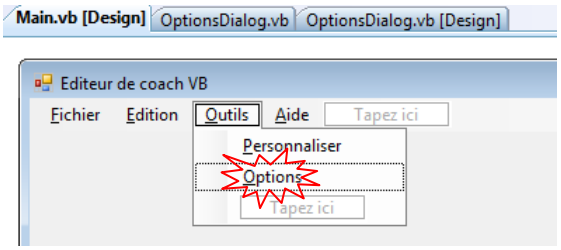
5. Codez l'affichage de la boîte de dialogue dans le formulaire principal de l'application :

   - Ouvrez le formulaire **Main.vb** en mode **Design** en double cliquant sur le fichier dans l'**Explorateur de solutions**.

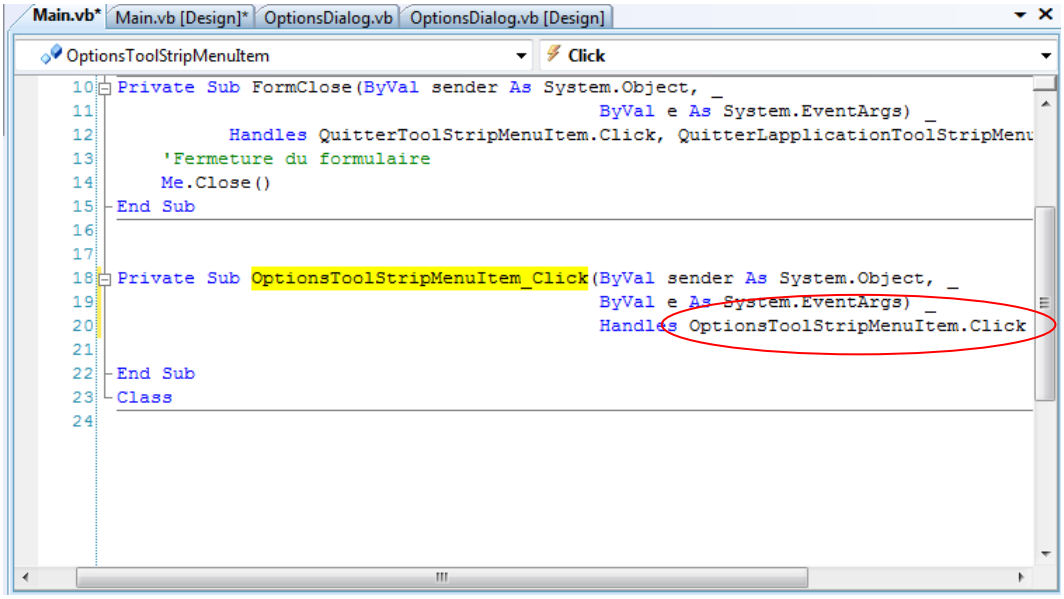
Comme il se trouve que **Click** est l'évènement par défaut d'une option de menu de type **MenuItem**, la méthode la plus rapide pour créer un gestionnaire d'évènement en réponse à cet évènement est donc de double cliquer sur l'option de menu dans le **Concepteur de formulaire** de Visual Studio.

Vous pouvez retrouver toutes les autres méthodes de création d'un gestionnaire d'évènement en vous reportant à l'atelier 2 précédent.

   - Double cliquez sur l'option de menu **Options** dans le menu **Outils** du formulaire **Main** pour générer une procédure de réponse à l'évènement **Click**.



Visual Studio bascule automatiquement sur le fichier de code du formulaire **Main** et créé une procédure **OptionsToolStripMenuItem_Click** en réponse à l'évènement **OptionsToolStripMenuItem.Click**.



- Créer un objet de type **OptionsDialog** puis afficher-le à l'aide de la méthode **ShowDialog** comme suit :

Code VB

```
Private Sub OptionsToolStripMenuItem_Click(ByVal sender As System.Object, _
```

```
                    ByVal e As System.EventArgs) _
                    Handles OptionsToolStripMenuItem.Click
    Dim formOptions As OptionsDialog
    formOptions = New OptionsDialog()
    formOptions.ShowDialog()
End Sub
```

Que font ces quelques lignes ?

La première déclare une variable nommée **formOptions** de type **OptionsDialog** à l'aide des mots clés **Dim** (abréviation de *Dimension*) et **As**.

Dim <NomDeLaVariable> As <TypeDeDonnées>

Qu'est ce qu'on entend par *variable* ?

On définit des variables pour stocker en mémoire des informations dont la valeur varie (d'où le nom de *variable*) dans un programme. Une variable est caractérisée par :

- son nom
- sa valeur
- son adresse mémoire (où est stockée sa valeur)
- son type de données
- sa portée
- sa durée de vie

C'est quoi un *Type de données* ?

Le type d'une variable fait référence au genre de données qu'elle peut contenir (date, entier etc.) et à la manière dont les données sont stockées. Il détermine notamment la taille que la variable va utiliser en mémoire (8 octets pour une date, 4 octets pour un entier etc.), les valeurs autorisées pour cette variable et donc les opérations possibles sur celle-ci. C'est pour cela qu'il est si important de toujours typer vos variables.

Comment ça marche dans les coulisses ?

Il faut savoir que le runtime du Framework.NET ne connait qu'un seul jeu de types de données géré par son Système de types communs. Il est dit « commun » parce qu'il est commun à tous les langages. En effet, lorsque vous déclarez une variable d'un certain type VB, le compilateur VB s'occupe de faire correspondre le type du langage avec le type correspondant défini dans le Système de types communs. Cette correspondance est faite quelque soit le langage avec lequel vous développez.

Ce système est fondamental parce qu'il est sécurisé et facilite l'intégration inter-langage. Par exemple c'est grâce à lui que vous pouvez vous permettre de réutiliser un composant écrit en VB dans une application C#.

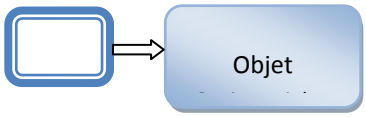
Pour en savoir plus sur le système de type commun du Framework .NET :

http://msdn.microsoft.com/fr-fr/library/zcx1eb1e.aspx

Type *valeur* ou type *référence* ?

Il faut distinguer deux grandes catégories de types de données : les types valeurs et les types références.

- Les variables de type valeur stockent directement leurs données (comme un entier ou une date). C'est possible parce que le système sait exactement la taille à allouer pour les données (un Int32 fait par exemple toujours 32 octets !).



- Les variables de type référence ne peuvent pas stocker directement leurs données parce que le système ne sait pas nécessairement la taille à allouer pour celles-ci. Une classe ou une chaîne de caractères par exemple sont des types références. Et oui, comment savoir à l'avance la taille d'une classe *OptionsDialog* ? C'est une variable bien trop complexe…

Si une variable de type référence ne stocke pas directement la donnée, alors que stocke-t-elle ? Elle stocke une *référence* vers la donnée, qui représente la localisation de la donnée en mémoire.

Si on en revient à la ligne , l'objectif étant d'afficher un formulaire, la première étape consiste à déclarer une variable en la nommant (formOptions) et en définissant son type (OptionsDialog qui est le nom de la classe du formulaire que l'on veut utiliser).

Dim formOptions As OptionsDialog

Dim sert à déclarer et allouer un espace de stockage pour la variable.

La clause **As** sert à spécifier le type de données de la variable.

Pour en savoir plus sur les mots clés **Dim** et **As** :

http://msdn.microsoft.com/fr-fr/library/7ee5a7s1.aspx

http://msdn.microsoft.com/fr-fr/library/97f49yhb.aspx

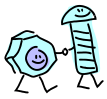
Attention ! *OptionsDialog* est une classe, donc il s'agit d'un type référence.

Que fait la ligne ? Elle déclare et alloue un espace pour stocker non pas la valeur de la variable mais une référence à celle-ci. Il nous faut maintenant créer l'objet proprement dit et lui allouer de la mémoire. C'est l'objet de la ligne qui utilise le mot clé **New** pour créer l'objet. On dit que l'on créé une *instance* de la classe OptionsDialog.

formOptions = New OptionsDialog()

Contient une référence vers l'objet

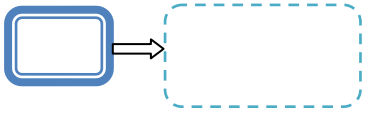
New est suivi du constructeur de la classe OptionsDialog.



Qu'est ce qu'un constructeur ?

Nous reviendrons sur cette notion très importante de la programmation orientée objet plus tard dans ce tutorial. Retenez pour l'instant que le constructeur est une méthode membre de la classe (d'où les parenthèses qui caractérisent l'appel d'une méthode) que l'on appelle pour (*construire)* créer une instance d'un nouvel objet. C'est typiquement dans un constructeur que l'on initialiserait l'objet.

A votre avis est-ce que le programme peut exploiter la variable **formOptions** avant que l'objet soit créé ? En clair, est-ce qu'on peut exploiter une référence qui n'est pas encore associée à un objet ?



La réponse est clairement non ! Que feriez-vous d'une porte d'accès qui s'ouvre sur le vide ? Justement pour matérialiser ce *vide*, VB fournit un mot clé : **Nothing**. Lorsqu'une variable de type référence est égale à **Nothing** c'est qu'elle n'est associée à (rien) aucun objet. Inversement lorsque vous voulez que la variable ne référence plus aucun objet, il suffit de lui assigner pour valeur **Nothing**.

Pour en savoir plus sur le mot clé **New** :

http://msdn.microsoft.com/fr-fr/library/77s47661.aspx

Pour en savoir plus sur le mot clé **Nothing** :

http://msdn.microsoft.com/fr-fr/library/0x9tb07z.aspx

Et voilà ! Nous disposons maintenant d'un objet de type formulaire **OptionsDialog**. Il ne nous reste plus qu'à l'afficher. C'est là qu'intervient la méthode **ShowDialog** . Elle a pour effet d'afficher le formulaire en tant que boîte de dialogue modale. L'exécution de la suite du code (juste après l'appel de la méthode) est suspendue jusqu'à ce que l'utilisateur ferme la boîte de dialogue. Au retour de l'appel, nous allons récupérer le code de fermeture configuré dans le formulaire au moyen de la propriété **DialogResult**.

Pour tout savoir sur la méthode **ShowDialog** :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.form.showdialog.aspx

La propriété **DialogResult** de la classe **Form** est de type **System.Windows.Forms.DialogResult**. Ce type de données est ce qu'on appelle une *énumération*.

Qu'est qu'une énumération ?

Une énumération est un ensemble de constantes symboliques nommées. On peut associer chaque constante à une valeur entière. Par exemple la définition du type **DialogResult** pourrait ressembler à ceci :

Enum DialogResult

None = 0

OK = 1

Cancel = 1

Abort = 2

Retry = 3

Ignore = 4

Yes = 5

No = 6

End Enum

Nous créerons une énumération personnalisée dans la suite de cet atelier. L'intérêt de ce type de données et qu'au lieu de manipuler des codes sous la forme de nombres entiers (0, 1, 2, etc…), on utilise des noms très explicites (OK, Cancel, Retry etc…) ce qui donne un code très lisible et facile à maintenir.

DialogResult.OK ou **DialogResult.No** sont des exemples de codes de fermeture d'une boîte de dialogue (tellement plus sympas que 1 ou 6…).

Pour en savoir plus sur la **propriété DialogResult** :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.form.dialogresult.aspx

Pour en savoir plus sur le **type** énuméré **DialogResult** :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.dialogresult.aspx

- Récupérez le code de fermeture de la boîte de dialogue en retour de l'appel à la méthode **ShowDialog** comme suit :

Code VB

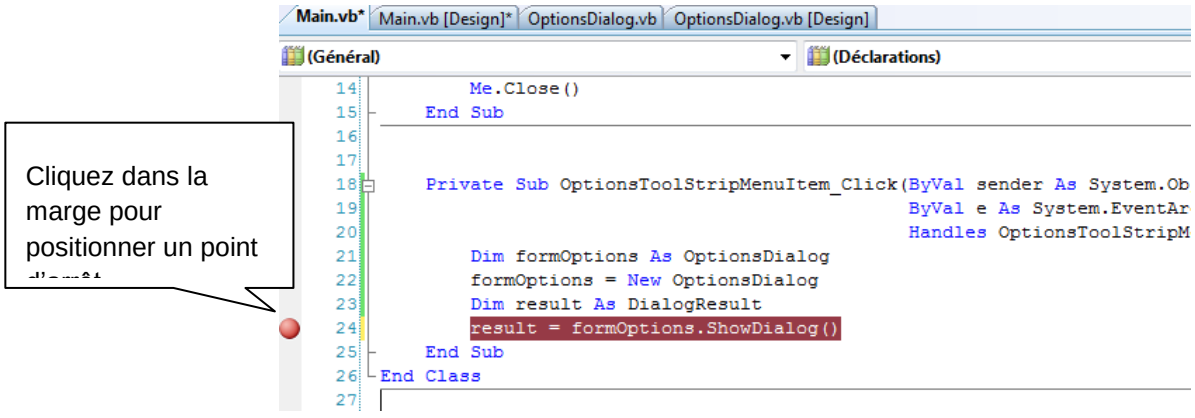
```
Private Sub OptionsToolStripMenuItem_Click(ByVal sender As System.Object, _
```

```
                                ByVal e As System.EventArgs) _
                                Handles OptionsToolStripMenuItem.Click
        Dim formOptions As OptionsDialog
        formOptions = New OptionsDialog()
        Dim result As DialogResult
        result =formOptions.ShowDialog()
End Sub
```

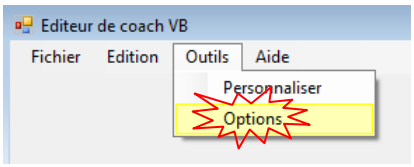
Pour l'instant, le principe consiste à déclarer une variable de type **System.Windows.Forms.DialogResult** et de récupérer la valeur de retour de l'appel à la méthode **ShowDialog** dans cette variable. Nous analyserons le code de fermeture correspondant dans la suite de cet atelier.

6. Testez le fonctionnement de la boîte de dialogue :

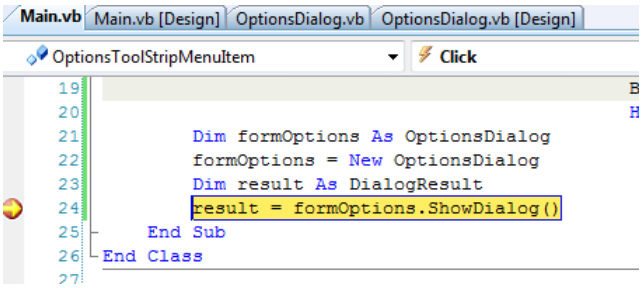
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Positionnez un point d'arrêt sur la ligne d'appel de la méthode **ShowDialog** en cliquant dans la marge (ou touche F9).



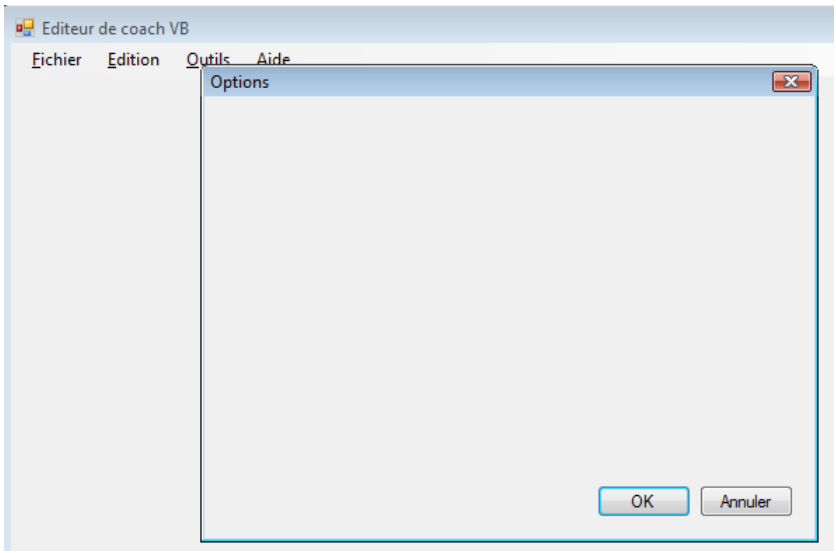
- Exécutez l'application **en mode Débogage** en cliquant sur (ou touche F5).
- Sélectionnez le menu **Outils** > **Options** de l'application.



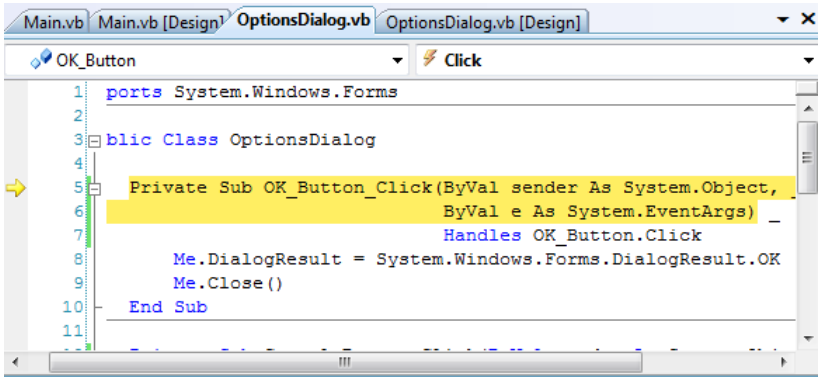
- Visual Studio s'arrête sur la ligne d'appel à la méthode **ShowDialog**.



- Cliquez **F8 (ou)** pour poursuivre l'exécution pas à pas de l'application. Vérifiez que la boîte de dialogue s'affiche.



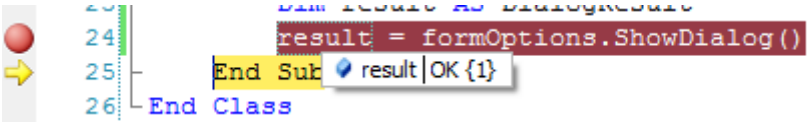
- Validez la boîte de dialogue en cliquant le bouton **OK**. Le débogueur bascule dans le fichier **OptionsDialog.vb** et vous amène sur le gestionnaire d'évènement **Click** du bouton **OK** de la boîte de dialogue.



- Continuez l'exécution pas à pas (F8) pour voir l'affectation du code de fermeture **DialogResult.OK** à la propriété **DialogResult** du formulaire.
- Après plusieurs clics sur la touche F8, vous êtes de retour dans le fichier **Main.vb** sur la ligne d'appel de la méthode **ShowDialog**.

Vous constatez au passage le fonctionnement *modal* de la boîte de dialogue. En effet, vous vous retrouvez exactement sur la même ligne d'exécution qu'avant l'affichage de la boîte de dialogue. Tant que cette dernière n'est pas fermée, l'exécution de la procédure en cours est restée suspendue.

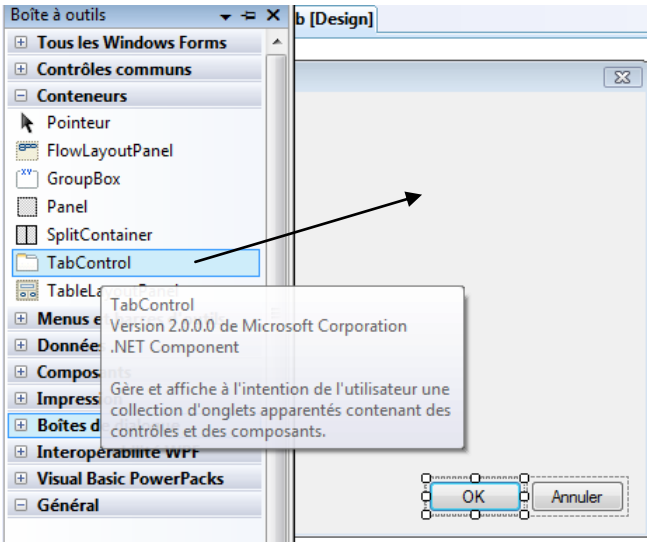
- Cliquez encore une fois **F8** pour basculer sur la ligne **End Sub**. Positionnez le curseur de la souris sur la variable **result** pour en observer la valeur. Vous devez voir le code de fermeture **OK** correspondant à **DialogResult.OK** (soit une constante de valeur 1).



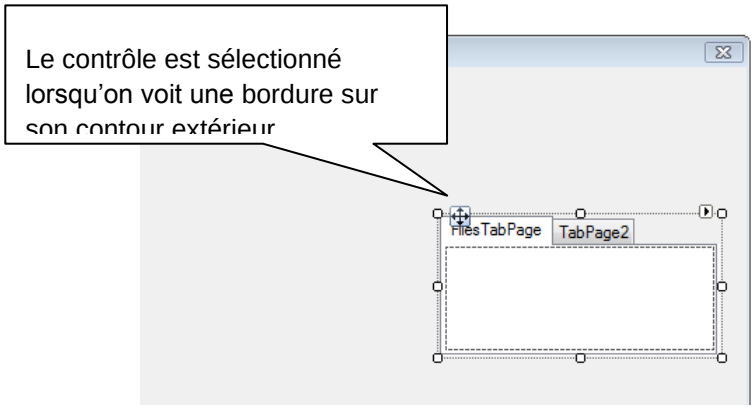
- Cliquez **F5** puis terminer l'application.
- Supprimez le point d'arrêt sur la ligne d'appel à la méthode **ShowDialog**.

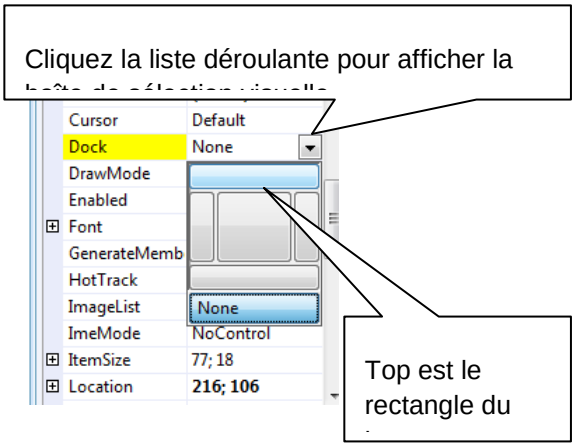
Bravo ! Vous venez de créer votre première boîte de dialogue ! Vous allez maintenant dessiner les contrôles de saisie des différentes options proposées dans la boîte.

7. Commencez par ajouter un contrôle de type TabControl au formulaire OptionsDialog pour construire une fenêtre à base d'onglets :
   - Affichez le fichier **OptionsDialog.vb** en mode **Design**.
   - Faites un glisser déplacer de la **Boîte à outils** > rubrique **Conteneurs** > **TabControl** n'importe où sur la surface du formulaire.



- Affichez la fenêtre de propriétés du contrôle en le sélectionnant puis en pressant la touche **F4**.
- Configurez sa propriété **Dock** à **Top** en vous aidant du mini éditeur graphique proposé.





Que fait la propriété **Dock** ?

Il s'agit d'une propriété de mise en forme de base des contrôles standards des Windows Forms. Elle décrit à quel bord du conteneur parent, un contrôle doit être fixé. Le contrôle

est alors déplacé aussi près que possible du bord spécifié et dimensionné de manière à remplir ce bord. Il reste en place même si le conteneur parent est redimensionné.

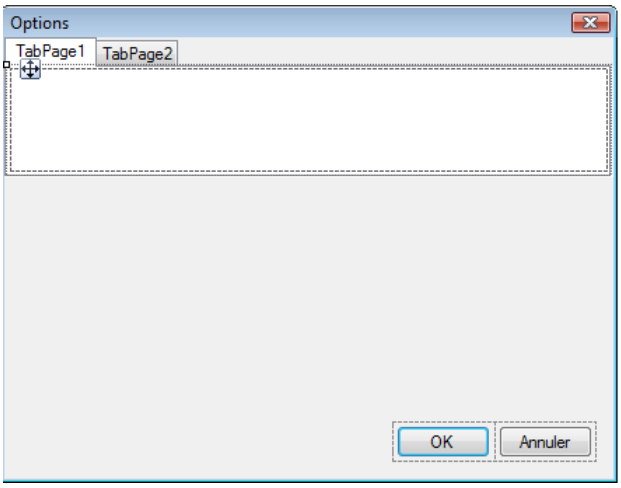
Pour en savoir plus sur la propriété **Dock** :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.control.dock(VS.80).aspx

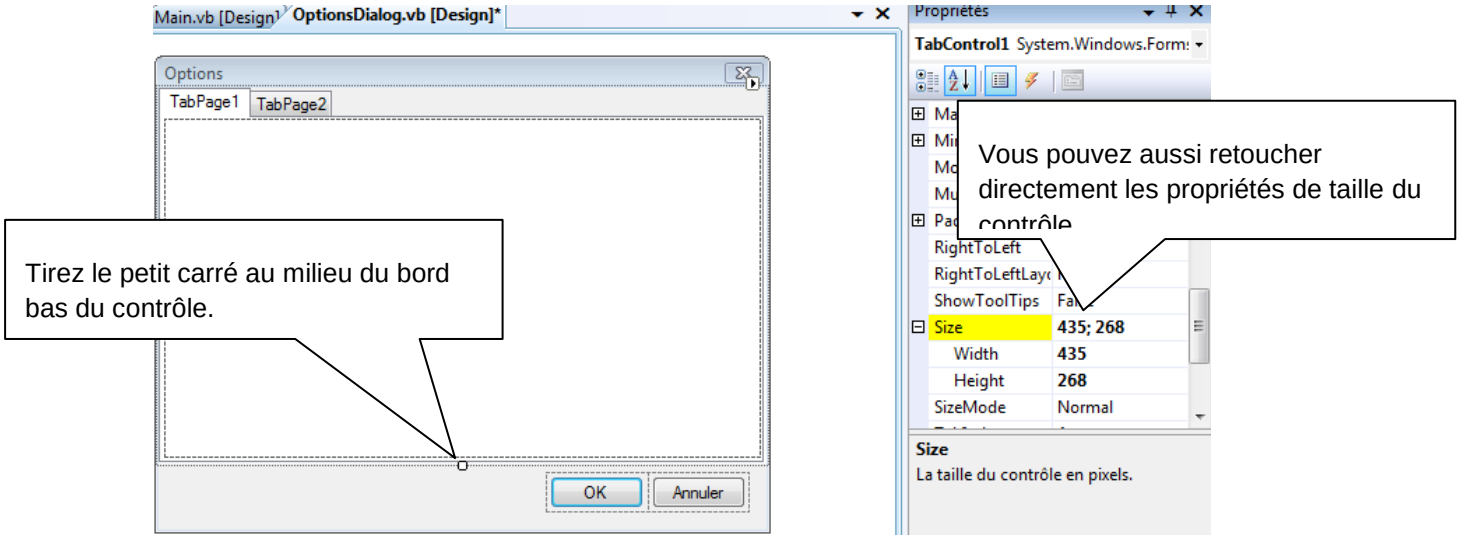
Le docking est en fait un cas spécial de l'ancrage défini par une autre propriété des contrôles nommée **Anchor**, que nous aurons l'occasion d'utiliser dans la suite de ce tutorial. Pour creuser le sujet dès maintenant :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.control.anchor(VS.80).aspx

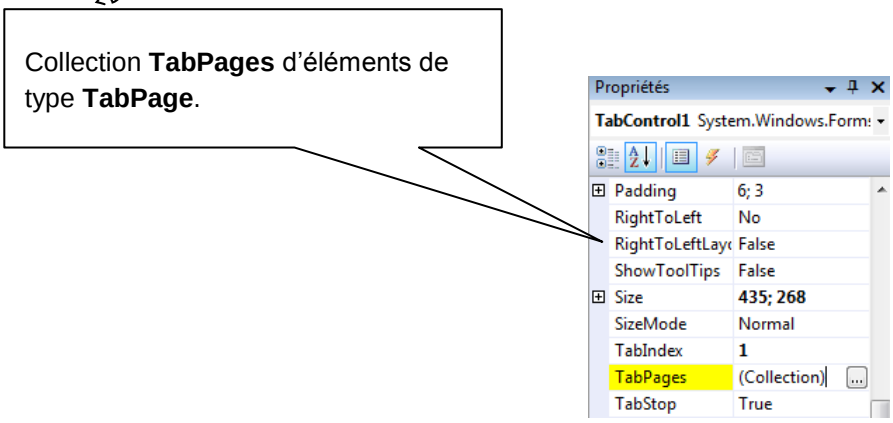
Vous devez obtenir :



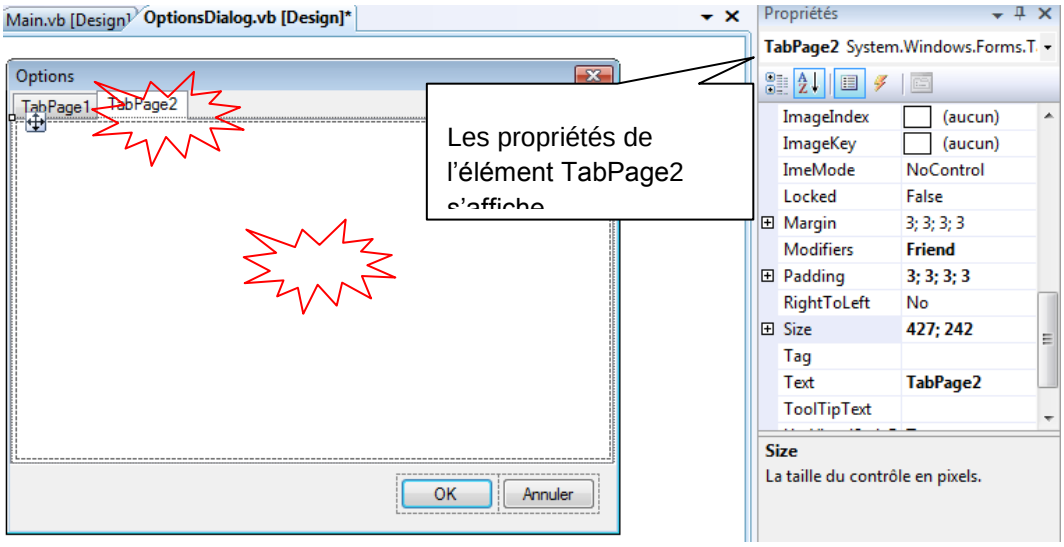
- Avec la souris, faites glisser le bord bas du contrôle **TabControl** pour l'étendre et l'amener juste au dessus des boutons.



Attention ! Un contrôle **TabControl** présente une collection de contrôles enfants **TabPages**, symbolisant les pages associées à chaque onglet.

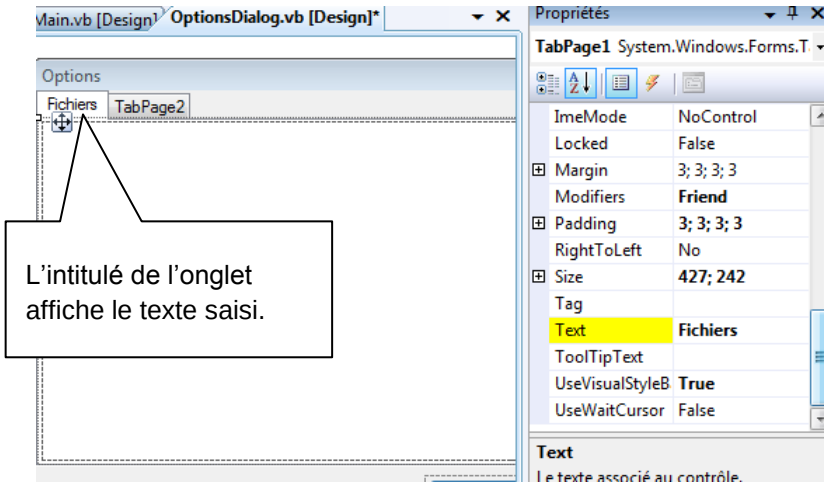


Pour sélectionnez une page, sélectionnez d'abord l'onglet puis cliquez n'importe où dans la page. Par exemple, pour dessiner la page du deuxième onglet, cliquez l'onglet **TabPage2** puis n'importe où dans la page de contenu de l'onglet.



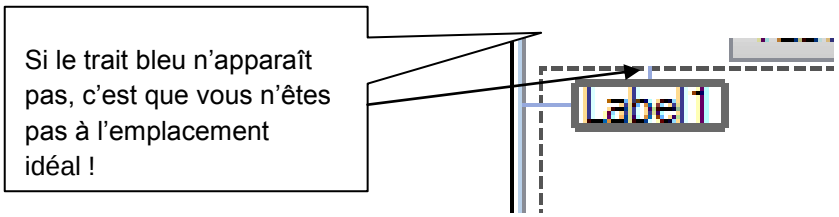
8. Intitulez le premier onglet **Fichiers** et ajoutez-lui les contrôles de saisie pour gérer les options de configuration de fichier :

   - Sélectionnez le premier onglet **TabPage1** du contrôle **TabControl1**.
   - Modifiez sa propriété **Text** en : **Fichiers**.



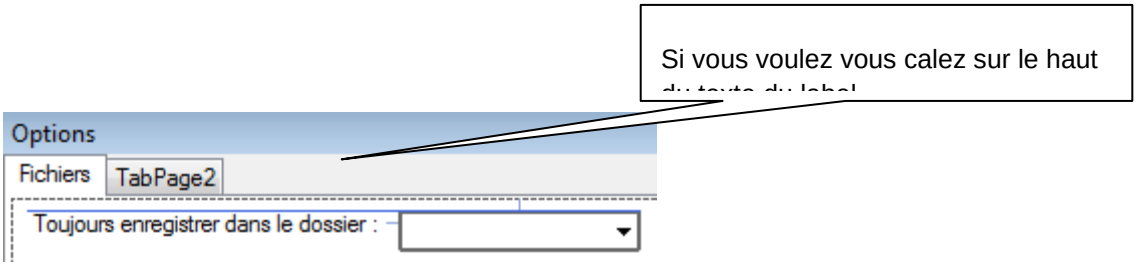
   - Faites un glisser déplacer de la **Boîte à outils** > rubrique **Contrôles commun** > d'un contrôle **Label** puis d'un contrôle **ComboBox** sur la surface de la page **Fichiers**.

Utilisez le concepteur de formulaire de Visual Studio pour positionner rapidement vos contrôles et les aligner. En effet, lorsque vous approchez un contrôle du bord gauche d'un autre contrôle, Visual Studio vous donne des indications sur la position idéale à adopter. Dès que vous êtes bien positionné, il vous indique l'espacement via des traits bleus.

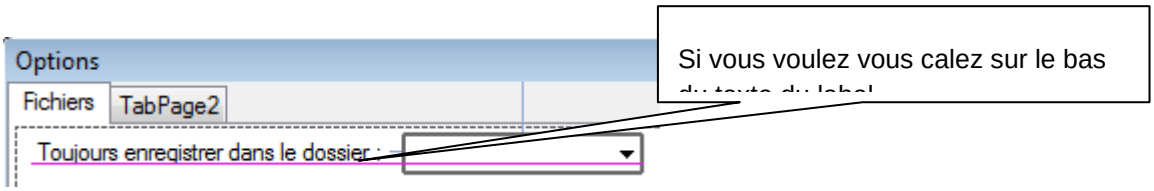


- Modifiez les propriétés du contrôle de type **Label** comme suit :
  - (Name) : **SaveDirectoryTypeLabel**
  - Text : **Toujours enregistrer dans ce dossier :**

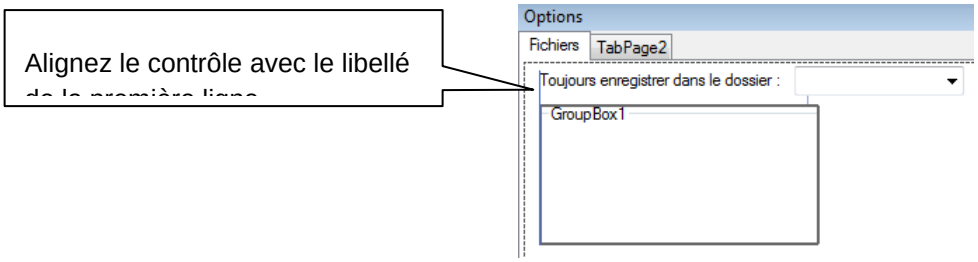
Aligner le contrôle de type **ComboBox** immédiatement après le contrôle de type **Label** en utilisant les indications du **Concepteur de formulaire**.



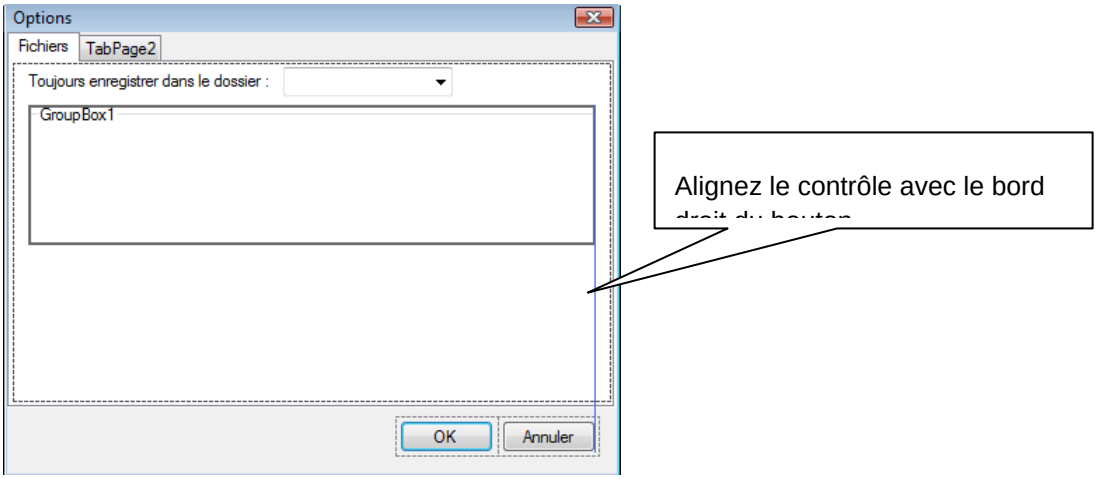
Ou :



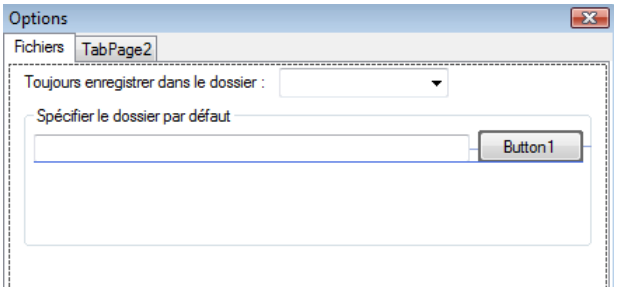
- Modifiez les propriétés du contrôle de type **ComboBox** :
    - (Name) : **SaveDirectoryTypeComboBox**
- Faites un glisser déplacer de la **Boîte à outils** > rubrique **Conteneurs** > d'un contrôle **GroupBox** sous les contrôles précédents :



- Redimensionnez le contrôle en largeur en vous calant sur la position du bouton le plus à droite au bas du formulaire :

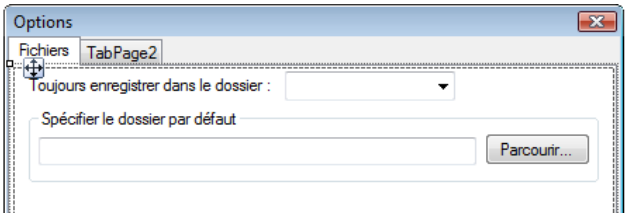


- Modifiez les propriétés du contrôle de type **GroupBox** comme suit :
    - (Name) : **SaveDirectoryPathGroupBox**
    - Text : **Spécifier le dossier par défaut**
- Glisser déplacer à l'intérieur du contrôle conteneur **GroupBox** un contrôle commun de type **TextBox** et un contrôle de type **Button**.
- Positionnez-les pour qu'ils soient alignés.



- Modifiez les propriétés du contrôle de type **TextBox** comme suit :
    - (Name) : **SaveDirectoryPathTextBox**
- Modifiez les propriétés du contrôle de type **Button** comme suit :
    - (Name) : **SaveDirectoryPathButton**
    - Text : **Parcourir…**

Vous devez obtenir une page du type :



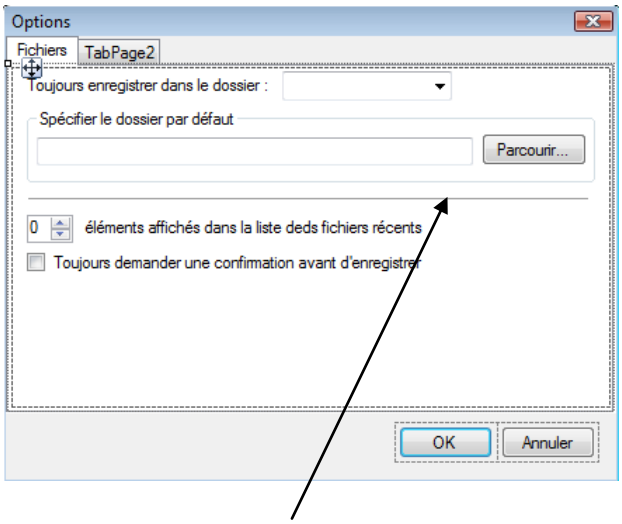
9. Terminez le dessin de la première page avec les contrôles suivants :

Type de contrôle	**NumericUpDown**
(Name)	**RecentFileListNumericUpDown**
Maximum	**5**
Width	**33**

Type de contrôle	**CheckBox**
(Name)	**ConfirmBeforeSaveCheckBox**
Text	**Toujours demander une confirmation avant d'enregistrer**

Type de contrôle	**Label**
(Name)	**RecentFileListLabel**
Text	**éléments affichés dans la liste des fichiers récents**

- Positionnez-les pour obtenir :



Tiens mais c'est quoi cette ligne horizontale qui sépare les contrôles ?

Malheureusement il n'existe pas de contrôle Windows Forms standard pour dessiner ce type de ligne. Pour le coup, il faut donc coder.

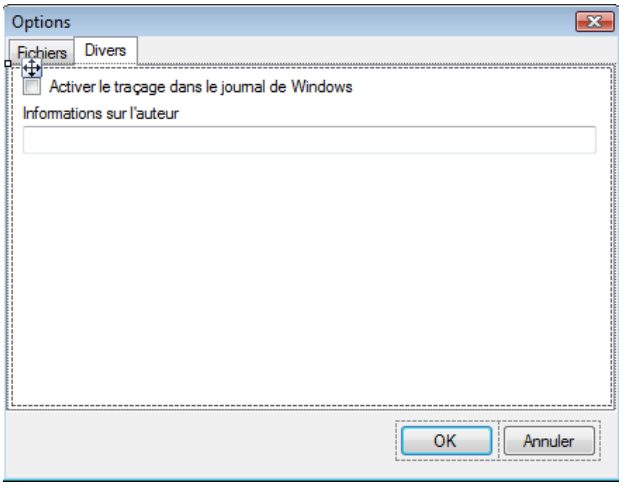
Sauf qu'il existe une petite feinte très simple…

Ajoutez au formulaire un contrôle standard de type **Label** et configurez le avec les propriétés suivantes :

- Text : (vide)
- AutoSize : **False** (super important sinon le contrôle se dimensionne tout seul et ne tient pas compte des paramètres qu'on lui donne)
- Size : **400;2** (à vous de voir pour la largeur mais configurez seulement 2 pixels de hauteur)
- BorderStyle : **Fixed3D**.

Et le tour est joué ☺ !

10. Maintenant que vous êtes chaud, vous n'avez plus qu'à dessiner le deuxième onglet comme suit :



- Voici les propriétés de chaque contrôle :

Type de contrôle	**TabPage2**
(Name)	**OtherTabPage**
Text	**Divers**

Type de contrôle	**CheckBox**
(Name)	**TraceCheckBox**
Text	**Activer le traçage dans le journal de Windows**

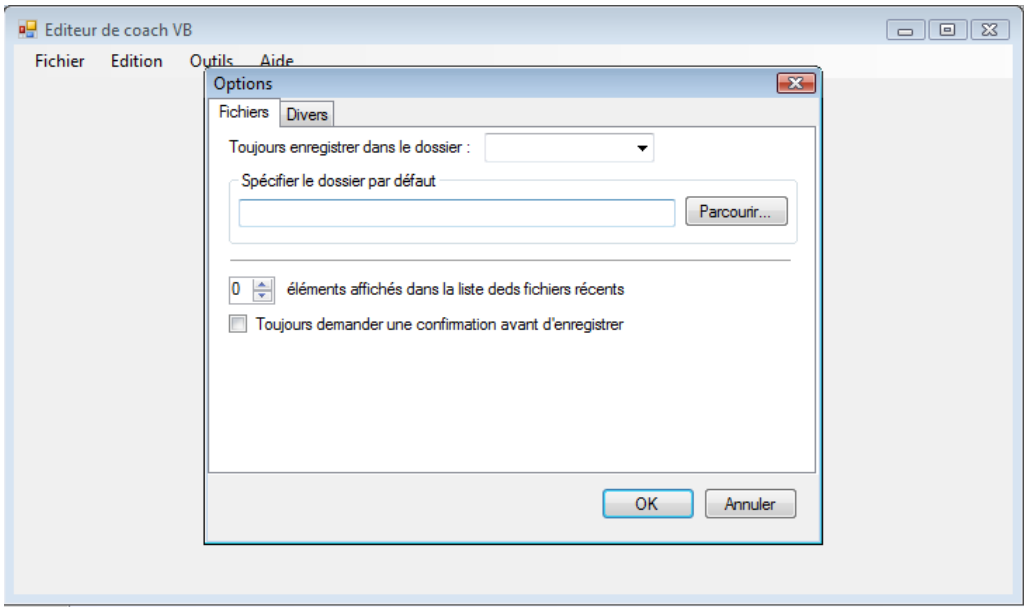
Type de contrôle	**Label**
(Name)	**AuthorInfoLabel**
Text	**Informations sur l'auteur**

Type de contrôle	**TextBox**
(Name)	**AuthorInfoTextBox**

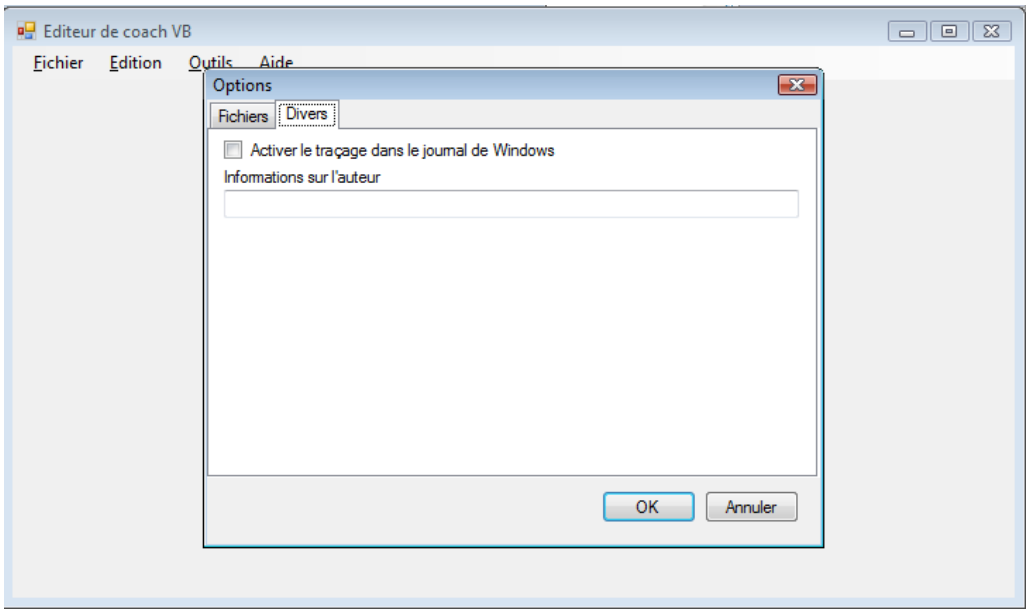
11. Testez le fonctionnement de la boîte de dialogue :

    - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
    - Exécutez l'application **en mode Débogage** en cliquant sur (ou touche F5).

- Sélectionnez le menu **Outils** > **Options** de l'application.



- Cliquez l'onglet **Divers** pour basculer sur la deuxième page d'options :



Super ! Maintenant il ne reste plus qu'à coder son fonctionnement !

- Fermez la boîte de dialogue puis fermez l'application.

UTILISER LA BOÎTE DE DIALOGUE STANDARD FOLDERBROWSERDIALOG

Le Framework .NET fournit en standard les boîtes de dialogue communes de Windows pour vous aider à construire une interface cohérente. Cela concerne des tâches telles que l'ouverture et l'enregistrement d'un fichier, la manipulation des polices ou des couleurs de texte, l'impression etc…

Voici un article intéressant sur le sujet :

http://msdn.microsoft.com/fr-fr/library/aa289504(VS.71).aspx

Nous vous proposons d'exploiter dans cet exercice celle qui permet de parcourir la structure de disques de la machine à la recherche d'un dossier particulier.

Contexte fonctionnel

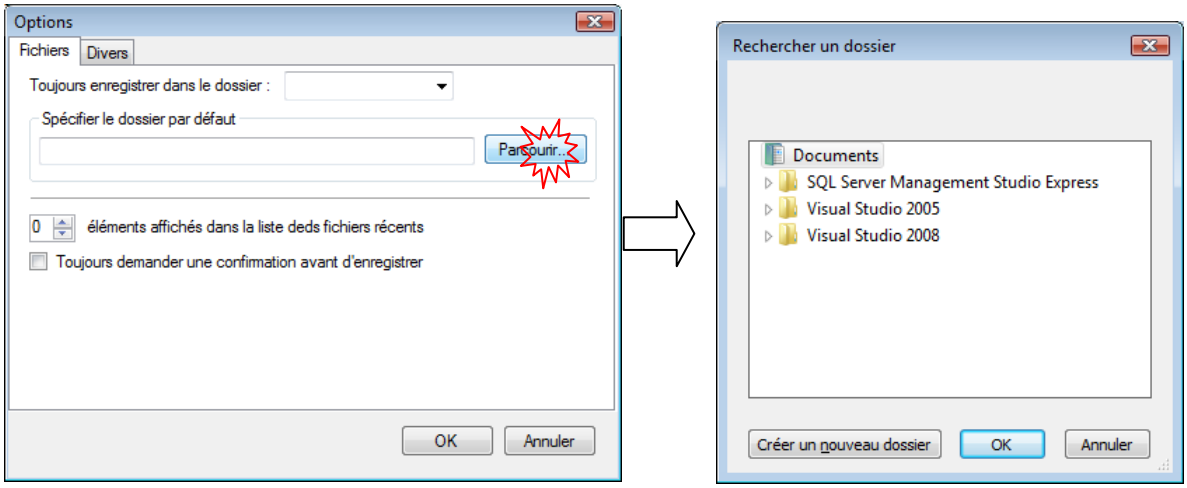
L'objectif est de configurer les options d'enregistrement des fichiers manipulés par l'éditeur. En effet, la boîte de dialogue *Options* permet de sélectionner le dossier qui est proposé par défaut à l'utilisateur au moment de l'enregistrement d'un fichier de données.

Deux choix s'offrent à lui :

- il peut choisir de toujours enregistrer ces fichiers par défaut dans son dossier **Mes Documents**.

- ou il peut décider d'un dossier particulier en entrant le chemin du dossier dans la zone de texte prévue à cet effet ou en sélectionnant le bouton **Parcourir…** de façon à rechercher le dossier sur la machine.

Dans cet exercice, nous allons nous concentrer sur le codage du bouton **Parcourir…** pour qu'il nous présente la boîte de dialogue standard de Windows de recherche d'un dossier.

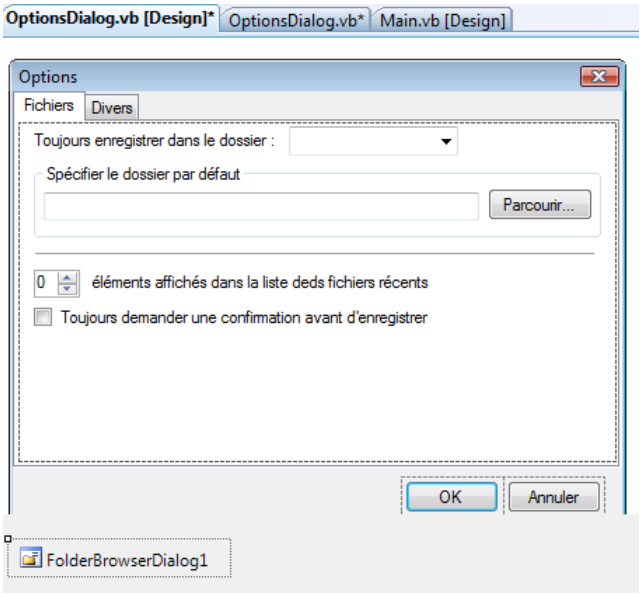


Déroulement de l'exercice :

1. Ajoutez une boîte de dialogue standard de type FolderBrowserDialog à la boîte de dialogue OptionsDialog :

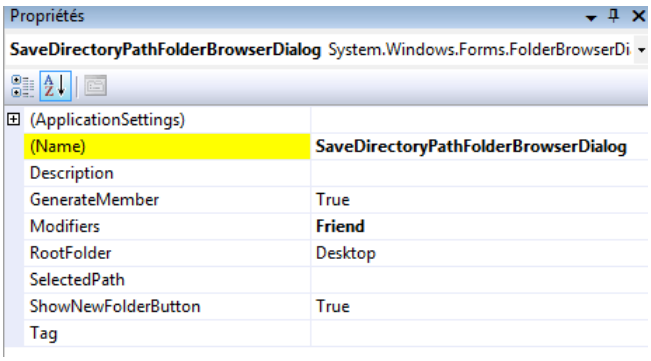
   - Affichez le formulaire **OptionsDialog** en mode **Design**.
   - Faites un glisser déplacer de la **Boîte à outils** > rubrique **Boîtes de dialogue** > du contrôle **FolderBrowserDialog** n'importe où sur la surface du formulaire.

Le **Concepteur de formulaire** positionne le composant dans la barre de composants juste au dessous du formulaire, puisqu'il ne lui est pas possible d'afficher sa représentation graphique directement sur la fenêtre.



L'utilisation de ce composant revient à créer une instance de la classe **FolderBrowserDialog** (c'est-à-dire un objet) du Framework .NET par simple glisser déplacer plutôt que de le faire par programmation dans le code. En plus, toutes les propriétés du composant sont configurables en mode **Design** via la fenêtre de **Propriétés** de Visual Studio.

- Configurez la propriété (Name) du contrôle à la valeur : **SaveDirectoryPathFolderBrowserDialog**.



Notez au passage les différentes propriétés du contrôle.

- Nous verrons comment configurer dynamiquement l'emplacement du dossier racine à partir duquel commence l'exploration, donné par la propriété **RootFolder**, dans la suite de cet atelier.
- La propriété **SelectedPath** nous donnera le chemin du dossier sélectionné par l'utilisateur dans la boîte de recherche.
- La propriété **ShowNewFolderButton** permet d'afficher un bouton **Créer un nouveau dossier**.

2. Codez l'affichage de la boîte de dialogue standard lorsque l'utilisateur clique sur le bouton :

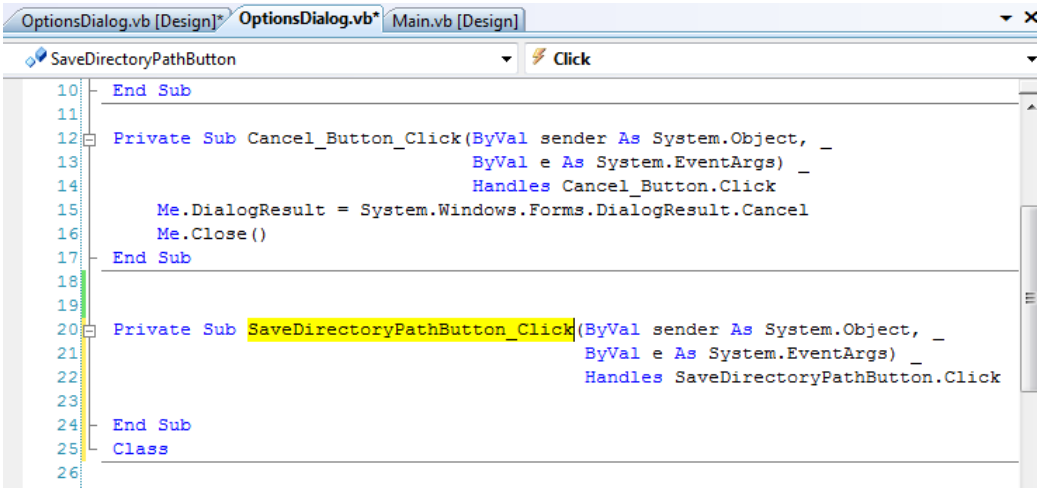
Ah, ça maintenant vous savez faire !

- Tout d'abord, il faut vous demander où vous allez brancher le code d'affichage de la boîte de dialogue. Rappelez-vous, c'est l'expression textuelle de votre besoin qui vous donne la réponse à cette question :

  « il faut afficher la boîte de dialogue **quand l'utilisateur clique** sur **le bouton Parcourir**… »

  Il s'agit donc de l'évènement **Click** du contrôle **SaveDirectoryPathButton**.
- Et pour afficher la boîte de dialogue, vous savez qu'il suffit d'invoquer la méthode **ShowDialog** sur l'instance de l'objet correspondant. L'objet en question, vous l'avez déjà, inutile de le créer et de l'instancier puisque c'est le composant **SaveDirectoryPathFolderBrowserDialog** que vous avez glissé sur votre formulaire.

- Basculez sur le formulaire **OptionsDialog** en mode **Design**.
- Double cliquez sur le bouton **Parcourir…** pour générer automatiquement un gestionnaire d'évènement en réponse au click sur le bouton **SaveDirectoryPathButton**.



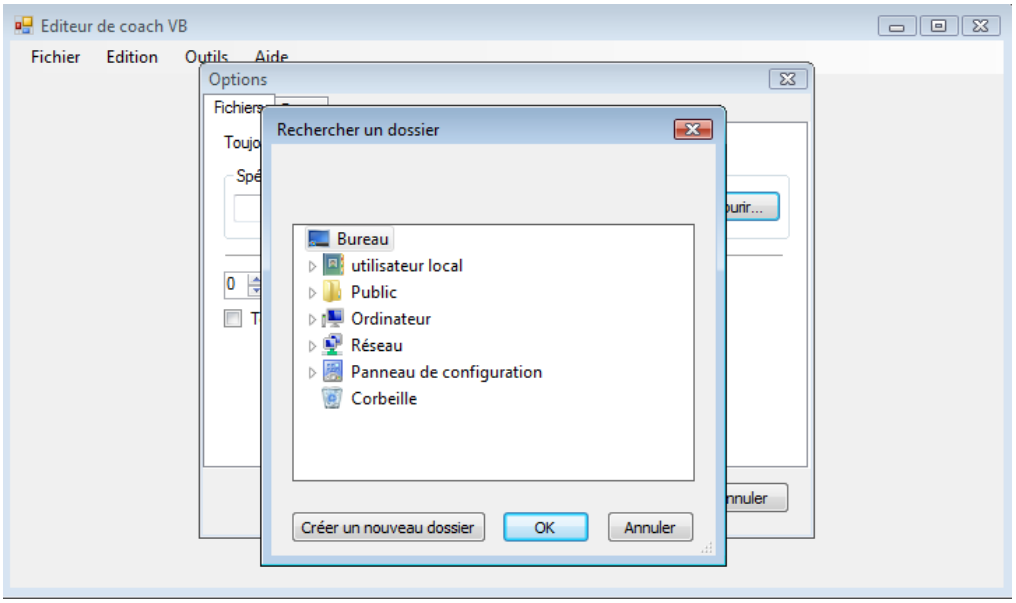
- Ajoutez le code d'affichage du formulaire comme suit :

Code VB

```
Private Sub SaveDirectoryPathButton_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles SaveDirectoryPathButton.Click
    SaveDirectoryPathFolderBrowserDialog.ShowDialog()
End Sub
```

3. Testez le fonctionnement de la boîte de dialogue :

   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application **en mode Débogage** en cliquant sur (ou touche F5).
   - Sélectionnez le menu **Outils** > **Options** de l'application.
   - Cliquez **Parcourir…** dans l'onglet **Fichiers**.



- Valider en cliquant deux fois sur OK.
- Fermer l'application.

Nous verrons dans la suite de cet atelier comment récupérer le chemin sélectionné par l'utilisateur dans la boîte de recherche de façon à l'afficher dans la zone de texte prévu à cet effet dans la boîte de dialogue *Options* de l'application.

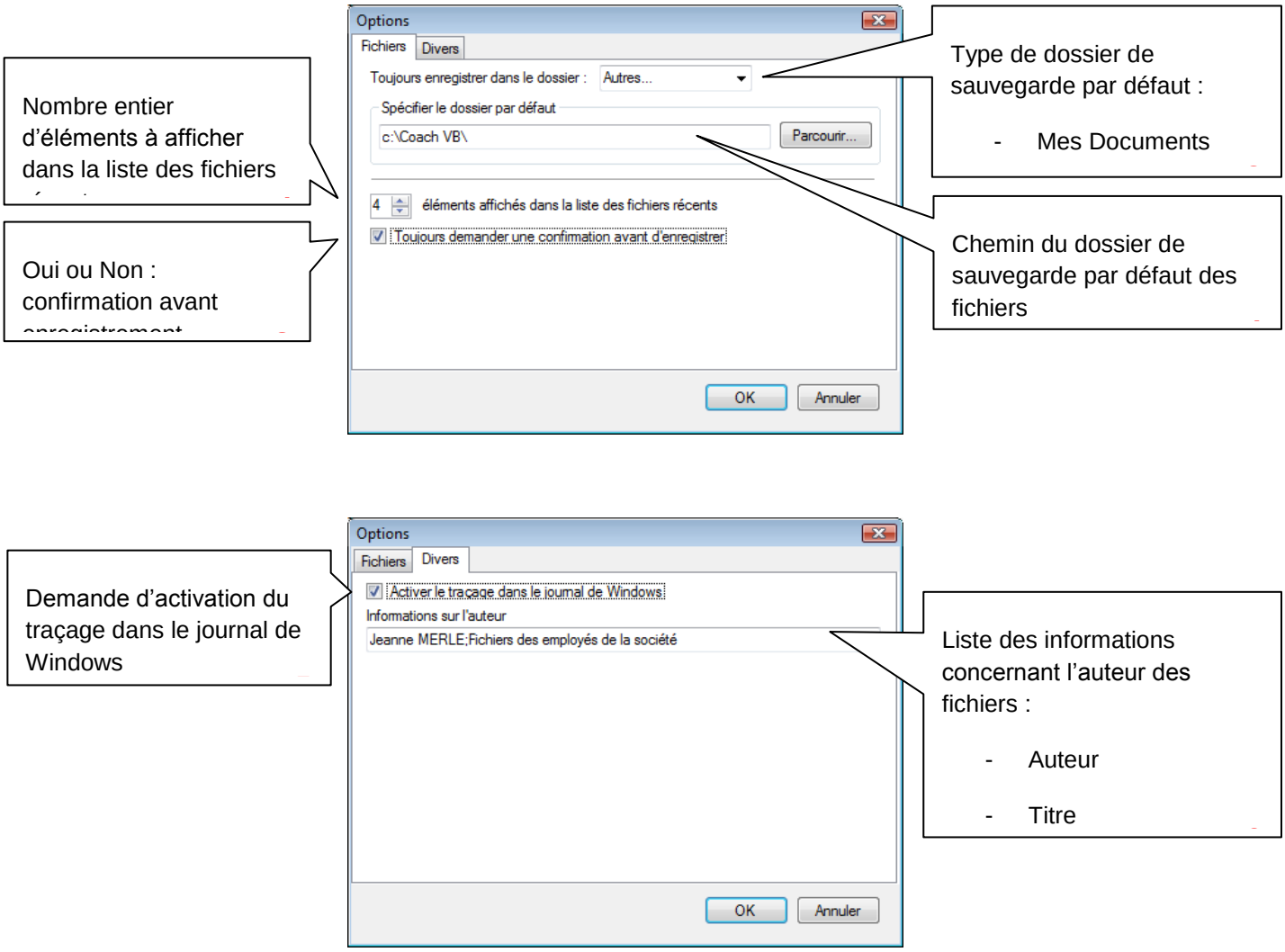
UTILISER LES TYPES DE DONNÉES COMMUNS

Jusqu'à maintenant nous avons vu comment créer et afficher des boîtes de dialogue. Dans la suite de cet atelier, nous vous proposons d'explorer plus avant les types de données standards du langage VB.

Contexte fonctionnel

Du point de vue fonctionnel, nous avons maintenant une belle boîte de dialogue *Options* pour que l'utilisateur puisse configurer l'application à sa façon. Mais si l'on veut prendre en compte sa configuration, il faut maintenant enregistrer les informations quelque part…

On veut enregistrer les informations suivantes :

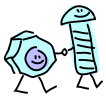


Du point de vue technique, enregistrer les options dans des variables mémoires revient donc à créer des variables avec le type de données approprié pour chacun des éléments de la boîte de dialogue *Options*.

Lorsqu'on définit une nouvelle variable, il faut se poser la question de sa *durée de vie*. C'est-à-dire quelle est la période d'exécution au cours de laquelle la variable doit être valide et utilisable ?

En théorie, il faudrait préserver la configuration des options au-delà de l'exécution de l'application pour que l'utilisateur puisse retrouver ses paramètres. Dans cet exercice, nous allons procéder plus simplement, c'est-à-dire que nous allons enregistrer les paramètres de l'utilisateur en mémoire dans des variables pour qu'il puisse au moins les retrouver le temps de l'exécution de l'application.

En conclusion, ces variables doivent être utilisables pendant toute la durée de l'exécution de l'application.



Ce qui nous amène à la question « où déclarer ces variables dans le projet » ? Dans la programmation orientée objet, tout fait nécessairement partie d'un objet.

Quels sont les objets dont nous disposons dans notre programme ?

Notre application **Coach.Editeur** contient essentiellement trois grandes classes d'objet différentes : Ce sont les trois classes **Main**, **OptionsDialog** et **SplashScreen**. Nos variables doivent donc être définies dans l'une ou l'autre de ces trois classes :

- dans la classe **SplashScreen**, on oublie…
- dans la classe **OptionsDialog** non plus, parce qu'un nouvel objet **formOptions** de type **OptionsDialog** est instancié à chaque demande d'affichage de la boîte, ce qui ne permettrait pas d'utiliser les variables pendant toute la durée de vie de l'application.

```
Private Sub OptionsToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles OptionsToolStripMenuItem.C
    Dim formOptions As OptionsDialog
    formOptions = New OptionsDialog
    Dim result As DialogResult
    result = formOptions.ShowDialog()
End Sub
```

- En revanche, la définition de nos variables dans la classe **Main** a du sens. Cela nous permettrait de recharger les contrôles d'affichage de la boîte de dialogue *Options* à chaque nouvel affichage et donc de les préserver tout au long de l'exécution de l'application.

Où précisément doit-on déclarer ces variables dans la classe Main ?

C'est la question de la *portée* des variables. En effet, lorsque vous déclarez une variable, vous devez vous assurer que toutes les parties du code qui en auront besoin y ont accès. Cette partie de code qu'il faut chercher à délimiter le plus précisément possible pour limiter l'utilisation des ressources de l'application, s'appelle la *portée* de la variable.

Si vous déclarer une variable dans une procédure, par exemple dans le gestionnaire d'évènement **OptionsToolStripMenuItem_Click** qui affiche la boîte de dialogue *Options*, la portée de la variable est limitée au code de la procédure. Si vous avez besoin de la variable depuis une autre partie du code du formulaire Main, ce qui va être notre cas, vous ne pouvez pas la déclarer là.

La portée d'une variable dépend donc de où vous la déclarez. Nous allons déclarer nos variables options au niveau global de la classe **Main** pour qu'elles soient utilisables depuis n'importe quelle portion de code de la classe.

Remarque : Vous pouvez tout à fait déclarer deux variables du même nom dans un programme du moment qu'elles n'ont pas la même portée.

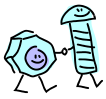
Pour creuser la question de :

- la durée de vie des variables :

http://msdn.microsoft.com/fr-fr/library/35styb3a.aspx

- la portée des variables :

http://msdn.microsoft.com/fr-fr/library/1t0wsc67.aspx



Vous pouvez contrôler également la portée d'une variable en précisant un *niveau d'accès* au moment de sa déclaration. Dans notre cas, nous allons utiliser le mot clé **Dim** sans préciser de niveau d'accès, ce qui revient à déclarer une variable privée, c'est-à-dire qu'elle sera accessible uniquement dans le bloc où nous allons la déclarer, donc dans la classe.

Nous aurons l'occasion de revenir sur cette notion plus tard dans ce tutorial lorsque nous aborderons la manipulation des objets.

Pour commencer à vous renseigner sur les différents niveaux d'accès :

http://msdn.microsoft.com/fr-fr/library/76453kax.aspx

Voici la liste des variables à déclarer. Nous vous proposons les noms et types de données suivants :

	Nom de la variable	**Type de données**
1	RecentFilListNumber	Decimal
2	ConfirmBeforeSave	Boolean
3	SaveDirectoryType	Type énuméré à définir
4	SaveDirectoryPath	String
5	TraceEnabled	Boolean
6	AuthorInfo	Tableau de String

Déroulement de l'exercice :

1. Avant toute chose, ajoutez une région pour délimiter la déclaration des variables au tout début de la classe **Main** :

Code VB

```
Public Class Main

#Region "Déclaration des variables privées de la classe"

#End Region

   Public Sub New…
   …
End Class
```

2. Ajoutez la déclaration des variables **1**, **2**, **4** et **5** de types élémentaires :

	Nom de la variable	Type de données
1	RecentFilListNumber	Decimal
2	ConfirmBeforeSave	Boolean
4	SaveDirectoryPath	String
5	TraceEnabled	Boolean

- Déclarez les variables comme suit :

Code VB

```
#Region "Déclaration des variables privées de la classe"
    'Déclaration des propriétés de la boîte des options
    Dim RecentFileListNumber As Short
    Dim ConfirmBeforeSave As Boolean
    Dim SaveDirectoryPath as String
    Dim TraceEnabled As Boolean
#End Region
```

Quelques remarques :

- Le type **Short** : il faut savoir qu'il existe de nombreux types de données numériques selon que vous voulez travaillez avec des nombres à virgule, ou des nombres négatifs, des petits ou des grands nombres… Sélectionnez le type le plus approprié pour maximiser les performances (notamment lors des opérations effectuées sur le nombre) et minimiser la place que votre variable va prendre dans la mémoire de l'application. Le type **Short** permet notamment de manipuler une plus petite plage d'entiers que le type **Integer**.
- Le type **Boolean** : une valeur booléenne permet de gérer des informations à deux états et est interprétée comme **True**(Vrai) ou **False**(Faux).
- Le type **String** : ce type de données permet de stocker une séquence de caractères, typiquement du texte dont on ne connait pas la longueur.

Retrouvez tous les types de données élémentaires de Visual Basic, dont les types numériques ici :

http://msdn.microsoft.com/fr-fr/library/asz1fz8b.aspx

Pour consulter tous les types de données du langage Visual Basic :

http://msdn.microsoft.com/fr-fr/library/ffsdktda.aspx

Lors de la déclaration d'une variable, Visual Basic lui attribue immédiatement une valeur par défaut. Par exemple, une variable de type **Date** est automatiquement initialisée au premier janvier de l'année 0001 à l'heure de minuit.

Un bon truc est de prendre l'habitude d'initialiser vos variables au moment de leur déclaration. Ainsi cela diminue le risque d'avoir des surprises au moment de leur utilisation.

- Initialisez les variables comme suit :

Code VB

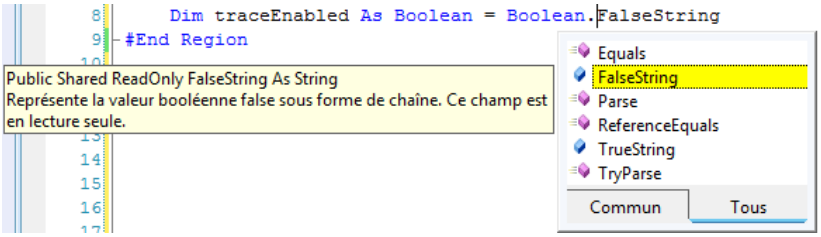
```
#Region "Déclaration des variables privées de la classe"
    'Déclaration des propriétés de la boîte des options
    Dim RecentFileListNumber As Short = 0
    Dim ConfirmBeforeSave As Boolean = Boolean.FalseString
    Dim SaveDirectoryPath as String = String.Empty
    Dim TraceEnabled As Boolean = Boolean.FalseString
#End Region
```

Notez qu'ici ces initialisations sont parfaitement inutiles puisque ce sont exactement les valeurs par défaut que Visual Studio attribuent aux variables des types concernés. Quoiqu'il en soit, cela rend le code très lisible et simplifie la maintenance…

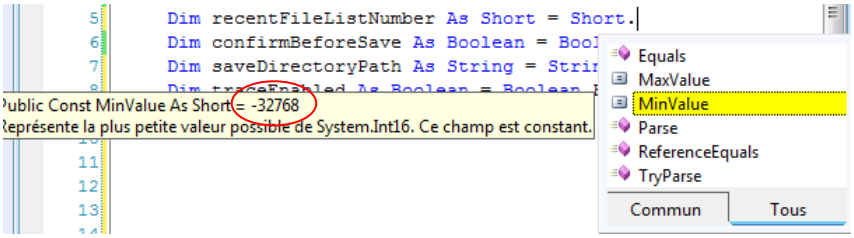
Dans son espace de nom **System**, le Framework.NET nous fournit des classes (ou structures) pour chacun des types de données du Système de type commun (**Boolean**, **String**, etc.). Ces classes sont très utiles pour manipuler des éléments du type

correspondant. Par exemple, elles servent ici à retrouver les valeurs d'initialisation standards des types correspondants.

Utilisez la fonctionnalité d'IntelliSense de Visual Studio pour voir quels sont les membres de ces classes et bénéficier d'explication.



Par exemple, vous constatez que pour un **Short**, on dispose des valeurs minimale (-32768) et maximale (+32768).

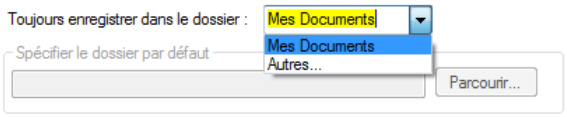


3. Ajoutez la déclaration des variables **3** et **6** de types plus complexes :

	Nom de la variable	**Type de données**
3	SaveDirectoryType	Type énuméré à définir

6	AuthorInfo	Tableau de String

Le type de dossier de sauvegarde par défaut (**SaveDirectoryType**) est fonction des valeurs que nous allons listées dans le contrôle **SaveDirectoryTypeComboBox**. A priori, nous n'en aurons que deux : soit l'utilisateur enregistre par défaut dans son dossier **Mes Documents**, soit dans un dossier dont il devra spécifier le chemin à l'aide de la zone de texte et du bouton **Parcourir…**.



Nous allons donc construire un nouveau type énuméré, sur la même base que la propriété **DialogResult** que nous avons vu précédemment dans cet atelier, mais cette fois-ci il s'agira d'un type personnalisé.

Pour consulter les spécificités d'une énumération :

http://msdn.microsoft.com/fr-fr/library/8h84wky1.aspx

- A la suite des déclarations précédentes, ajoutez la définition d'une énumération **DirectoryType** à l'aide du mot clé **Enum** :

Code VB

```
#Region "Déclaration des variables privées de la classe"

    'Déclaration des propriétés de la boîte des options

    Dim RecentFileListNumber As Short = 0
```

```
    Dim ConfirmBeforeSave As Boolean = Boolean.FalseString
    Dim SaveDirectoryPath as String = String.Empty
    Dim TraceEnabled As Boolean = Boolean.FalseString
    'Déclaration des énumérations
    Enum DirectoryType
      MyDocuments = 0
      Other = 1
    End Enum
#End Region
```

J'attire votre attention sur le fait que :

- une énumération ne peut pas être déclarée au niveau d'une procédure. Vous devez la définir nécessairement au niveau plus global d'une classe ou d'un espace de noms.
- vous n'êtes pas obligé de définir une valeur d'initialisation. Par défaut, Visual Basic assigne 0 à la première valeur, puis incrémente les valeurs de 1 en 1.

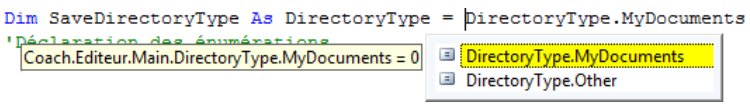
- Déclarez maintenant une variable du type énuméré :

Code VB

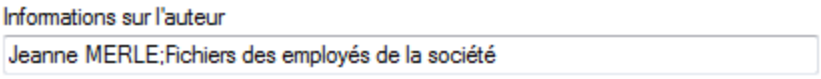
```
#Region "Déclaration des variables privées de la classe"
    'Déclaration des propriétés de la boîte des options
    Dim RecentFileListNumber As Short = 0
    Dim ConfirmBeforeSave As Boolean = Boolean.FalseString
```

```
    Dim SaveDirectoryPath as String = String.Empty
    Dim TraceEnabled As Boolean = Boolean.FalseString
    Dim SaveDirectoryType As DirectoryType = DirectoryType.MyDocuments
    'Déclaration des énumérations
    Enum SaveDirectoryType
      MyDocuments = 0
      Other = 1
    End Enum
#End Region
```

Avez-vous remarqué que l'IntelliSense reconnaît parfaitement votre type de données ?



Il nous reste à définir une variable de type tableau pour stocker les informations saisies dans la zone de texte **AuthorInfoTextBox**. L'idée est que l'utilisateur saisit dans la boîte de dialogue les différentes informations séparées par des points virgules. A nous de récupérer chacune de ces informations en les stockant dans un tableau.

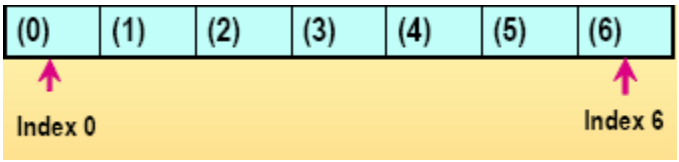


Il est clair que le procédé utilisé ici pour récupérer une telle liste d'informations manque d'élégance …mais par contre, c'est top pour jouer un peu avec la notion de tableau ☺.

En quoi consiste un *tableau* ?

Un tableau est une séquence d'éléments de données liés de manière logique et ayant le même type de données. Les éléments sont rangés au moyen d'un index (ou indice). Le premier élément du tableau est rangé à l'index 0 et ainsi de suite :

Exemple d'un tableau de 7 éléments (de 0 à 6) :



Pour définir un tel tableau on écrirait :

Dim MonTableau(6) As <TypeDeDonnéesDesElements>

Vous pouvez aussi définir des tableaux à plusieurs dimensions pour ranger des données sous forme de matrice multidimensionnelle.

- Dans notre cas, il s'agit d'un tableau unidimensionnel dont on ne connait pas à l'avance le nombre d'éléments, puisqu'on ne sait pas combien d'informations seront saisies par l'utilisateur. Déclarez le tableau comme suit :

Code VB

```
#Region "Déclaration des variables privées de la classe"

    'Déclaration des propriétés de la boîte des options

    Dim RecentFileListNumber As Short = 0

    Dim ConfirmBeforeSave As Boolean = Boolean.FalseString
```

```
    Dim SaveDirectoryPath as String = String.Empty
    Dim TraceEnabled As Boolean = Boolean.FalseString
    Dim SaveDirectoryType As DirectoryType = DirectoryType.MyDocuments
    Dim AuthorInfo() As String
    'Déclaration des énumérations
    Enum SaveDirectoryType
      MyDocuments = 0
      Other = 1
    End Enum
#End Region
```

Pour en savoir plus sur l'utilisation des tableaux en Visual Basic :

http://msdn.microsoft.com/fr-fr/library/wak0wfyt.aspx

Tant qu'on y est, profitons en aussi pour parler de la notion de *constante*.

Qu'est-ce qu'une *constante* ?

Une constante est une variable dont la valeur est immuable et ne change pas (elle reste *constante* justement) lors de l'exécution du programme. C'est très utile notamment pour remplacer la valeur correspondante par un nom significatif plus lisible et donc plus facile à maintenir.

Prenons l'exemple de l'utilisateur qui saisit les informations dans la zone de texte des informations sur l'auteur en les séparant par un point virgule. Plutôt que de manipuler un séparateur du type **";"**, nous pourrions travailler sur la base d'une constante avec un nom très explicite.

Notez les avantages suivants :

- Si vous avez besoin du séparateur à plusieurs reprises dans le programme, cela vous permet de le définir qu'une seule fois de manière centralisée et donc d'optimiser la clarté du programme tout en simplifiant sa maintenance.
- En plus une constante est moins gourmande en mémoire qu'une variable…alors pourquoi s'en priver ?

4. Ajoutez la déclaration d'une constante **SEPARATOR_SEMICOLON** à l'aide du mot clé **CONST** comme suit :

Code VB

```
#Region "Déclaration des variables privées de la classe"
    'Déclaration des propriétés de la boîte des options
    …
    End Enum
    'Déclaration des constantes
    Const SEPARATOR_SEMICOLON As String = ";"
#End Region
```

Pour en savoir plus sur les constantes en Visual Basic :

http://msdn.microsoft.com/fr-fr/library/cyxe49xw.aspx

Bon, nous disposons maintenant de toutes les variables utiles pour enregistrer la configuration de l'utilisateur dans la boîte *Options* de l'application.

Il n'y a plus qu'à coder la logique de l'application ! Ca va faire mal…

UTILISER LES STRUCTURES DU LANGAGE

Dans cet exercice, vous allez apprendre à :

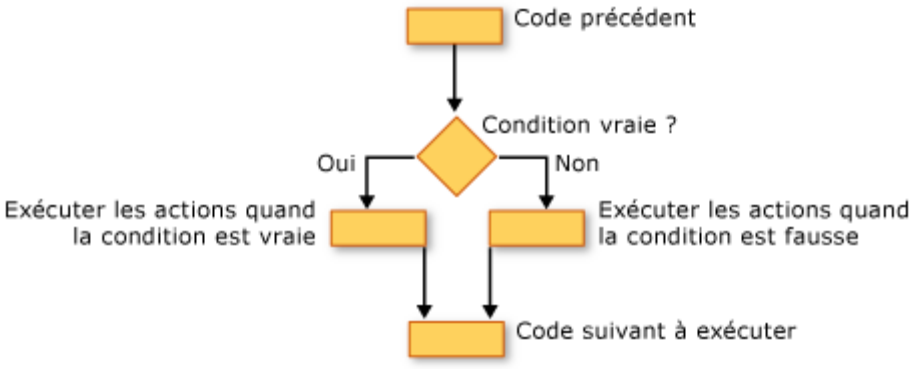
- Exécuter des instructions en fonction d'une ou plusieurs conditions (If…Then…Else),
- Tester plusieurs valeurs d'une expression (Select),
- Exécuter plusieurs actions sur un objet (With),
- Exécuter plusieurs instructions à plusieurs reprises (For),
- Exécuter plusieurs instructions pour chaque élément d'une collection ou d'un tableau (Foreach).

Objectif

Nous allons profiter de coder la logique de traitement de la boîte de dialogue *Options* de l'application pour découvrir quelques structures importantes du langage Visual Basic, telles que les structures de décision ou de boucle.

STRUCTURES DE DÉCISION

Une structure de décision est utile pour exécuter une ou plusieurs instructions en fonction du résultat d'une condition.

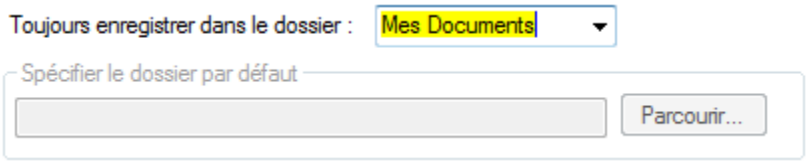


Contexte fonctionnel

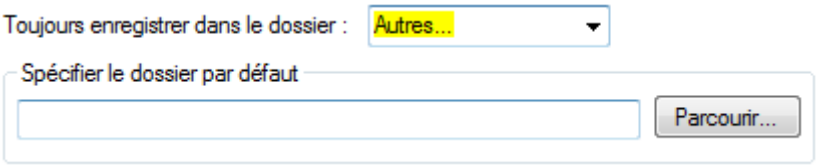
L'objectif de cet exercice est de commencer à coder le fonctionnement de l'option de configuration du dossier de sauvegarde par défaut des fichiers de notre éditeur.

Le fonctionnement doit être le suivant :

- Lorsque l'utilisateur préfère enregistrer par défaut dans le dossier **Mes Documents**, vous devez désactiver le groupe de contrôles encapsulés dans le conteneur **Spécifier le dossier par défaut** parce qu'ils ne sont d'aucune utilité.



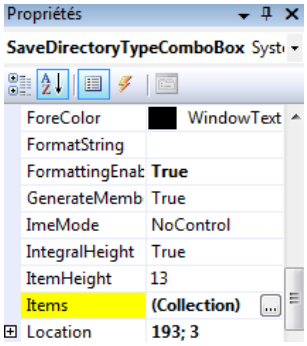
- En revanche, lorsque l'utilisateur sélectionne **Autres** dans la liste déroulante, vous devez activer le groupe de contrôles pour qu'il puisse saisir le chemin du dossier par défaut de son choix à l'aide de la boîte de recherche de Windows :



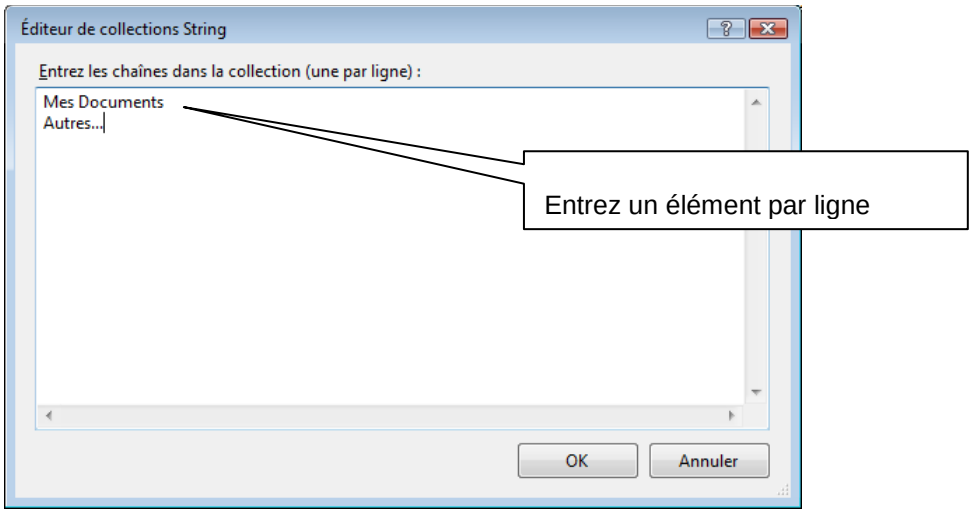
Déroulement de l'exercice :

1. Dans un premier temps, nous allons compléter le design de la boîte de dialogue *Options* pour qu'elle intègre la liste des dossiers de sauvegarde possibles dans le contrôle SaveDirectoryTypeComboBox :

   - Affichez le formulaire **Options** en mode **Design**.
   - Dans la page **Fichiers** du contrôle de type **TabControl**, sélectionnez le contrôle **SaveDirectoryTypeComboBox**.
   - Affichez les propriétés du contrôle.
   - Cliquez sur en face de la propriété **Items** pour éditer la collection des éléments de la liste.



   - Entrez deux éléments : **Mes Documents**, suivi à la ligne de **Autres…** :

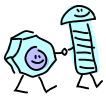


- Validez en cliquant **OK**.

2. Codez maintenant le comportement dynamique du groupe de contrôles SaveDirectoryPathGroupBox :

Il faut savoir que chaque contrôle standard Windows Form comporte une propriété **Enabled** de type **Boolean**. Il suffit de configurer cette propriété à la valeur **False** pour que le contrôle soit désactivé.

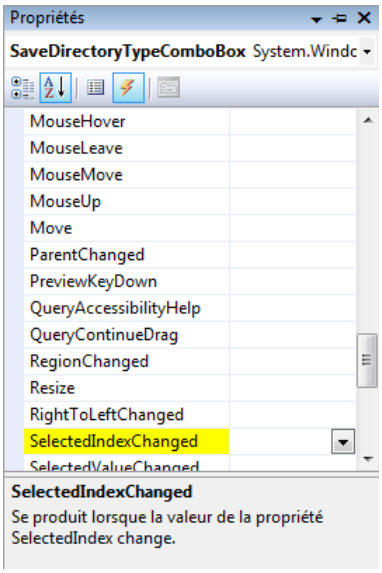
L'avantage d'avoir plongé tous les contrôles concernés dans un même contrôle conteneur est qu'il suffit de désactiver le conteneur pour que tous les contrôles qu'il contient soient désactivés du même coup !



Où allez-vous brancher le code de traitement de la sélection de dossier ?

Textuellement, vous voulez réagir **selon l'option sélectionnée** par l'utilisateur dans **la liste déroulante SaveDirectoryTypeComboBox**.

- Affichez la fenêtre de propriétés du contrôle **SaveDirectoryTypeComboBox**.
- Cliquez dans la barre d'outils de la fenêtre de propriétés pour consulter la liste des évènements disponibles pour cet élément.



Cherchez l'évènement qui correspond au changement de valeur sélectionnée dans la liste. Il s'agit de l'évènement par défaut du contrôle **SelectedIndexChanged**.

- Double cliquez à droite de l'évènement **SelectedIndexChanged** pour générer automatiquement un gestionnaire de cet évènement.

```
Private Sub SaveDirectoryTypeComboBox_SelectedIndexChanged( _
                    ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) _
                    Handles SaveDirectoryTypeComboBox.SelectedIndexChanged

End Sub
```

3. Utilisez la structure de décision **If…Then…Else** pour coder le traitement d'activation du groupe de contrôles en fonction de la valeur sélectionnée dans la liste déroulante :

   - Ajoutez le code suivant au gestionnaire d'évènement :

Code VB

```
Private Sub SaveDirectoryTypeComboBox_SelectedIndexChanged(…) _
                Handles SaveDirectoryTypeComboBox.SelectedIndexChanged
    If SaveDirectoryTypeComboBox.SelectedItem = "Mes Documents" Then
        'Désactiver les contrôles de sélection
        SaveDirectoryPathGroupBox.Enabled = False
    Else
        'Activer les contrôles de sélection
        SaveDirectoryPathGroupBox.Enabled = True
    End If
End Sub
```

Que fait la structure **If…Then…End If** ?

Elle exécute la (ou les) condition(s) située(s) après le mot clé **If**. Si la condition est remplie, c'est-à-dire qu'elle renvoie **True**, alors elle exécute le code situé juste après le mot clé **Then** jusqu'à la ligne marquée par **End If**.

Combinée avec le mot clé **Else**, elle peut également effectuer un traitement dans le cas où au contraire la condition renvoie la valeur **False**.

Dans notre cas, la condition à évaluer est la valeur de **SaveDirectoryTypeComboBox.SelectedItem** est-elle égale à **"Mes Documents"** ?

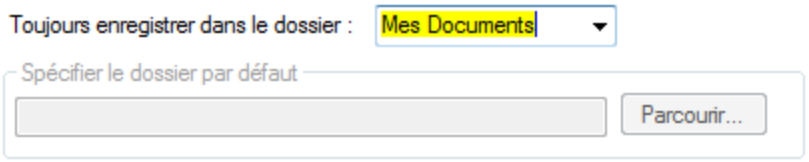
La propriété **SelectedItem** du contrôle de type **ComboBox** donne l'élément sélectionné dans la liste. Toute liste de ce type étant composée d'une paire d'éléments [Index, Valeur], vous pouvez également travailler sur la propriété **SelectedIndex** qui donne l'index de l'élément sélectionné.

Pour en savoir plus sur ces propriétés **SelectedItem** et **SelectedIndex** :

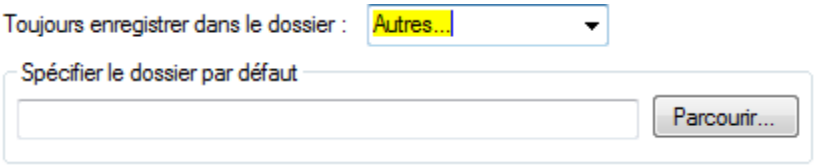
http://msdn.microsoft.com/fr-fr/library/system.windows.forms.combobox.selecteditem.aspx

4. Testez le fonctionnement de la sélection dans la liste :

   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application **en mode Débogage** en cliquant sur (ou touche F5).
   - Sélectionnez le menu **Outils** > **Options** de l'application.
   - Vérifiez que le groupe de contrôles est désactivé lorsque vous sélectionnez **Mes Documents** dans la liste :



   - Et qu'il est actif dans le cas contraire :



Bravo ! Ca fonctionne !

Pour tout savoir sur les structures de décision :

http://msdn.microsoft.com/fr-fr/library/hh892482.aspx

STRUCTURES DE CONTRÔLE

L'objectif de cet exercice est de découvrir l'une des structures de contrôle très pratique du langage VB à savoir l'instruction **With**.

Contexte fonctionnel

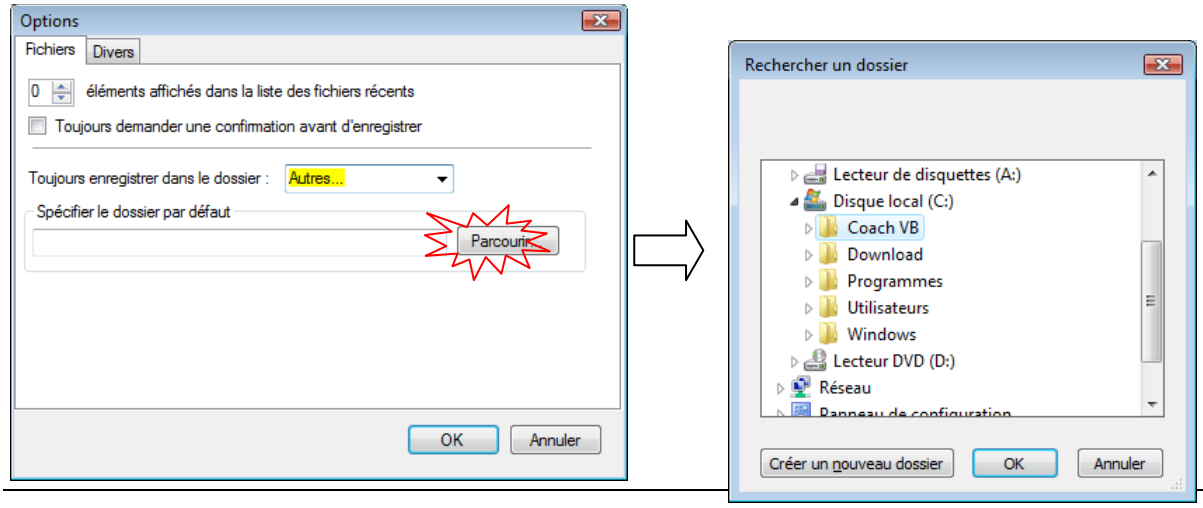
L'idée est de terminer le code de traitement de la configuration du chemin du dossier de sauvegarde par défaut des fichiers de l'application.

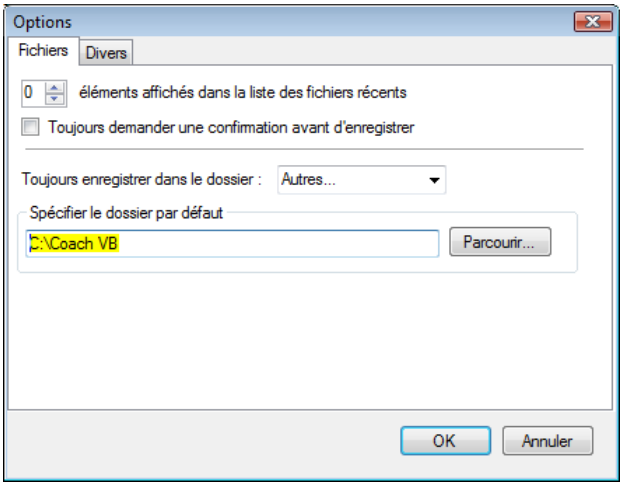
Le principe est le suivant :

- si l'utilisateur clique sur **OK** dans la boîte de recherche de dossiers, il faut récupérer ce chemin et l'afficher dans la zone de texte prévue à cet effet.

- en revanche s'il clique sur le bouton **Annuler**, la zone de texte n'a pas lieu d'être mise à jour.

Notez que l'utilisateur peut saisir un chemin directement dans la zone de texte sans utiliser le bouton **Parcourir…**.





1. Utilisez la structure de décision **If…Then** pour afficher le chemin du dossier choisi dans la zone de texte au retour de la boîte de dialogue de recherche des dossiers de Windows.

Rappelez-vous que nous avons vu que pour contrôler le code de fermeture d'une boîte de dialogue, il faut utiliser sa propriété **DialogResult**. Dans le premier exercice, c'est nous qui devions coder ce code de fermeture puisqu'il s'agissait d'une boîte de dialogue personnalisée. Dans le cas de la boîte standard **FolderBrowserDialog**, le code de fermeture est automatiquement renseigné ☺. Nous n'avons plu qu'à tester le code de fermeture pour renseigner la zone de texte en correspondance.

- Retrouvez le gestionnaire d'évènement en réponse au clic du bouton **SaveDirectoryPathButton** dans le fichier **OptionsDialog.vb** :

```
Private Sub SaveDirectoryPathButton_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles SaveDirectoryPathButton.Click
    SaveDirectoryPathFolderBrowserDialog.ShowDialog()
End Sub
```

- Modifiez le code comme suit :

Code VB

```
Private Sub SaveDirectoryPathButton_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles SaveDirectoryPathButton.Click
    If SaveDirectoryPathFolderBrowserDialog.ShowDialog() _
                                = DialogResult.OK  Then
        'Afficher le chemin du dossier sélectionné dans la zone de texte
        SaveDirectoryPathTextBox.Text = _
                        SaveDirectoryPathFolderBrowserDialog.SelectedPath
    End If
End Sub
```

Notez que c'est la propriété **SelectedPath** de la classe **FolderBrowserDialog** qui donne le chemin du dossier sélectionné dans la boîte de dialogue.

Franchement c'est trop long ce nom **SaveDirectoryPathFolderMachinCouette**, vous ne trouvez pas ?

Heureusement, il existe un moyen pour simplifier l'écriture tout en conservant un nom

malgré tout explicite ☺. Utilisez un bloc du type :

With <MonObjetQueJeNeDesigneQuUneSeuleFoisEnDebutDeBloc>

…

End With

A l'intérieur du bloc, vous pouvez invoquer n'importe quels membres de votre objet par un point (.) comme habituellement mais en étant dispensé de répéter le nom de l'objet lui-même !

- Modifiez le code pour utiliser l'instruction **With…End With**:

Code VB

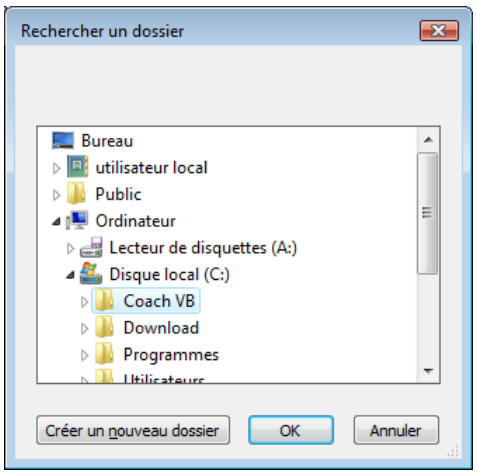
```
Private Sub SaveDirectoryPathButton_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles SaveDirectoryPathButton.Click
    With SaveDirectoryPathFolderBrowserDialog
      If .ShowDialog() = DialogResult.OK Then
        'Afficher le chemin du dossier sélectionné dans la zone de texte
        SaveDirectoryPathTextBox.Text = .SelectedPath
      End If
    End With
End Sub
```

Pour tout savoir sur l'instruction **With…End With** :

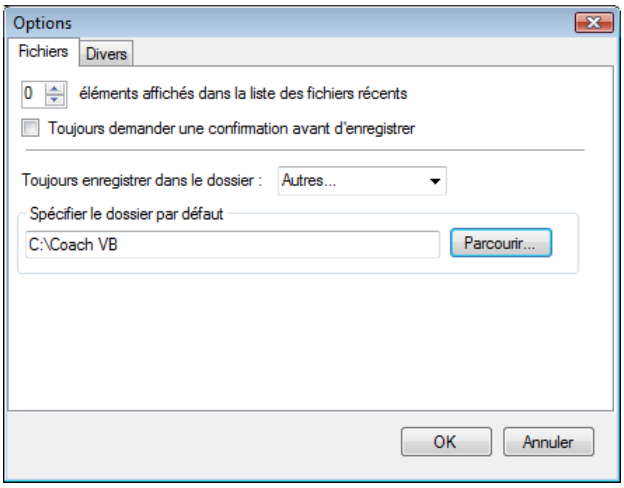
http://msdn.microsoft.com/fr-fr/library/wc500chb.aspx

2. Testez le fonctionnement de la boîte de dialogue de recherche :

   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application **en mode Débogage** en cliquant sur (ou touche F5).
   - Sélectionnez le menu **Outils** > **Options** de l'application.
   - Sélectionnez **Autres** dans la liste **Toujours enregistrer dans le dossier** pour activer le groupe de contrôles **Spécifier le dossier par défaut**.
   - Cliquez **Parcourir…** pour sélectionner un dossier à l'aide de la boîte de recherche de Windows.
   - Sélectionnez un dossier quelconque.



   - Quittez la boîte en cliquant **OK**.
   - Vérifiez que le chemin du dossier choisi s'affiche dans la zone de texte.



Supposons qu'on veuille amener l'utilisateur sur sa structure de dossier **Mes Documents** par défaut. Comment faire ?

Le dossier racine à partir duquel démarre la recherche dans la boîte de dialogue est configurable via la propriété **RootFolder** du contrôle **FolderBrowserDialog**.

3. Initialisez la boîte de recherche sur son dossier **Mes Documents** dans le cas où aucun chemin n'est spécifié dans la zone de texte :

    - Modifiez le code du gestionnaire d'évènement **SaveDirectoryPathButton_Click** comme suit :

Code VB

```
Private Sub SaveDirectoryPathButton_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles SaveDirectoryPathButton.Click
    With SaveDirectoryPathFolderBrowserDialog
      If String.IsNullOrEmpty(SaveDirectoryPathTextBox.Text) Then
```

```
            'Si la zone de texte est vide, démarrez à la racine de Mes Documents
            .RootFolder = Environment.SpecialFolder.MyDocuments
        Else
            .SelectedPath = SaveDirectoryPathTextBox.Text
        End If
        If .ShowDialog() = DialogResult.OK Then
            'Afficher le chemin du dossier sélectionné dans la zone de texte
            SaveDirectoryPathTextBox.Text = .SelectedPath
        End If
    End With
End Sub
```

Plusieurs points intéressants ici :

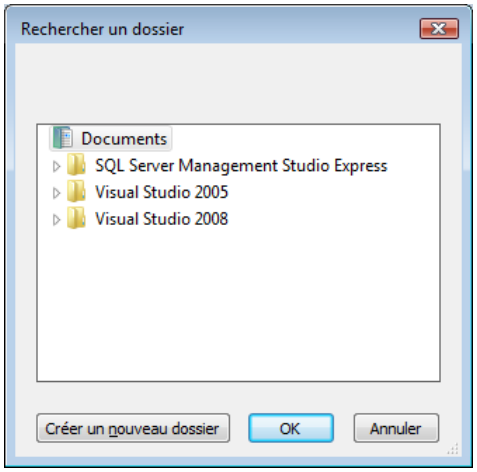
- Notez que pour tester si une chaîne de caractères est vide, il faut invoquer la méthode **IsNullOrEmpty** de la structure de type **String** en lui passant la chaîne à valider en paramètre.
- La classe **Environment** du Framework.NET nous donne tous les renseignements dont nous avons besoin sur l'environnement de l'utilisateur. Retrouvez par exemple tous les chemins des dossiers spéciaux du système grâce à l'énumération **Environment.SpecialFolder**.

Pour connaître les chemins proposés par l'énumération **SpecialFolder** :

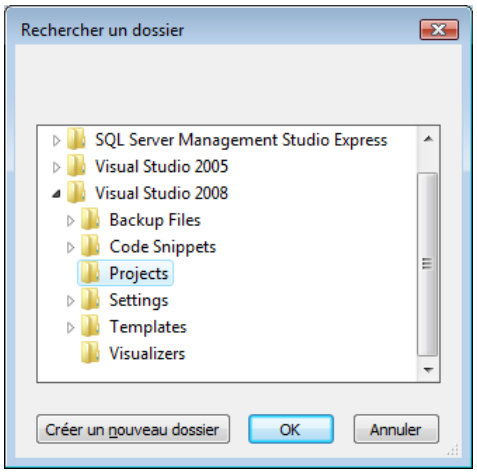
http://msdn.microsoft.com/fr-fr/library/system.environment.specialfolder.aspx

4. Testez l'initialisation de la boîte de recherche :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.

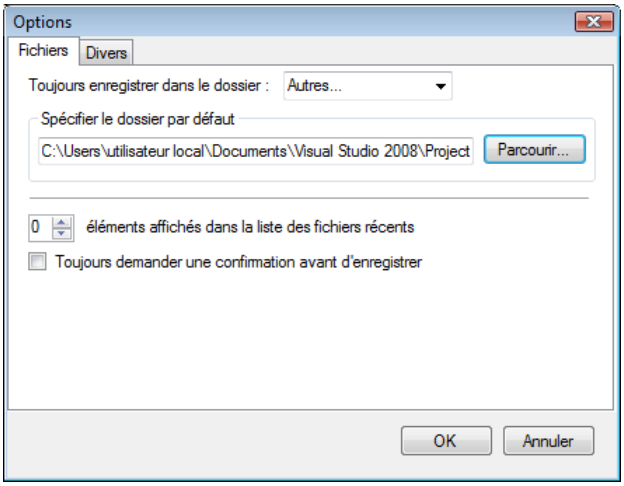
- Exécutez l'application **en mode Débogage** en cliquant sur (ou touche F5).
- Sélectionnez le menu **Outils** > **Options** de l'application.
- Sélectionnez **Autres** dans la liste **Toujours enregistrer dans le dossier** pour activer le groupe de contrôles **Spécifier le dossier par défaut**.
- Cliquez **Parcourir…** pour vérifier que le dossier par défaut est **Mes Documents**.



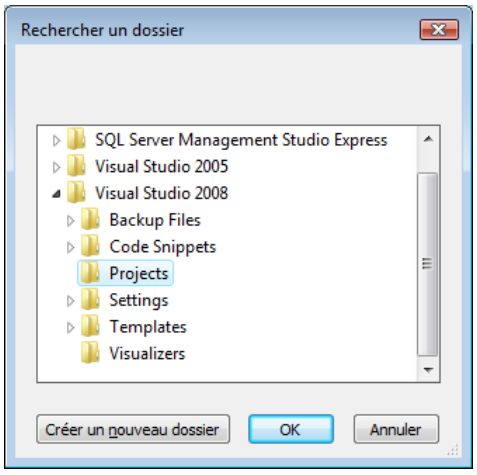
- Sélectionnez un dossier quelconque.



- Quittez la boîte en cliquant **OK**. Vérifiez que le chemin du dossier choisi s'affiche dans la zone de texte.

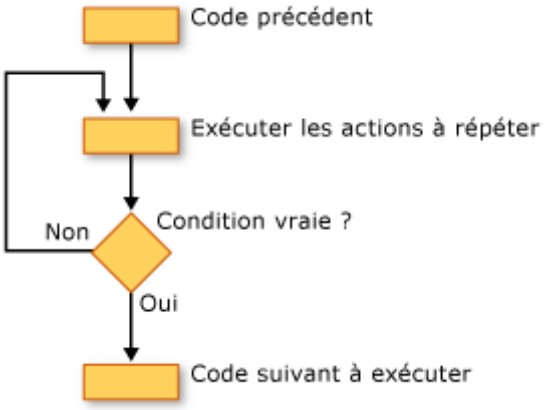


- Cliquez à nouveau sur le bouton **Parcourir…**.Vérifiez que la boîte s'initialise à la racine du chemin précédent.



STRUCTURES DE BOUCLE

Une structure de boucle est utile pour exécuter plusieurs fois de suite une ou plusieurs instructions, soit un nombre déterminé de fois, soit en fonction d'une certaine condition ou jusqu'à ce qu'une condition soit remplie.

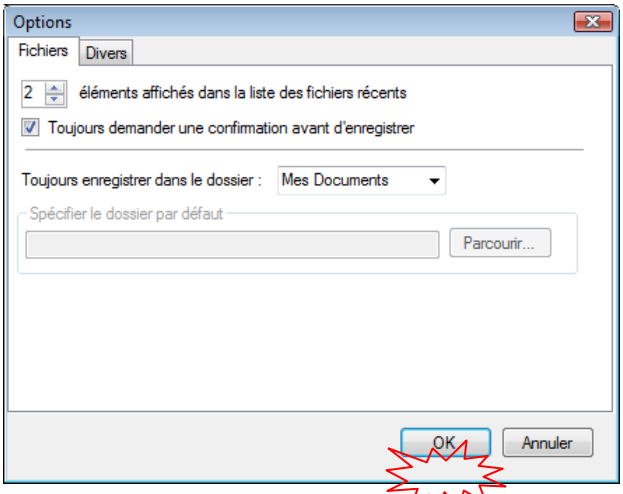


Dans cet exercice, nous allons coder l'initialisation et la sauvegarde des différentes options de la boîte de dialogue **OptionsDialog** à l'aide des variables que nous avons préparées à l'exercice précédent. Nous en profiterons au passage pour découvrir quelques unes des structures de boucle de VB.

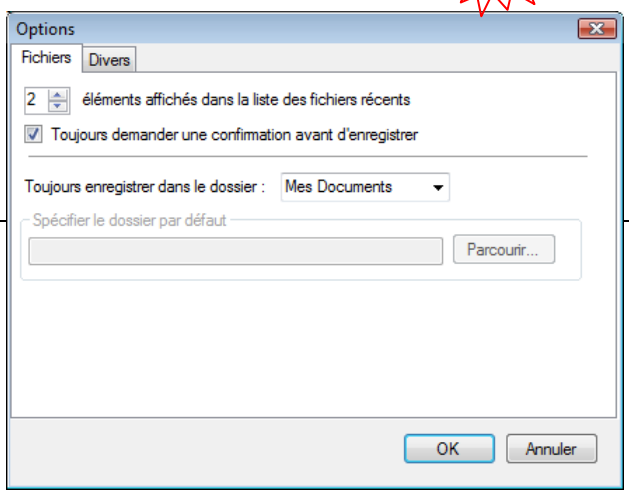
Contexte fonctionnel

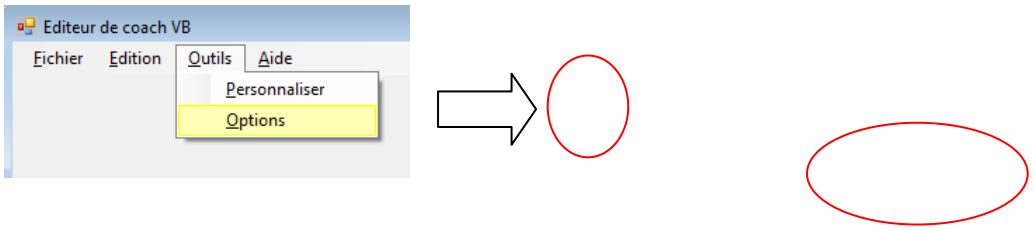
L'objectif de cet exercice est d'enregistrer les valeurs configurées par l'utilisateur dans la boîte de dialogue **Options** lorsqu'il clique le bouton **OK**.

Pour l'instant ces options ne servent pas directement à la logique de l'application. Elles seront prises en compte dans la suite de ce tutorial par exemple, pour les options de l'onglet **Fichiers**, au moment de la sauvegarde sur disque des données de notre éditeur.



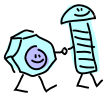
Durant l'exécution de l'application, à l'inverse, chaque fois que l'utilisateur réaffiche la boîte d'options, les valeurs qu'il a configurées doivent bien évidemment être restituées aux bons endroits dans la boîte de dialogue :





Déroulement de l'exercice :

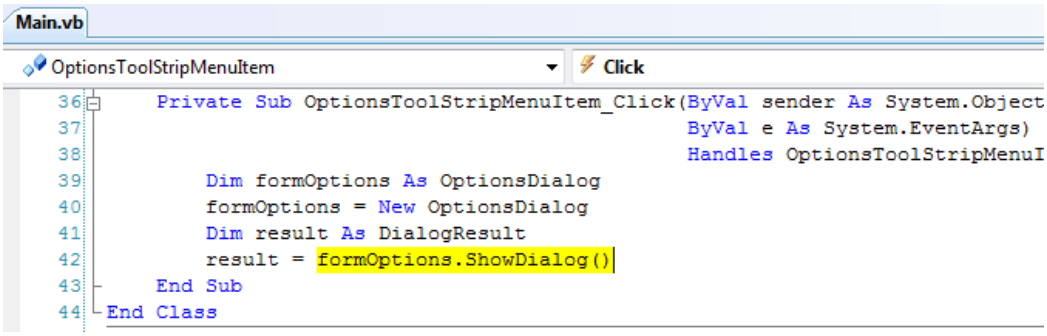
1. Repérez dans le code l'emplacement du code de sauvegarde des options de la boîte :



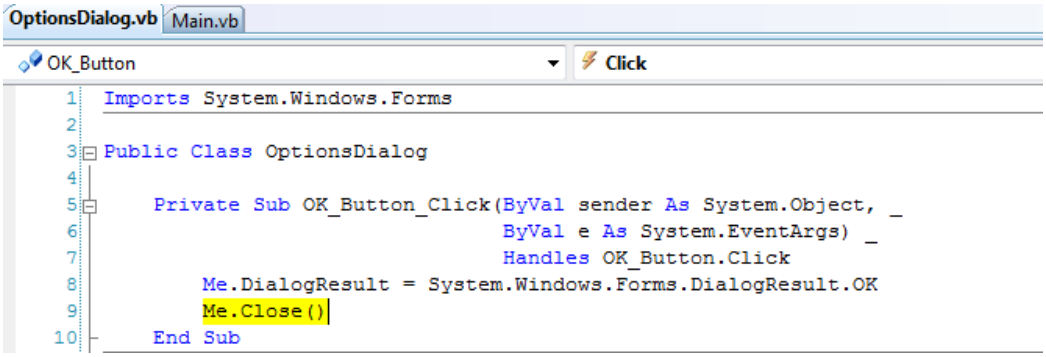
Votre premier réflexe est peut-être de coder la sauvegarde des options de la boîte de dialogue sur le clic du bouton **OK** du formulaire **OptionsDialog**. En effet, la portée des variables que nous avons définies dans la classe **Main** pourrait être configurée de façon à ce que les variables soient accessibles en dehors de la classe dans laquelle elles sont définies, donc depuis la classe **OptionsDialog**.

Mais d'un point de vue objet, ce ne serait pas une bonne approche dans la mesure où cette solution créerait un lien de dépendance entre la boîte de dialogue et sa fenêtre parent. Une bonne pratique consiste à penser les objets enfants en évitant à tout prix les liens de dépendance avec un objet parent de façon à pouvoir les réutiliser plus facilement dans d'autres contextes.

Nous allons donc coder la sauvegarde des options dans la classe **Main** juste après le retour de la méthode **ShowDialog** qui affiche la boîte **OptionsDialog**.



Oui, mais une fois que la boîte de dialogue est fermée, l'appel à la méthode **Close** de la classe **Form** a dû détruire complètement le formulaire en mémoire et quepouic pour récupérer les options configurées par l'utilisateur via les contrôles d'affichage de la boîte.



Oui, mais non…

Parce qu'en réalité cette méthode **Close** ne se comporte pas du tout comme ça justement dans le cas où le formulaire a été affiché via la méthode **ShowDialog**. En clair, le formulaire reste intact en mémoire tant que nous n'indiquons pas au runtime que nous n'en avons plus besoin.

Rassurez vous, quand bien même vous oubliez de le préciser au runtime, le Garbage Collector (GC) qui est la femme de ménage du Framework .NET finit toujours par passer et à faire le nettoyage qui s'impose.

Mais du coup, ce comportement nous arrange bien car nous allons pouvoir avoir accès

aux différents contrôles du formulaire **OptionsDialog** directement après le retour de la méthode **ShowDialog** dans la classe **Main** ☺ !

Pour tout savoir sur la méthode **Close** de la classe **Form** :

http://msdn.microsoft.com/fr-fr/vbasic/system.windows.forms.form.close.aspx

- Affichez le fichier de code **Main.vb**.
- Retrouvez le code d'affichage de la boîte de dialogue **OptionsDialog** dans le gestionnaire d'évènement associé au clic du menu **Outils** > **Options** de notre application :

```
Private Sub OptionsToolStripMenuItem_Click(ByVal sender As System.Object,
                                           ByVal e As System.EventArgs) _
                                           Handles OptionsToolStripMenuIt
    Dim formOptions As OptionsDialog
    formOptions = New OptionsDialog
    Dim result As DialogResult
    result = formOptions.ShowDialog()
End Sub
```

- Ajoutez une boucle de décision pour marquer la sauvegarde des options de la boîte dans le cas où le code de fermeture de la boîte *Options* est **OK**.

Code VB

```
Private Sub OptionsToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles OptionsToolStripMenuItem.Click
    Dim formOptions As OptionsDialog
    formOptions = New OptionsDialog
```

```
    Dim result As DialogResult

    result = formOptions.ShowDialog()

    If result = Windows.Forms.DialogResult.OK Then

       'Enregistrer les options configurées dans la boîte par l'utilisateur

    End If

End Sub
```

Même si pour la maintenance, c'est plutôt bien de favoriser l'écriture des lignes de code de la manière la plus détaillée possible, le langage VB permet aussi d'être plus concis, ce qui ne présente pas que des inconvénients ☺. Par exemple, la déclaration et l'instanciation de la variable **formOptions** peut se faire en une seule ligne comme ceci :

```
Dim formOptions As OptionsDialog = New OptionsDialog
```

Et vous pourriez vous passer de la variable **result** en écrivant directement :

```
Private Sub OptionsToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles OptionsToolStripMenuItem.C
    Dim formOptions As OptionsDialog = New OptionsDialog
    If formOptions.ShowDialog() = Windows.Forms.DialogResult.OK Then
        'Enregistrer les options configurées dans la boîte par l'utilisateur
    End If
End Sub
```

2. Repérez maintenant de la même façon l'emplacement du code d'initialisation des options de la boîte :

Le chargement des contrôles d'affichage de la boîte de dialogue doit se faire juste avant l'appel à la méthode **ShowDialog**.

- Ajoutez un commentaire pour marquer l'emplacement dans le code comme suit :

```
Private Sub OptionsToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles OptionsToolStripMenuItem.C
    Dim formOptions As OptionsDialog = New OptionsDialog
    'Chargement des options dans la boîte de dialogue
    If formOptions.ShowDialog() = Windows.Forms.DialogResult.OK Then
        'Enregistrer les options configurées dans la boîte par l'utilisateur
    End If
End Sub
```

3. Créez une procédure pour gérer la sauvegarde des options :

   - Ajoutez une nouvelle région à la suite de la déclaration des variables de la classe **Main** intitulée par exemple « **Traitement de la boîte de dialogue Options** » :

Code VB

```
Public Class Main


#Region "Déclaration des variables privées de la classe"
…
#End Region


#Region "Traitement de la boîte de dialogue Options"


#End Region
```

   - Dans cette région, créez une procédure **SaveOptions** :

Code VB

```
#Region "Traitement de la fenêtre d'options"

    Private Sub SaveOptions(ByVal form As OptionsDialog)

    End Sub

#End Region
```

C'est quoi une procédure ?

Une procédure regroupe une série d'instructions qui peut être utilisée à plusieurs reprises dans le code. C'est un peu comme quand vous donnez votre repassage à faire à une tierce personne (comment ? vous le faites vous-même ? C'est ça je vais vous croire…).

Du coup, il faut lui transmettre quelques consignes. Cela s'appelle des *paramètres*. Et si la personne a quelque chose à vous dire en retour, on dit que la procédure renvoie une *valeur de retour*. Pour que le repassage démarre, vous appelez la personne par son nom.

<Valeur retour> = <Nom procédure>(<Liste paramètres>)

Il faut savoir qu'il existe trois types de procédure dans le langage VB :

- les procédures **Function** qui effectue des actions puis retourne une valeur au programme appelant,
- les procédures **Sub** qui ne renvoient aucune valeur après l'exécution des actions,
- et les procédures **Property** que vous aurons l'occasion d'explorer plus tard dans ce tutorial lorsque nous travaillerons avec des objets.

Comme nous allons devoir récupérer l'état des contrôles de la boîte de dialogue, nous devons transmettre à la procédure l'objet correspondant à notre instance de la boîte

OptionsDialog en tant que paramètres.

Le mot clé **ByVal** indique la manière dont est transmise la donnée par l'appelant à la procédure. Il y a deux façons de passer des paramètres à une procédure : passage par valeur (**ByVal**) ou par référence (**ByRéf**).

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

- Dans le cas du passage par valeur, l'appelant transmet à la procédure une *copie distincte* de la donnée. Quand la procédure est terminée, la copie est détruite. Du coup, si les actions de la procédure modifient la donnée, puisqu'il s'agit d'une copie, la donnée initiale dans l'appelant reste inchangée au sortir de la procédure.

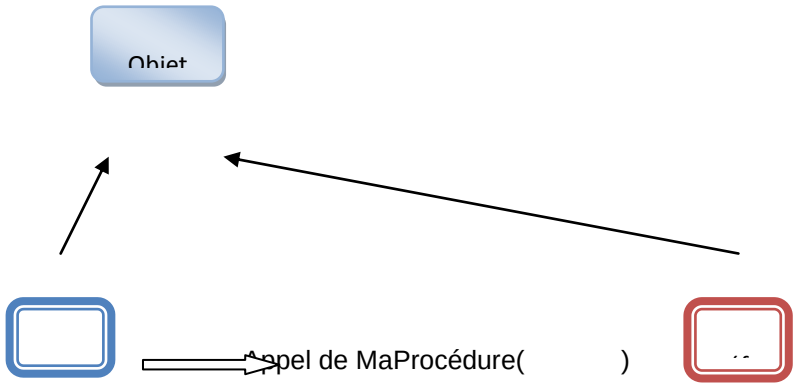
Si MaProcédure modifie angée.

- Dans le cas du passage par référence, c'est une référence (qui représente la localisation de la donnée en mémoire) qui est transmise à la procédure. C'est un peu comme si vous transmettiez à la procédure une deuxième porte d'accès à la donnée. Du coup toute modification effectuée durant les actions de la procédure modifie la donnée initiale de l'appelant, même si au sortir de la procédure la deuxième porte est détruite.

Si MaProcédure agit sur , pacte aussi la donnée dans l'appelant.

Attention toutefois aux types de données par référence. C'est le cas typiquement si vous manipulez des objets.

- Passage par valeur d'un type référence :

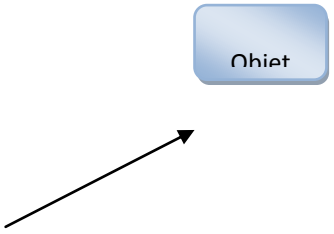


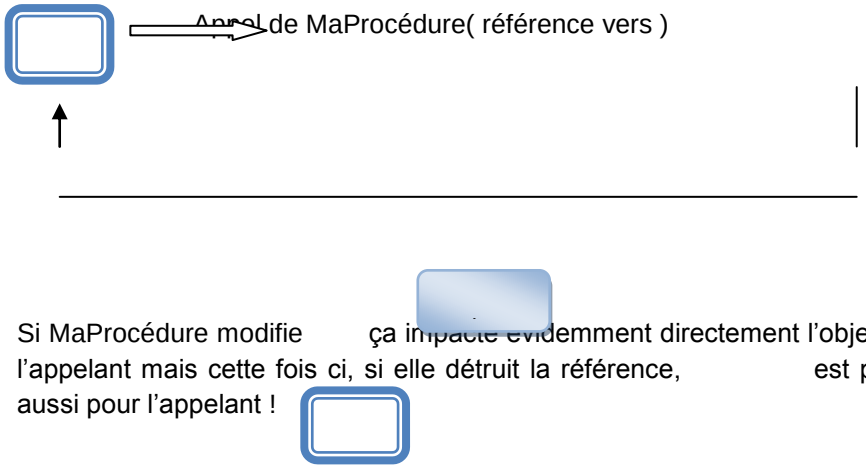
Si MaProcédure modifie ça impacte directement l'objet dans l'appelant puisque les deux références pointent sur le même objet mémoire.

Par contre, si MaProcédure détruit ence

est toujours intacte dans l'appelant.

- Passage par référence d'un type référence :



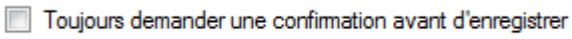


Pour en savoir plus sur les procédures **Sub** :

http://msdn.microsoft.com/fr-fr/library/dz1z94ha.aspx

Pour tout savoir sur le passage de **paramètres par valeur** ou **par référence** : http://msdn.microsoft.com/fr-fr/library/ddck1z30.aspx

- Ajoutez le code de sauvegarde de la demande de confirmation :

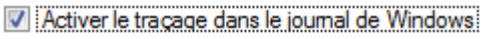


L'état d'un contrôle **CheckBox** est donné par sa propriété **Checked** qui a la valeur **True** lorsque la case est cochée et **False** sinon.

Code VB

```
#Region "Traitement de la fenêtre d'options"
    Private Sub SaveOptions(ByVal form As OptionsDialog)
        ConfirmBeforeSave = form.ConfirmBeforeSaveCheckBox.Checked
    End Sub
#End Region
```

- Ajoutez le code de sauvegarde de l'activation du traçage dans le journal de Windows :



Code VB

```
#Region "Traitement de la fenêtre d'options"
    Private Sub SaveOptions(ByVal form As OptionsDialog)
        ConfirmBeforeSave = form.ConfirmBeforeSaveCheckBox.Checked
        TraceEnabled = form.TraceCheckBox.Checked
    End Sub
#End Region
```

- Ajoutez le code de sauvegarde du nombre d'éléments dans la liste des fichiers récents :

Attention !

L'IntelliSense nous dit que la propriété **Value** qui donne la valeur d'un contrôle **NumericUpDown** est une donnée de type **Decimal**.

```
form.RecentFileListNumericUpDown.Value
```

Public Property Value() As Decimal
Obtient ou définit la valeur assignée à la zone de sélection numérique (également appelée contrôle up-down).

Or nous avons choisi de définir une variable de type **Short** dans la classe **Main** pour sauvegarder cette information. La solution la plus simple serait de modifier le type de la donnée que nous avons utilisé pour la variable de stockage. Mais sachez qu'il est possible aussi de *convertir* une donnée pour la forcer à apparaître sous la forme d'un autre type de données. Attention à la perte d'information qui pourrait être occasionnée !!

Pour ce faire, nous allons utiliser la fonction de conversion du type **Short**, appelée **CShort**, de façon à forcer le format du résultat dans le type choisi pour notre variable.

Pour en savoir plus sur les fonctions de conversion de types de données :

http://msdn.microsoft.com/fr-fr/library/s2dy91zy.aspx

Code VB

```
#Region "Traitement de la fenêtre d'options"

    Private Sub SaveOptions(ByVal form As OptionsDialog)

        ConfirmBeforeSave = form.ConfirmBeforeSaveCheckBox.Checked

        TraceEnabled = form.TraceCheckBox.Checked

        RecentFileListNumber = _
```

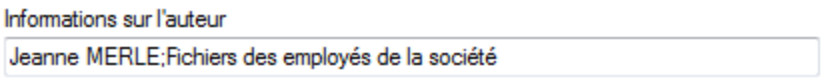
```
            CShort( form.RecentFileListNumericUpDown.Value)
    End Sub
#End Region
```

C'est qu'on est quand même habitué à mieux… est-ce que VB n'aurait pas pu s'en sortir tout seul en convertissant automatiquement la valeur dans le type attendu ?

Conversion *implicite* ou *explicite* ?

C'est vrai que vous pouvez laisser VB s'occuper de la conversion c'est-à-dire sans avoir besoin d'invoquer une fonction de conversion. On parle alors de conversion **implicite** alors que nous avons réalisé une conversion **explicite** en utilisant *explicitement* une fonction de conversion bien déterminée. Mais une conversion implicite est plus lourde puisqu'elle donne du travail supplémentaire au runtime et elle est susceptible de produire des résultats inattendus en pleine exécution ! Dans certain cas, elle n'est d'ailleurs pas autorisée et le compilateur vous en informe…

- Ajoutez le code de sauvegarde des informations sur l'auteur :



Rappelez-vous, nous avons décidé de sauvegarder cette information sous la forme d'un tableau de chaîne de caractères appelé **AuthorInfo**, dont chacun des éléments est l'une des informations séparées par un point virgule dans le texte de la zone de saisie. Nous avons d'ailleurs défini une constante **SEPARATOR_SEMICOLON** pour représenter la chaîne caractérisant le point virgule.

Heureusement pour nous, VB n'est pas en manque lorsqu'il s'agit de gérer les chaînes de caractère. Si vous voulez enlever un morceau d'une chaîne, comparer deux chaînes, supprimer les espaces blancs, ou que sais-je, comme c'est le cas ici, découper la chaîne en plusieurs morceaux en fonction d'un séparateur, il suffit de retrouver la fonction de

traitement adéquate parmi la liste des fonctions de traitement de caractères de VB.

Dans notre cas, il s'agit de la fonction **Split** qui découpe une chaine en suivant le séparateur fourni en paramètre et retourne un tableau de chaînes.

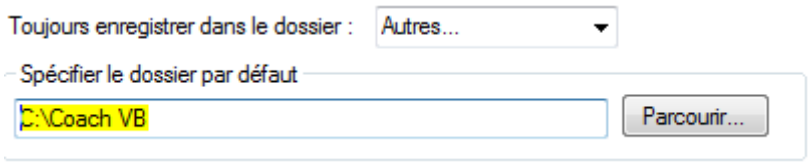
Consulter la liste des manipulations de chaînes ici :

http://msdn.microsoft.com/fr-fr/library/e3s99sd8.aspx

Code VB

```
#Region "Traitement de la fenêtre d'options"
    Private Sub SaveOptions(ByVal form As OptionsDialog)
        ConfirmBeforeSave = form.ConfirmBeforeSaveCheckBox.Checked
        TraceEnabled = form.TraceCheckBox.Checked
        RecentFileListNumber = _
                CShort( form.RecentFileListNumericUpDown.Value)
        AuthorInfo =  _
            form.AuthorInfoTextBox.Text.Split(SEPARATOR_SEMICOLON)
    End Sub
#End Region
```

- Reste le code de sauvegarde des informations de dossier. Utilisez la structure de décision **If…Then...End If** pour enregistrer le dossier sélectionné par l'utilisateur dans la variable **SaveDirectoryPath** :



Code VB

```
#Region "Traitement de la fenêtre d'options"
    Private Sub SaveOptions(ByVal form As OptionsDialog)
        …
        If form.SaveDirectoryTypeComboBox.SelectedIndex = _
                                        DirectoryType.MyDocuments Then
            SaveDirectoryPath = _
                    My.Computer.FileSystem.SpecialDirectories.MyDocuments
        Else
            SaveDirectoryPath = form.SaveDirectoryPathTextBox.Text
        End If
    End Sub
#End Region
```

Notez que le chemin du dossier **Mes Documents** de l'utilisateur est donné par l'objet **My.Computer** que nous avons déjà rencontré à l'atelier précédent.

(A ne pas confondre avec **Environment.SpecialFolder.MyDocuments** que nous avons vu dans le premier exercice qui est une énumération qui caractérise le type de dossier).

Qu'en est-il si la liste **SaveDirectoryTypeComboBox** comprenait plus de deux valeurs ? On ne pourrait plus se contenter de la structure **Else** pour déterminer le bon chemin de dossier.

Il existe une autre structure de décision qui permet de vérifier même dans le bloc **Else** qu'une condition est valide : l'instruction **ElseIf**.

Cela donne le code suivant :

Code VB

```
#Region "Traitement de la fenêtre d'options"
   Private Sub SaveOptions(ByVal form As OptionsDialog)
      …
      If form.SaveDirectoryTypeComboBox.SelectedIndex =  _
                                        DirectoryType.MyDocuments Then
         SaveDirectoryPath = _
                  My.Computer.FileSystem.SpecialDirectories.MyDocuments
      ElseIf form.SaveDirectoryTypeComboBox.SelectedIndex =  _
                  DirectoryType.Other Then
         SaveDirectoryPath = form.SaveDirectoryPathTextBox.Text
      End If
   End Sub
#End Region
```

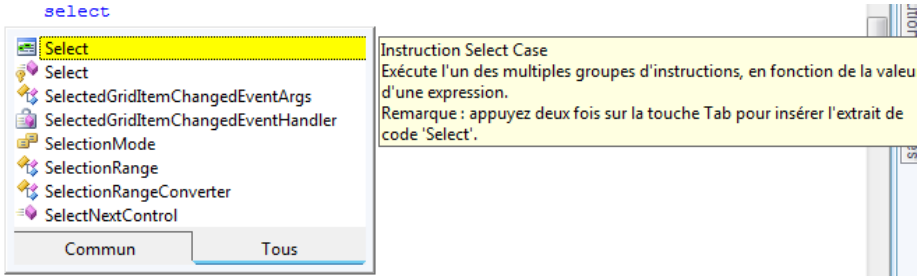
S'il est possible d'enchaîner les **ElseIf** dans le cas où l'on aurait beaucoup de valeurs dans la **ComboBox**, il faut dire que cela finirait par donner un code trop lourd.

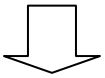
Du coup, il existe une autre structure de décision qui permet de tester plusieurs valeurs d'une expression avant de coder une action : l'instruction **Select**. Elle permet de

comparer une expression à plusieurs valeurs différentes et d'exécuter des actions en correspondance. Dans notre cas nous devons comparer la valeur de l'index de la sélection avec l'ensemble des index possibles de la liste déroulante.

Pensez à utiliser les **extraits** (snippets) de VB fournis dans Visual Studio pour coder les structures. Il suffit de taper le premier mot clé de la structure puis d'appuyer la touche **TAB**, et le bloc de code type apparaît automatiquement :

Tapez le mot clé **Select** puis enfoncer la touche **TAB**





```
Select Case VariableName
    Case 1

    Case 2

    Case Else

End Select
```

Pour tout savoir sur **Select** :

http://msdn.microsoft.com/fr-fr/library/ms172856.aspx

- Modifiez le code pour qu'il utilise la structure de décision **Select** :

Code VB

```
#Region "Traitement de la fenêtre d'options"
    Private Sub SaveOptions(ByVal form As OptionsDialog)
        …
        Select Case form.SaveDirectoryTypeComboBox.SelectedIndex
            Case DirectoryType.MyDocuments
                SaveDirectoryPath = _
                        My.Computer.FileSystem.SpecialDirectories.MyDocuments
            Case DirectoryType.Other
                SaveDirectoryPath = form.SaveDirectoryPathTextBox.Text
        End Select
    End Sub
#End Region
```

Sachez que chaque instruction **Case** peut aussi contenir une ou plusieurs valeurs, une plage de valeurs etc… Ici, on se sert de notre énumération **DirectoryType** qui facilite grandement la lisibilité du code. Vous pouvez aussi prévoir une dernière instruction **Case Else** pour gérer le cas où aucune des valeurs proposées dans les autres **Case** ne correspondraient.

- Enregistrez tous vos changements avant de poursuivre.

4. Avant de pouvoir tester le bon fonctionnement de la sauvegarde des options, nous allons écrire le code inverse qui initialise la boîte de dialogue *Options* avec les variables de stockage de la classe Main.

    - Ajoutez à la suite de la procédure **SaveOptions**, une nouvelle procédure nommée **LoadOptions** comme suit :

Code VB

```
#Region "Traitement de la fenêtre d'options"
    Private Sub SaveOptions(ByVal form As OptionsDialog)
        …
    End Sub
    Private Sub LoadOptions(ByVal form As OptionsDialog)
    End Sub
#End Region
```

- Ajoutez le code d'initialisation des deux cases à cocher :

Code VB

```
#Region "Traitement de la fenêtre d'options"
    Private Sub LoadOptions(ByVal form As OptionsDialog)
        form.ConfirmBeforeSaveCheckBox.Checked = ConfirmBeforeSave
        form.TraceCheckBox.Checked = TraceEnabled
    End Sub
#End Region
```

- Ajoutez le code d'initialisation de la liste des fichiers récents en prévoyant la conversion de type de données inverse à l'aide de la fonction **CDec** :

Code VB

```
#Region "Traitement de la fenêtre d'options"
  Private Sub LoadOptions(ByVal form As OptionsDialog)
      form.ConfirmBeforeSaveCheckBox.Checked = ConfirmBeforeSave
      form.TraceCheckBox.Checked = TraceEnabled
      form.RecentFileListNumericUpDown.Value =  _
                              CDec(RecentFileListNumber)
  End Sub
#End Region
```

- Ajoutez le code d'initialisation des informations de dossier. Dans ce sens, il faut également coder l'initialisation de la liste déroulante **SaveDirectoryTypeComboBox**.

Code VB

```
#Region "Traitement de la fenêtre d'options"
  Private Sub LoadOptions(ByVal form As OptionsDialog)
      form.ConfirmBeforeSaveCheckBox.Checked = ConfirmBeforeSave
      form.TraceCheckBox.Checked = TraceEnabled
      form.RecentFileListNumericUpDown.Value =  _
                              CDec(RecentFileListNumber)
    Dim resultComparison As Integer = _
              String.Compare(SaveDirectoryPath, _
              My.Computer.FileSystem.SpecialDirectories.MyDocuments)
    If resultComparison = 0 _
      Or String.IsNullOrEmpty(SaveDirectoryPath) Then
```

```
                form.SaveDirectoryTypeComboBox.SelectedIndex = _
                DirectoryType.MyDocuments
        Else
                form.SaveDirectoryTypeComboBox.SelectedIndex = _
                DirectoryType.Other
                form.SaveDirectoryPathTextBox.Text = SaveDirectoryPath
        End If
    End Sub
#End Region
```

Notez :

- l'utilisation de l'opérateur logique **Or** qui permet d'évaluer deux conditions et d'effectuer une action si l'une ou l'autre des conditions s'avère vraie.

  Retrouvez la liste de tous les opérateurs du langage VB ici :

  http://msdn.microsoft.com/fr-fr/library/f15ea9k8.aspx

- l'utilisation de la méthode **Compare** de la classe du Framework .NET correspondant au type **String** qui compare deux chaînes de caractères. Attention au résultat renvoyé par cette méthode qui peut surprendre un peu car il n'est pas binaire. Tous les détails ici : http://msdn.microsoft.com/fr-fr/library/84787k22(VS.85).aspx

- Reste le code de chargement de la zone de texte avec la liste des informations d'auteur du tableau de chaînes **AuthorInfo**.

C'est là que nous allons mettre en œuvre une première structure de boucle de VB ☺. En effet, il faut parcourir chaque élément du tableau et pour chacun d'eux, écrire une action qui ajoute l'information à la zone de texte dans la boîte *Options*. En somme il faut

réitérer l'exécution d'une action autant de fois que le tableau comporte d'éléments.

La boucle **For** permet justement d'exécuter une action en boucle. Elle incrémente un index sur un nombre d'occurrences données et exécute les actions à chaque incrément.

Ce qui donne :

Code VB

On recommence

```
#Region "Traitement de la fenêtre d'options"
  Private Sub LoadOptions(ByVal form As OptionsDialog)
    …
    For index As Integer = 0 To AuthorInfo.Count - 1
      form.AuthorInfoTextBox.AppendText( _
          AuthorInfo(index) & SEPARATOR_SEMICOLON)
    Next
  End Sub
#End Region
```

Plusieurs remarques :

- Pensez à l'IntelliSense pour générer un bloc **For** automatiquement !
- Vous devez définir un compteur en précisant son type (**index As Integer**) et sa valeur de départ (**= 0**). Le code contenu dans la structure de boucle est exécuté une première fois puis le mot clé **Next** marque le passage à l'itération suivante. Le compteur est incrémenté de 1 et le pointeur d'exécution remonte sur la première action de la boucle.

  Le code est alors ré exécuté autant de fois que d'incrément nécessaire pour que le compteur atteigne la valeur maximale donnée par le mot clé **To**. Une boucle **For** présume donc que vous connaissez à l'avance le nombre d'itération que vous voulez effectuer ! Ce qui est notre cas ici puisque la taille d'un tableau est donnée par sa propriété **Count**.
- Attention à l'indexation d'un tableau qui démarre toujours à **0** et se termine donc à **<Tableau>.Count – 1**…
- Pour accéder à un élément du tableau, il suffit d'indiquer le nom du tableau suivi de l'index de l'élément entre parenthèse. Le premier élément de notre tableau est **AuthorInfo(0)** puis **AuthorInfo(1)** etc….
- Notez l'utilisation de la méthode **AppendText** qui s'applique à toute variable de

type **String**. Elle permet de concaténer une chaîne de caractères à la suite de la chaîne courante.

Il existe une structure de boucle encore plus adaptée ici car nous fonctionnons sur un tableau d'éléments. Il s'agit de la structure **For Each** qui permet d'itérer sur chaque élément d'une collection d'objets ou d'un tableau plutôt que d'incrémenter un index. Elle est plus performante que la précédente, ce qui ne gâche rien.

- Remplacez le code précédent avec une boucle **For Each** comme suit:

Code VB

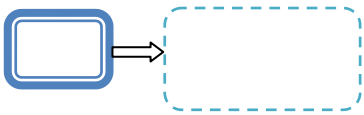
```
#Region "Traitement de la fenêtre d'options"
  Private Sub LoadOptions(ByVal form As OptionsDialog)
      …
      For Each Info As String In AuthorInfo
          form.AuthorInfoTextBox.AppendText(Info & _
                  SEPARATOR_SEMICOLON)
      Next
  End Sub
#End Region
```

Pour tout savoir sur les structures de boucle :

http://msdn.microsoft.com/fr-fr/library/ezk76t25.aspx

Lorsque vous utilisez un objet c'est-à-dire une variable de type référence comme c'est le cas de notre tableau **AuthorInfo**, vous devez toujours vous demander si la variable est bien associée avec un objet avant de vous lancer à l'utiliser.

Par exemple, dans notre cas, la toute première fois que nous allons afficher la boîte de dialogue *Options* et initialiser son contenu, le tableau **AuthorInfo** sera encore vide c'est-à-dire que la référence ne pointe sur aucun objet.



Et vous vous en doutez, c'est le genre de situation que le runtime d'exécution risque de ne pas apprécier. Il génère une belle exception pour indiquer qu'il a rencontré une référence nulle.

Comment vérifier qu'une référence est nulle ?

Il suffit de vérifier qu'elle n'est pas égale à **Nothing**. Il faut savoir que lorsqu'il s'agit de comparer deux références, VB fournit l'opérateur **Is** plutôt que de travailler sur la base de l'opérateur **=** standard.

Pour tout savoir sur l'utilisation de **Is** en tant qu'opérateur :

http://msdn.microsoft.com/fr-fr/library/kb136x1y.aspx

- Vérifiez à l'aide d'une structure de décision **If** que le tableau n'est pas vide avant de commencer à l'utiliser :

Code VB

```
#Region "Traitement de la fenêtre d'options"
```

```
  Private Sub LoadOptions(ByVal form As OptionsDialog)
    …
    If Not AuthorInfo Is Nothing Then
      For Each Info As String In AuthorInfo
        form.AuthorInfoTextBox.AppendText(Info & _
                    SEPARATOR_SEMICOLON)
      Next
    End If
  End Sub
#End Region
```

Bravo ! C'est presque terminé, il reste un tout petit détail à régler. En effet, lorsque la boucle traite le dernier élément du tableau elle ajoute un point virgule en trop à la suite de la zone de texte de la boîte *Options*. C'est embêtant car dans le processus inverse de sauvegarde du tableau, la méthode **Split** interpréterait le point virgule supplémentaire comme un dernier élément vide.

Une méthode simple consiste à supprimer le dernier point virgule juste après l'instruction **Next** c'est-à-dire une fois que le programme sort de la boucle. Pour cela nous allons utiliser une autre méthode de traitement de chaîne de caractères, **TrimEnd**, qui gère la suppression d'un morceau de chaîne à partir de sa fin.

- Ajoutez une ligne juste après l'exécution de la boucle pour supprimer le dernier point virgule :

Code VB

```
#Region "Traitement de la fenêtre d'options"
```

```
  Private Sub LoadOptions(ByVal form As OptionsDialog)
      …
      If Not AuthorInfo Is Nothing Then
          For Each Info As String In AuthorInfo
            form.AuthorInfoTextBox.AppendText(Info &  _
                                SEPARATOR_SEMICOLON)
          Next
          form.AuthorInfoTextBox.Text = _
              form.AuthorInfoTextBox.Text.TrimEnd(SEPARATOR_SEMICOLON)
      End If
  End Sub
#End Region
```

Vous voyez que la constante **SEPARATOR_SEMICOLON** est bien utile ! Cela fait déjà trois fois que vous l'utilisez à différents endroits dans le code, autant d'emplacements qu'il faudrait retoucher si le séparateur venait à changer et que vous n'aviez pas défini une constante …

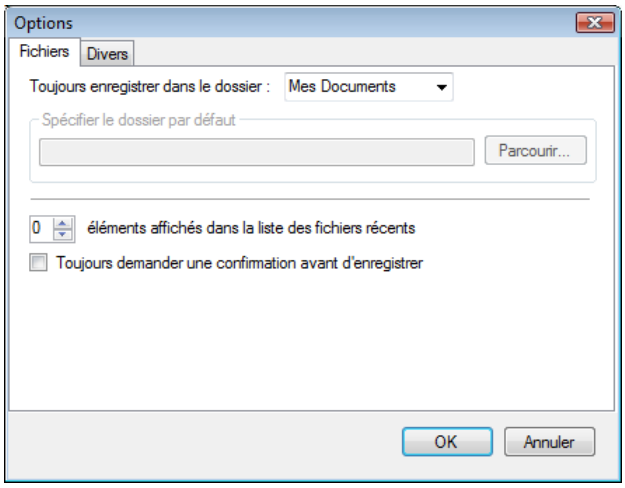
5. Testez maintenant le fonctionnement de la sauvegarde et du chargement des options de la boîte de dialogue.
    - Ajoutez l'appel des deux procédures **LoadOptions** et **SaveOptions** que vous venez d'écrire dans le gestionnaire d'évènement **OptionsToolStripMenuItem_Click** aux emplacements prévus à cet effet :

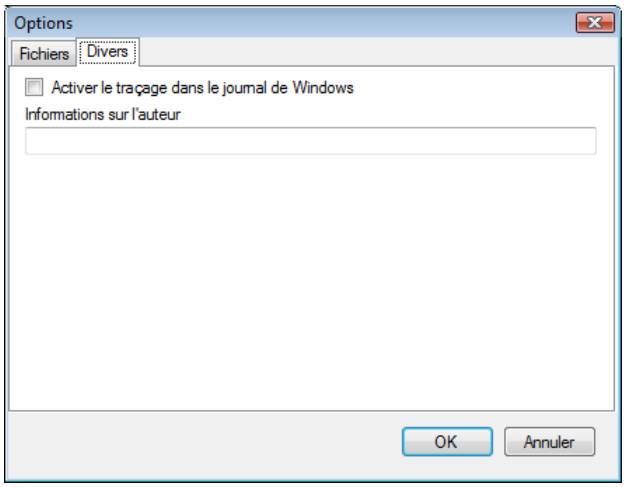
Code VB

```
Private Sub OptionsToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
```

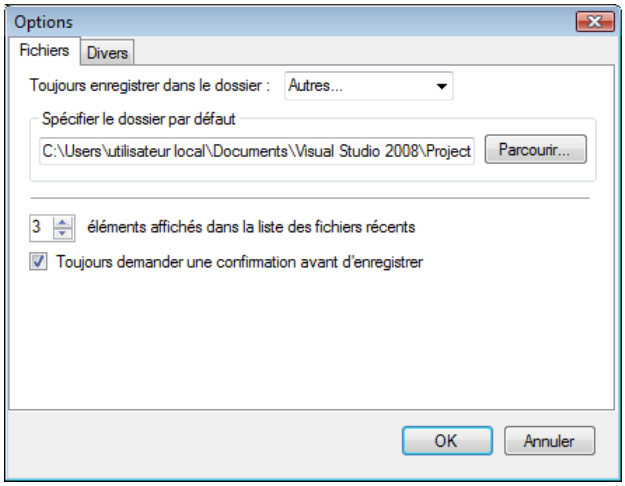
```
                Handles OptionsToolStripMenuItem.Click
        Dim formOptions As OptionsDialog = New OptionsDialog
        'Chargement des options dans la boîte de dialogue
        LoadOptions(formOptions)
        If formOptions.ShowDialog() = Windows.Forms.DialogResult.OK Then
            'Enregistrer les options configurées dans la boîte par l'utilisateur
            SaveOptions(formOptions)
        End If
End Sub
```

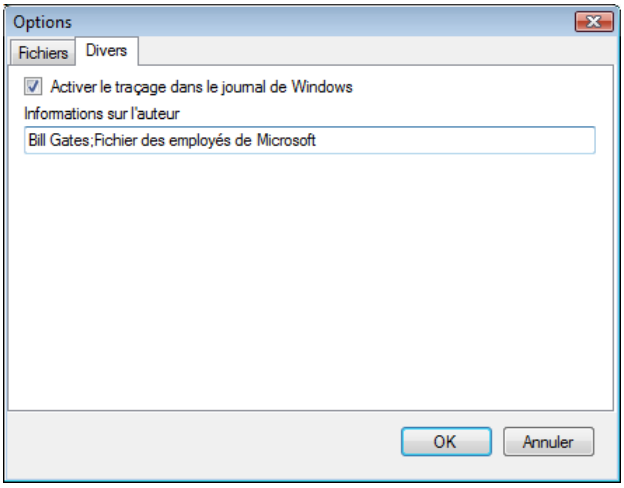
- Enregistrez tous vos changements.
- Exécutez l'application (**F5**).
- Cliquez le menu **Outils** > **Options**.
- Vérifiez que la boîte de dialogue est correctement initialisée. La liste déroulante du type de dossier d'enregistrement doit notamment être initialisée à **Mes Documents** :



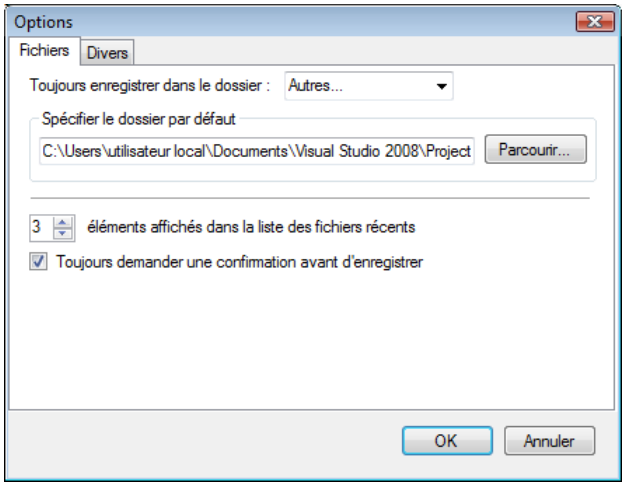


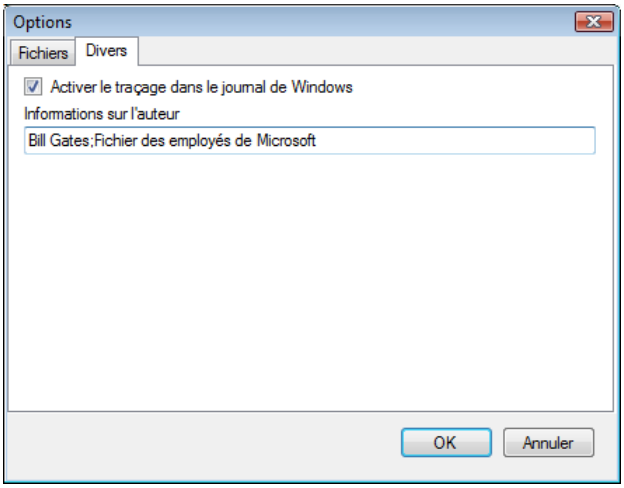
- Renseignez les différentes options de la boîte de dialogue à votre guise. Par exemple :





- Validez la boîte en cliquant **OK**.
- Réaffichez la boîte de dialogue en cliquant **Outils** > **Options** pour vérifier que vous retrouvez correctement toutes les options configurées précédemment.



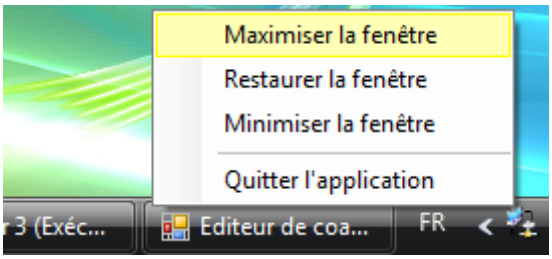


Bien ! Votre boîte d'options est maintenant opérationnelle…

POUR ALLER PLUS LOIN…

TRAITEMENT DU MENU CONTEXTUEL DE L'ICÔNE DE NOTIFICATION

Et si vous traitiez le code correspondant au clic sur les différentes options du menu contextuel de l'icône de notification de l'application que vous avez construit à l'atelier 2 précédent ?



Il se trouve que le Framework fournit l'énumération **FormWindowState** dans l'espace de noms **System.Windows.Forms**, pour vous aider à spécifier la façon dont une fenêtre Windows Form doit être affichée :

Nom de membre	Description
Maximized	Fenêtre agrandie.
Minimized	Fenêtre réduite.
Normal	Fenêtre ajustée par défaut.

A vous de jouer !

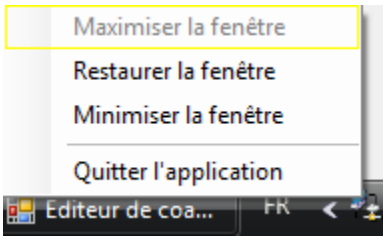
Retrouvez la définition de l'énumération **FormWindowState** ici :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.formwindowstate(VS.80).aspx

Vous devez obtenir le code suivant dans la classe **Main** :

```
92  #Region "Traitement des options du menu contextuel de l'icône de notification"
93      Private Sub MaximiserLaFenêtreToolStripMenuItem_Click(ByVal sender As System.
94          Me.WindowState = FormWindowState.Maximized
95      End Sub
96
97      Private Sub RestaurerLaFenêtreToolStripMenuItem_Click(ByVal sender As System.
98          Me.WindowState = FormWindowState.Normal
99      End Sub
100
101     Private Sub MinimiserLaFenêtreToolStripMenuItem_Click(ByVal sender As System.
102         Me.WindowState = FormWindowState.Minimized
103     End Sub
104 #End Region
```

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

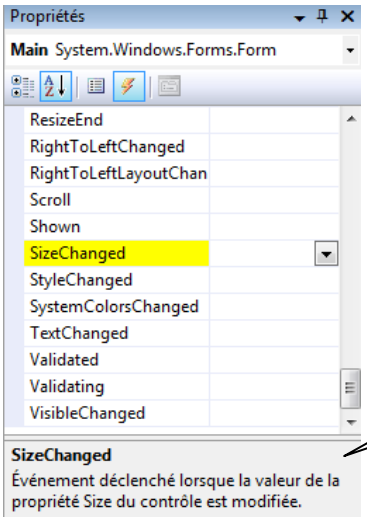


Dans ce cas, il faut exploiter la propriété **Enabled** des options du menu que nous avons déjà rencontrée dans cet atelier pour les contrôles de la boîte *Options*. Indiquez la valeur **True** ou **False** pour respectivement activer ou griser les options de menu.

Reste à déterminer où brancher le code correspondant ?

Le besoin est le suivant : le menu **Maximiser la fenêtre** doit être grisé **quand la fenêtre est en état maximisé**. L'objet sur lequel porte l'évènement déclencheur est donc le formulaire **Main**, et il faudrait chercher sur cette classe un évènement qui caractérise le changement d'état de la fenêtre.

- Affichez le formulaire **Main** en mode **Design**.
- Affichez la fenêtre de **Propriétés** de l'objet **Main** en cliquant **F4**.
- Basculez sur la liste des évènements en cliquant sur dans la barre d'outils de la fenêtre de **Propriétés** :



L'évènement qui nous intéresse ici est **SizeChanged** qui caractérise un changement de taille de la fenêtre. Notez que le nom de l'évènement inclut le verbe **Change** qui détermine l'action correspondante, et que celui-ci est ici au prétérit (au passé) caractérisé par la terminaison **ed** en anglais. Cela n'est pas tout à fait insignifiant puisque la forme passée permet d'indiquer que l'évènement sera donc déclenché après que l'action aura eu lieu.

Au contraire, certains noms d'évènement utilisent le verbe au présent progressif, avec la terminaison **ing**, pour indiquer que l'évènement est déclenché au début de l'action. La distinction est importante puisque dans ce dernier cas, vous pouvez carrément interagir

avec l'action, voire l'annuler. Par exemple, **FormClosing** est déclenché en début de processus de fermeture de la fenêtre et il est possible d'annuler ce processus en forçant la fenêtre à rester ouverte.

- Faites un double clic sur l'évènement **SizeChanged** pour générer un gestionnaire d'évènement.
- Le code pour griser les options de menu en fonction de l'état de la fenêtre pourrait être le suivant :

Code VB

```
Private Sub Main_SizeChanged(ByVal sender As System.Object, _
                             ByVal e As System.EventArgs) _
                             Handles MyBase.SizeChanged
    Me.MaximiserLaFenêtreToolStripMenuItem.Enabled = _
                Not (Me.WindowState = FormWindowState.Maximized)
    Me.MinimiserLaFenêtreToolStripMenuItem.Enabled = _
                Not (Me.WindowState = FormWindowState.Minimized)
    Me.RestaurerLaFenêtreToolStripMenuItem.Enabled = _
                Not (Me.WindowState = FormWindowState.Normal)
End Sub
```

Not est l'opérateur de négation logique qui applique une négation sur l'expression qui suit. Il est défini pour un opérande de type booléen et retourne **True** *si, et seulement si*, l'opérande est **False**, et inversement. Tout simplement il inverse le résultat de l'expression booléenne entre parenthèses…

Or notre besoin est de vérifier l'état de la fenêtre et de griser le menu correspondant en conséquence. Par exemple, le code **(Me.WindowState = FormWindowState.Maximized)** retourne **True** dans le cas où l'état de la fenêtre est

maximisé. Or c'est précisément dans ce cas que l'option de menu **Maximiser la fenêtre** doit être grisée. Donc il suffit d'inverser la valeur de l'expression de vérification de l'état de la fenêtre pour en déduire la valeur de la propriété **Enabled** de l'option de menu.

Pour tout savoir sur l'opérateur **Not** :

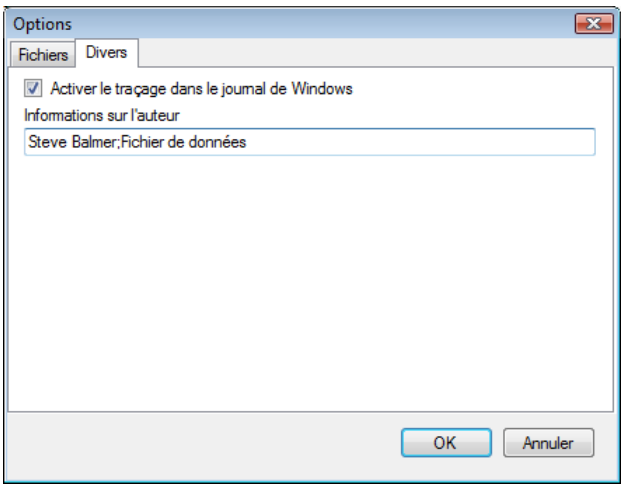
http://msdn.microsoft.com/fr-fr/library/2cwcswt4.aspx

- Il ne vous reste qu'à tester l'application pour vérifier que le menu contextuel se comporte comme souhaité !

ECRIRE DANS LE JOURNAL DE WINDOWS

Un dernier petit exercice pour la forme ☺…

Que fait-on si l'utilisateur active le traçage dans le journal de Windows ? On pourrait par exemple écrire un message dans le journal de Windows à chaque fois qu'une erreur de logique se produit dans l'application.



Dans cet exercice, nous allons journaliser la configuration des options de dossier de l'utilisateur dans la boîte des options de Windows.

1. Ajoutez une procédure appelée LogOptions dans la classe Main.
    - Créez une nouvelle procédure à la suite des précédentes dans la classe **Main** :

Code VB

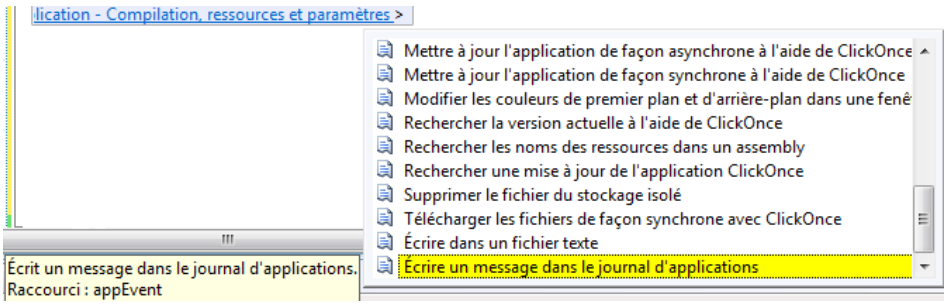
```
Private Sub LogOptions()

End Sub
```

- Branchez tout de suite l'appel de cette procédure lors de la sauvegarde des options de la boîte dans la procédure **SaveOptions**, à condition bien sûr que le traçage soit activé :

Code VB

```
Private Sub SaveOptions(ByVal form As OptionsDialog)
    …
    'Sauvegarde de l'activation du traçage dans le journal de Windows
    TraceEnabled = form.TraceCheckBox.Checked
    If form.TraceCheckBox.Checked Then
       LogOptions()
    End If
    …
End Sub
```

Pour apprendre à journaliser une information dans le journal de Windows, pensez une fois de plus aux extraits de code (snippets) fournis par VB. Faites un clic droit **Insérer un extrait…** > **Application – Compilation, ressources et paramètres** > **Ecrire un message dans le journal d'applications**.



Par contre, la ligne de code générée fait référence à l'objet **My.Application.Log** et à sa méthode **WriteEntry** qui écrit par défaut dans la fenêtre de sortie de Visual Studio plutôt que dans le journal de Windows par défaut.

```
Sub LogOptions()
    My.Application.Log.WriteEntry("Action complete.")
End Sub
```

En fait elle écrit dans ce qu'on appelle **les écouteurs de journalisation des évènements de l'application** dont l'emplacement est défini au niveau des fichiers de configuration de la machine.

Pour creuser la question sur la méthode **WriteEntry** de l'objet **My.Application.Log** :

http://msdn.microsoft.com/fr-fr/library/xbw77c0x(VS.80).aspx

ou encore ce lien :

http://msdn.microsoft.com/fr-fr/library/7fx0fexe(VS.80).aspx

- Voici le code que nous vous proposons en réponse à ce dernier petit exercice. Il n'utilise pas l'extrait de code précédent car nous voulons illustrer comment écrire dans le journal de Windows en configurant la source du message de manière spécifique pour bien identifier la provenance du message :

Code VB

```
Sub LogOptions()
    My.Application.Log.WriteEntry(logMessage.ToString())
    'Construction du message à enregistrer'
    Dim logMessage As System.Text.StringBuilder = _
```

```
                    New System.Text.StringBuilder
    logMessage.AppendFormat("Prefered Save Directory Path : {0}", _
                                saveDirectoryPath)
    logMessage.AppendFormat("Confirm before save :  {0}", _
                                confirmBeforeSave.ToString())
    'Création d'une source d'évènement spécifique à notre application
    If Not EventLog.SourceExists("Coach VB.NET") Then
       System.Diagnostics.EventLog.CreateEventSource("Coach VB.NET", _
                                                      "Application")
    End If
    'Création d'un journal dont la source est celle définie précédemment
    Dim myLog As System.Diagnostics.EventLog = _
                                 New System.Diagnostics.EventLog()
    myLog.Source = "Coach VB.NET"
    'Ecriture du message dans le journal
    myLog.WriteEntry(logMessage.ToString())
End Sub
```

Plusieurs petites choses sympas dans ce morceau de code :

- Tout d'abord parlons de l'utilisation de la classe **StringBuilder** pour concaténer les différents morceaux de texte que l'on veut ajouter au message du journal.

  Pourquoi définir un objet de type **StringBuilder** ?

  Il faut savoir qu'une fois que vous assignez une valeur à une variable de type **String**, cette chaîne est *immuable*, ce qui signifie que vous ne pouvez pas modifier sa longueur ou son contenu. En réalité, VB vous autorise à modifier la valeur d'une variable de type **String** autant de fois que vous le voulez mais

dans les coulisses, il crée à chaque modification une nouvelle chaîne en mémoire et abandonne la précédente. La variable **String** pointe ensuite vers la nouvelle chaîne.

Du coup, comme c'est le cas ici, lorsque vous devez effectuer plusieurs changements successifs sur la valeur d'une chaîne, ça n'est pas très performant ☹.

Avec la classe **StringBuilder** de l'espace de noms **System.Text**, le Framework .NET vous propose une sorte de type String *mutable*. La méthode **AppendFormat** permet de concaténer plusieurs morceaux successivement à la chaîne en gérant son format.

Pour en savoir plus sur la classe **StringBuilder** :

http://msdn.microsoft.com/fr-fr/library/system.text.stringbuilder.aspx

- Ensuite, le code utilise un objet de type **EventLog** pour créer une source nommée **Coach VB.NET** pour qu'on puisse repérer facilement les messages du journal en provenance de notre application. Une fois que cette source est enregistrée auprès du journal **Application** de Windows, il n'est plus nécessaire de la redéfinir d'où le test d'existence de la source avant l'ordre de création de celle-ci.
- L'écriture proprement dite dans le journal se fait au travers de la méthode **WriteEntry** que nous avons vu dans l'extrait de code précédent.

Pour en savoir plus sur la classe **EventLog** de l'espace de noms **System.Diagnostics.EventLog** :

http://msdn.microsoft.com/fr-fr/library/system.diagnostics.eventlog(VS.80).aspx

- Notez que nous utilisons plusieurs fois dans ce code la méthode **ToString**. Il s'agit d'un autre mécanisme de conversion de type de données que celui que nous avons explicité plus haut dans cet atelier. En fait, tout objet quel qu'il soit comporte une méthode membre **ToString** pour convertir sa valeur en chaîne équivalente.

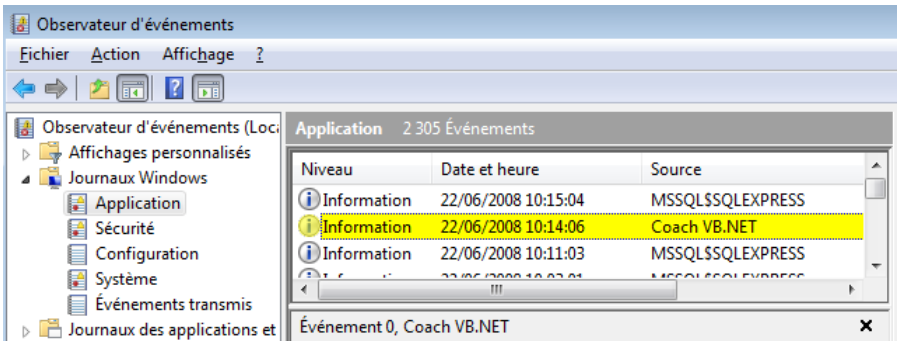
  Evidemment, le résultat de la conversion est directement lié au type de la donnée correspondante.

  Par exemple, pour convertir la valeur booléenne de la variable **confirmBeforeSave**, on applique la méthode **ToString** du type **Boolean**. Le résultat est la châine **« True »** si la valeur est **True** et **« False »** sinon.

Pour en savoir plus sur le comportement de la méthode **ToString** :

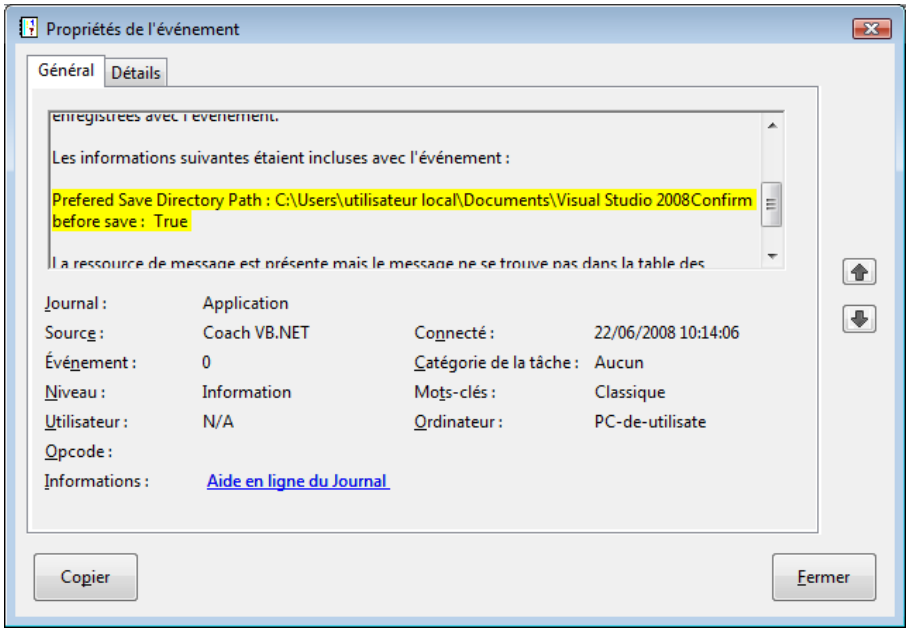
http://msdn.microsoft.com/fr-fr/library/system.object.tostring.aspx

- Si vous testez l'application en configurant le traçage dans la boîte des options, vous obtenez le message suivant dans le journal de Windows :



Pour afficher le journal de Windows, lancez **l'Observateur d'évènements** depuis le **Panneau de configuration** de la machine.

- Double cliquez sur le message pour lire son contenu :



Manipuler des données de fichier

Sommaire

1 INTRODUCTION 3

1.1 CONTEXTE FONCTIONNEL 3

2 TRAVAILLER AVEC LES CONTRÔLES DE DONNÉES 6

2.1 PRÉPARER LE POSITIONNEMENT DES CONTRÔLES À L'AIDE D'UNE STRUCTURE DE TABLEAU 6
2.2 DESSINER LES CONTRÔLES DE DONNÉES 14

3 ALIMENTER LES CONTRÔLES DE DONNÉES AVEC UN FICHIER CSV 22

3.1 CRÉER UNE TABLE MÉMOIRE (DATATABLE) 22
3.2 COMPRENDRE LA LIAISON DE DONNÉES (DATABINDING) 30
3.3 MANIPULER LE FICHIER CSV 35
3.3.1 Lire le fichier 35
3.3.2 Ecrire dans le fichier 57

4 POUR ALLER PLUS LOIN… 72

4.1 L'OBJET MY.COMPUTER.FILESYSTEM 72
4.2 EVÈNEMENTS DE FICHIER 73

1 Introduction

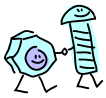
1.1 Contexte fonctionnel

Rappel du contexte fonctionnel du tutorial du coach VB

L'objectif du tutorial du Coach VB est d'accompagner les développeurs à la découverte et la prise en main du langage Visual Basic (VB) pour la construction d'applications avec une approche orientée objet.

Pour rappel, vous pouvez repérer facilement deux caractéristiques importantes du langage à l'aide des logos suivants en marge :

Ce logo marque une fonctionnalité de VB ou de Visual Studio qui permet de développer vite (et juste ☺).

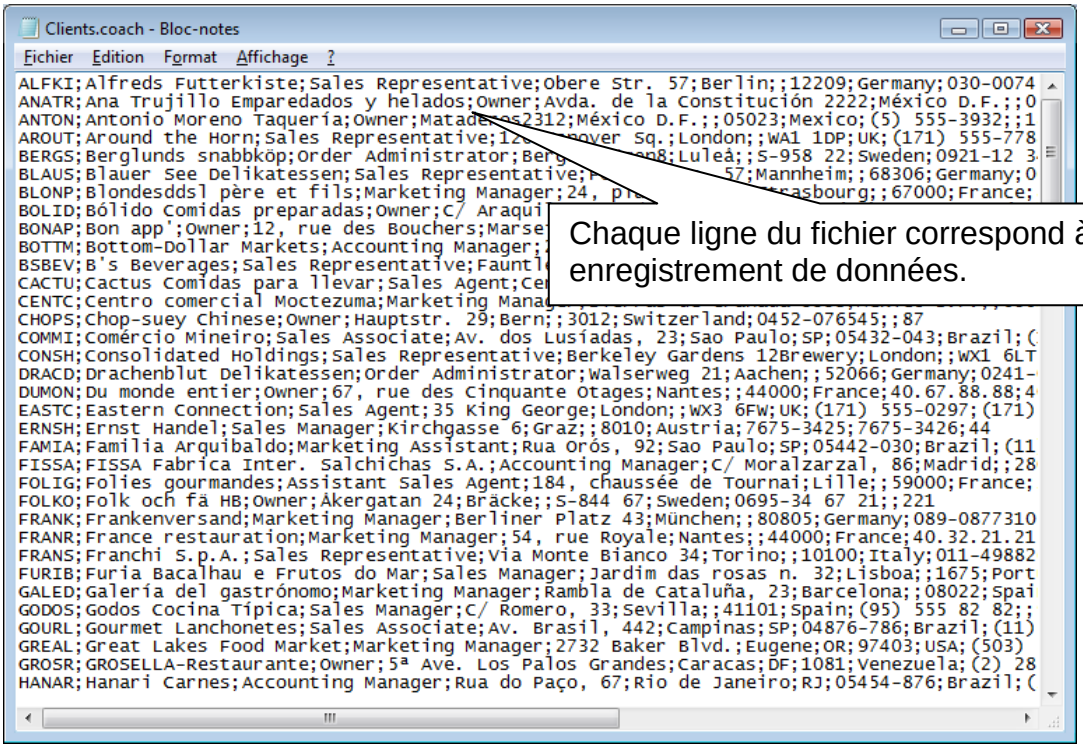


Ce logo met en évidence une caractéristique de la programmation orientée objet.

Contexte fonctionnel du quatrième atelier

Dans ce quatrième atelier, nous allons commencer à manipuler des données. Pour l'instant celles-ci sont chargées et sauvegardées sous la forme d'un fichier délimité CSV d'extension *.coach. Nous verrons dans la suite de ce tutorial comment manipuler ces mêmes données à partir d'une base de données.

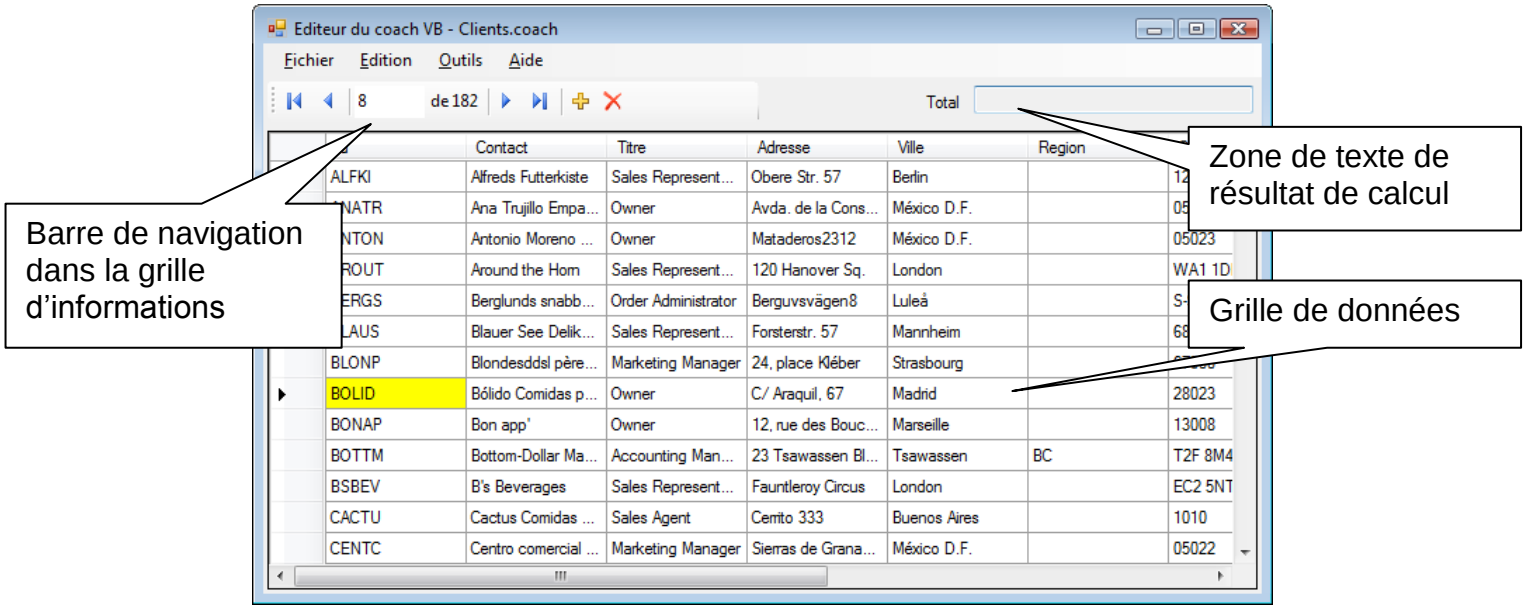
Exemple d'un fichier **Clients.coach** :



Il s'agit d'une liste de contacts client comprenant les informations suivantes :

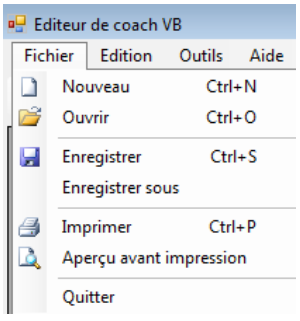
Nom	*Description*	*Type*
Id	Code d'indentification du client	string
Contact	Nom du client	string
Titre	Fonction du client	string
Adresse	Adresse du client	string
Ville	Ville de résidence du client	string
Region	Région de résidence du client	string
CodePostal	Code postal du bureau postal distributeur	string
Pays	Pays de résidence du client	string
Telephone	Numéro de téléphone du client	string
Telecopie	Numéro de la télécopie du client	string
CA	Chiffre d'affaire arrondi, en millier d'euros, que vous réalisez avec ce client	int

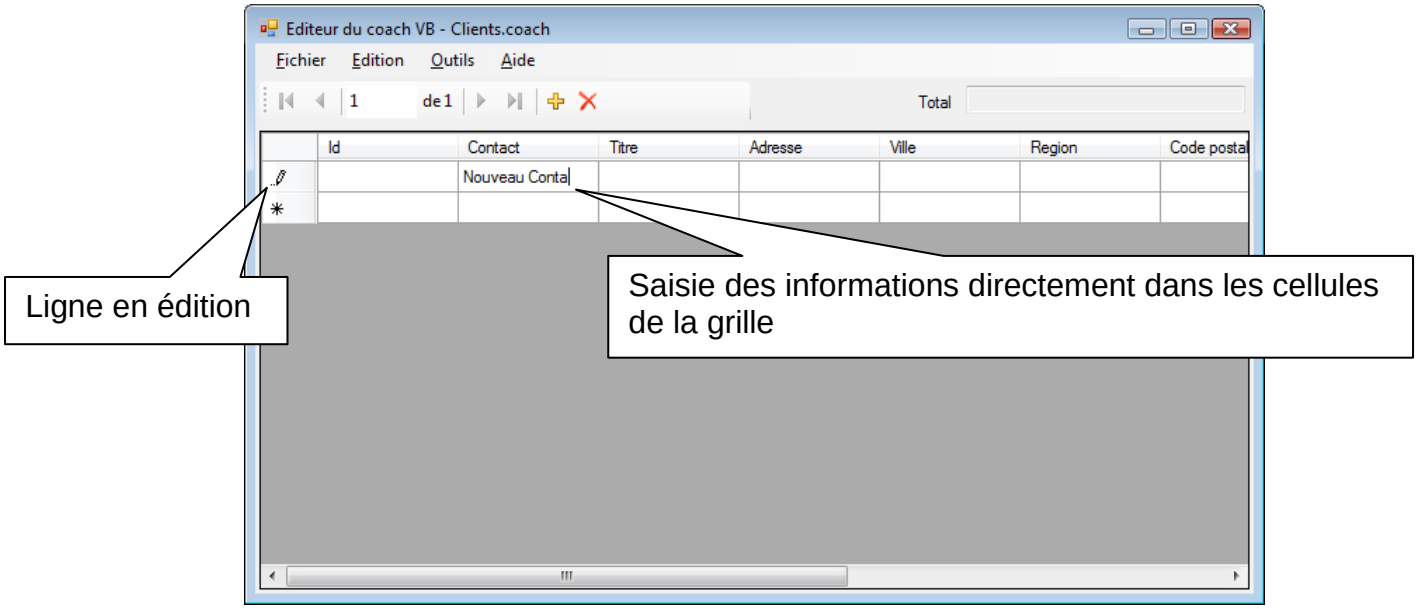
L'objectif est d'éditer les données dans notre application Editeur sous la forme d'une grille de données :



L'utilisateur doit pouvoir :

- Ouvrir un fichier existant via le menu **Fichier** > **Ouvrir** :
  - Ajouter de nouveaux contacts,
  - Supprimer et modifier des contacts existants,
  - Enregistrer ses modifications dans le fichier initial via le menu **Fichier** > **Enregistrer**
  - ou enregistrer sous un nouveau nom de fichier via le menu **Fichier** > **Enregistrer sous**,
- Ou créer un nouveau fichier via le menu **Fichier** > **Nouveau** :





Contexte technique
L'objectif de cet atelier est d'introduire les principes de gestion de données en commençant avec une approche « à l'ancienne », c'est-à-dire sans exploiter la puissance de la programmation orientée objet, et sans base de données.

A quoi bon programmer comme ça si ce n'est pas ce qu'il faut faire ?
Ne vous méprenez pas, cela va être très utile. C'est un peu comme si on allait apprendre à nager la brasse avant de se lancer à nager en brasse coulée. Nous allons travailler sur deux axes :

- Le premier concerne l'affichage des données au moyen de contrôles Windows Form adaptés. Nous verrons que l'affichage d'une source de données dans un contrôle d'affichage se fait au moyen du mécanisme de *databinding (liaison de données)*.
- Le second concerne la gestion de fichiers texte au format CSV. Nous allons apprendre à ouvrir, lire puis à écrire et enregistrer un fichier.

A la fin de cet atelier, vous saurez comment :

- Lire et écrire dans un fichier texte,
- Utiliser le databinding,
- Manipuler un tableau de données en mémoire,
- Positionner des contrôles d'affichage à l'aide de conteneur,
- Utiliser les principaux contrôles de données.

La solution de cet atelier est disponible dans le répertoire **..\Atelier 4\Solution**. Les fichiers utiles, auxquels font référence les exercices sont disponibles dans le répertoire **..Atelier 4\Fichiers utiles**.

2 Travailler avec les contrôles de données

Dans cet exercice, vous allez apprendre à :

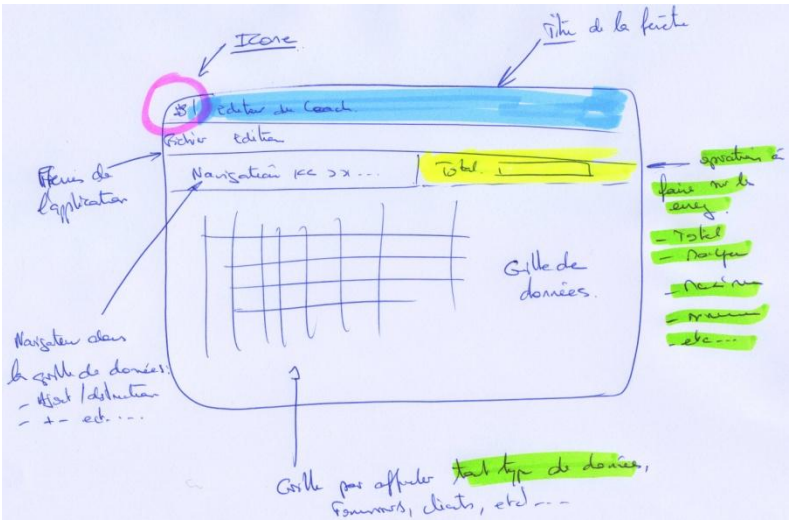
- Utiliser le contrôle conteneur TableLayoutPanel,
- Dessiner une grille de données et une barre de navigation,
- Positionner des contrôles dans un conteneur.

Objectif
L'objectif de ce premier exercice est de se familiariser avec quelques contrôles standards d'affichage et de manipulation de données.

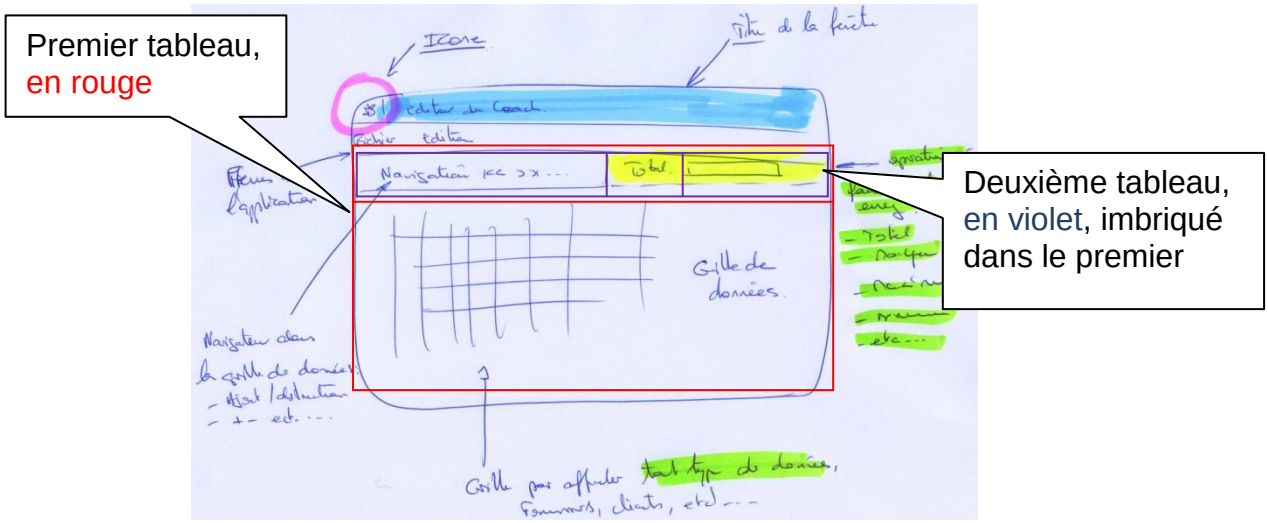
2.1 Préparer le positionnement des contrôles à l'aide d'une structure de tableau

Avant de dessiner les contrôles de données, il s'agit de les positionner. Pour cela nous allons utiliser un contrôle conteneur qui permet de structurer des contrôles sous forme de tableau.

Contexte fonctionnel
Vous vous rappelez le cahier des charges ? C'est ce petit bout de papier qui est ressorti d'une réunion d'analyse :



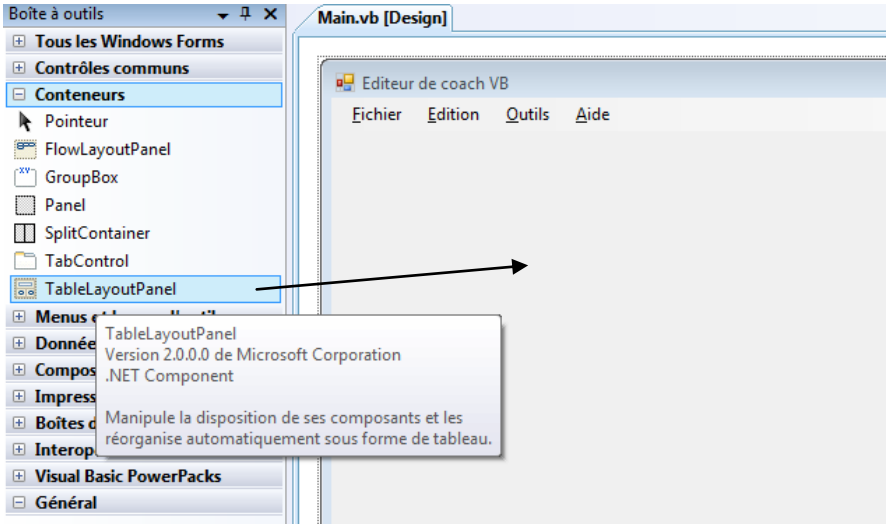
Oui, je sais…cela demande pas mal d'imagination ☹.
N'empêche qu'en y regardant bien, on peut voir que les différentes parties du formulaire sont organisées sous forme de tableaux imbriqués :



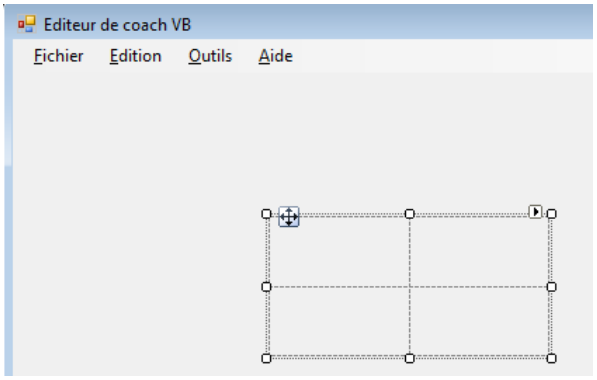
Alors pourquoi ne pas carrément dessiner des tableaux (dont les bordures seraient transparentes) pour nous aider à positionner les contrôles de manière alignée ?
Le premier tableau contiendrait une colonne et deux lignes. Le second, imbriqué dans la première ligne du précédent, comprendrait une seule ligne et trois colonnes.

Déroulement de l'exercice :

1. Ouvrez le projet précédent réalisé lors de l'atelier 3 :
   - Lancez Visual Studio à partir du menu **Démarrer** > **Tous les programmes** > **Microsoft Visual Basic 2008 Express Edition**.
   - Menu **Fichier** > **Ouvrir un projet**.
   - Retrouvez le fichier **Atelier 3.sln** que vous avez créé lors de l'atelier 3 ou, si vous n'avez pas fait l'atelier précédent, récupérez le projet de démarrage fourni avec le code de cet atelier dans le répertoire : **..\Atelier 4\Démarrage\Atelier 4.sln**.
2. Ajoutez un premier tableau (le rouge) au formulaire :
   - Ouvrez le fichier **Main.vb** en mode **Design**.
   - Faites un glisser déplacer de la **Boîte à outils** > rubrique **Conteneurs** > du contrôle **TableLayoutPanel** sur la surface centrale du formulaire.



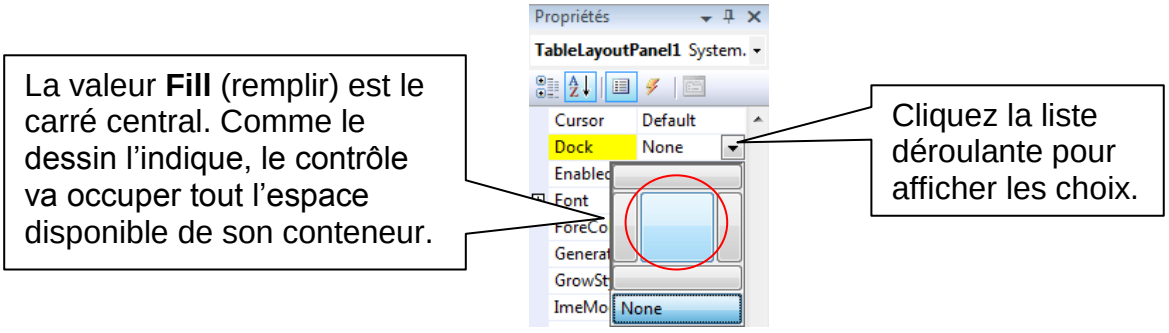
- Vous obtenez un tableau comprenant par défaut deux lignes et deux colonnes :



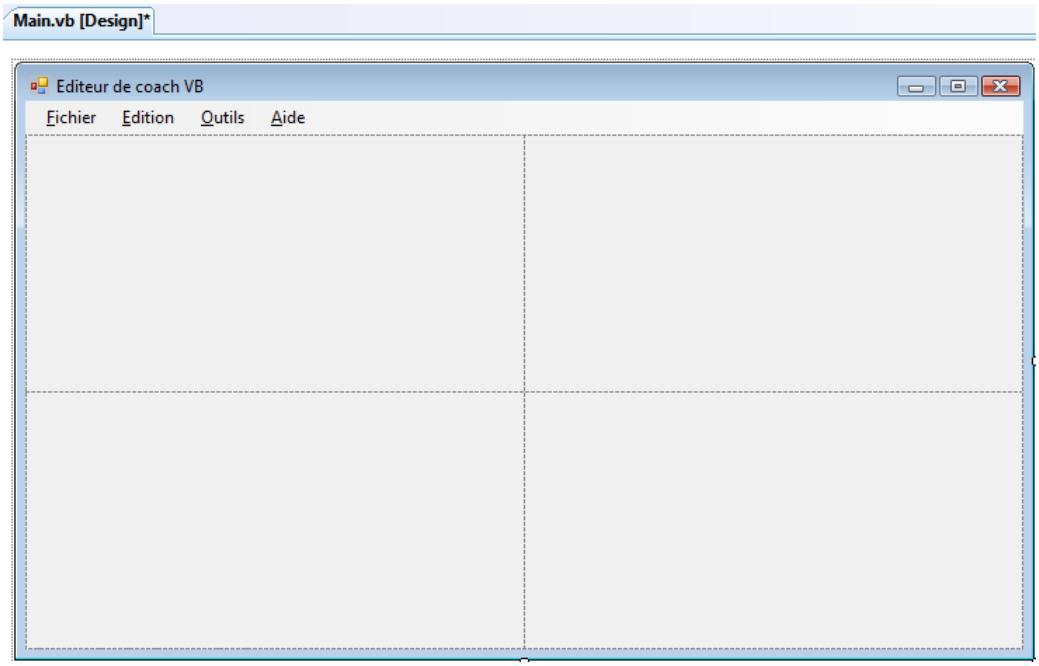
Vous pouvez déplacer le contrôle en cliquant l'icône en haut à gauche du contrôle. L'idéal serait de positionner le contrôle de façon à ce qu'il occupe toute la surface disponible du formulaire. Rappelez-vous, il s'agit du mécanisme *d'ancrage* des contrôles dont nous avons déjà parlé dans l'atelier précédent. Il se configure notamment à l'aide de la propriété **Dock** des contrôles.

- Affichez la fenêtre de **Propriétés** du contrôle (**F4**).
- Configurez la propriété **Dock** à la valeur **Fill** pour remplir toute la surface disponible du formulaire.

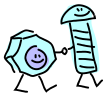
Visual Studio vous propose un éditeur visuel pour configurer cette propriété plus intuitivement :



Vous devez obtenir :



Le principal travail sur ce type de contrôles consiste à définir les lignes et les colonnes. Cela revient à configurer les propriétés **Columns** et **Rows** du contrôle **TableLayoutPanel** qui représentent respectivement une collection de colonnes et une collection de lignes.

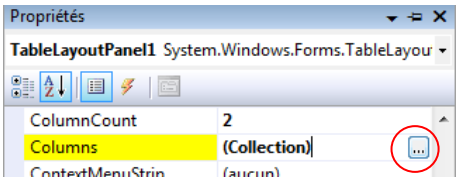


Qu'est ce qu'on entend par *collection* ?
Une collection est un objet qui sert à regrouper des objets apparentés. C'est en quelque sorte un tableau d'objets. Vous ne vous en êtes peut-être pas aperçu mais il y en a partout. Le Framework par exemple, utilise et fournit massivement des collections d'objet.

msdn

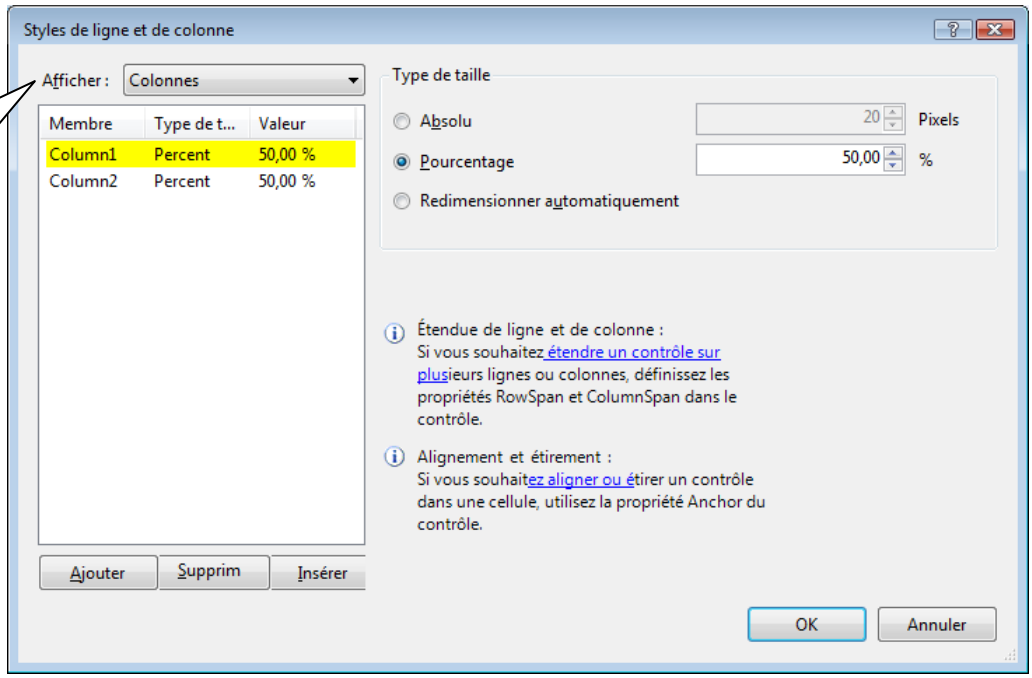
Pour tout savoir sur les collections en Visual Basic :
http://msdn.microsoft.com/fr-fr/library/a1y8b3b3(VS.80).aspx

3. Configurez le tableau pour qu'il ait deux lignes et une seule colonne :
   - Toujours dans la fenêtre de **Propriétés** du contrôle **TableLayoutPanel1**, sélectionnez la propriété **Columns** qui représente la collection de colonnes du tableau. Cliquez sur le bouton [...] qui s'affiche en face de la propriété :

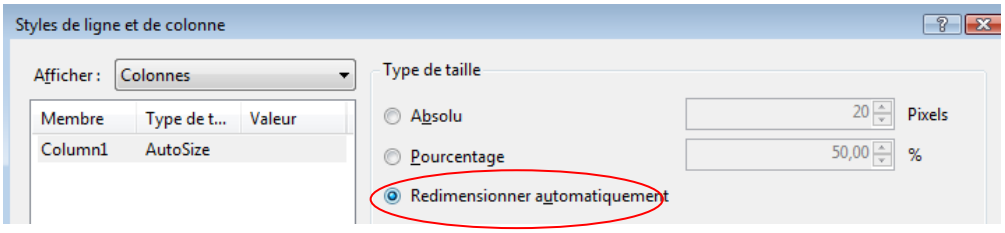


Là encore Visual Studio vous propose un éditeur pour vous assister. Il s'agit d'une boîte de dialogue commune aux propriétés **Columns** et **Rows** dans laquelle vous pouvez configurer à la fois les colonnes et les lignes du tableau. Utilisez la liste déroulante **Afficher** pour basculer des unes aux autres.

Affichage des colonnes ou des lignes



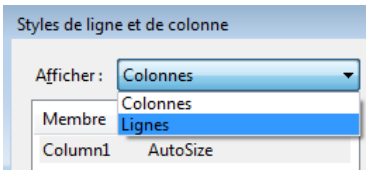
- Sélectionnez la colonne **Column2**, et cliquez le bouton **Supprimer** de l'éditeur de propriétés pour détruire la deuxième colonne qui nous est inutile.
- Sélectionnez la colonne **Column1**, et indiquez que sa dimension est automatiquement calculée en cliquant le radio-bouton **Redimensionner Automatiquement** :



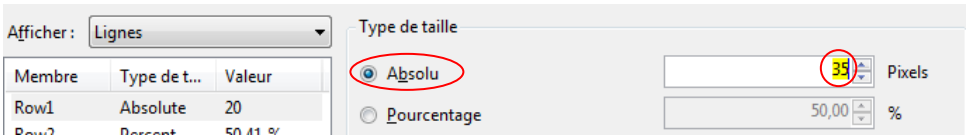
msdn

Pour voir comment fonctionne le redimensionnement automatique dans le contrôle **TableLayoutPanel** :
http://msdn.microsoft.com/fr-fr/library/ms171690.aspx

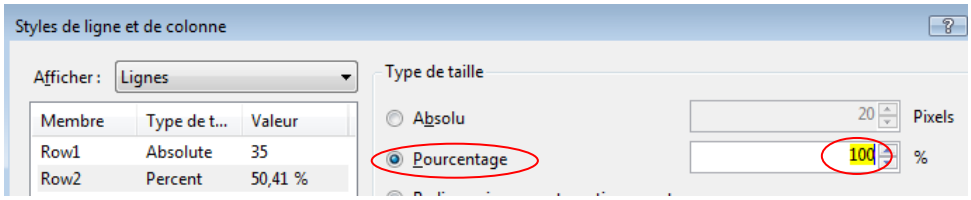
- Basculez maintenant sur la configuration des lignes en sélectionnant **Lignes** dans la liste déroulante **Afficher** :



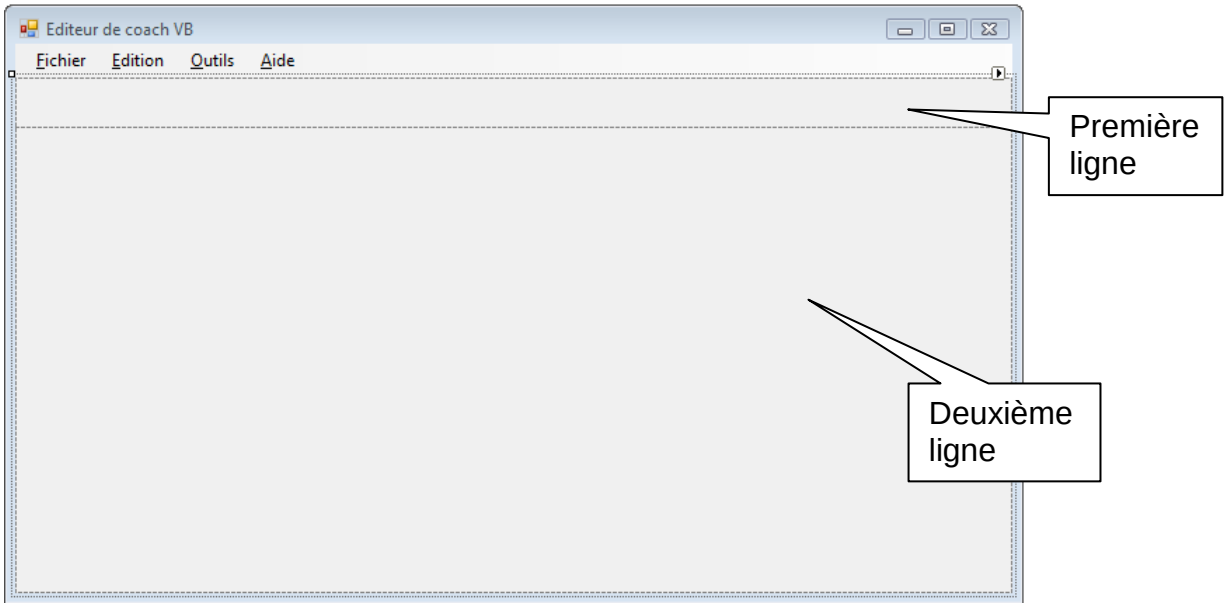
- Pour la première ligne, dont le nom est **Row1**, indiquez une taille de type **Absolu** et de **35 pixels** :



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

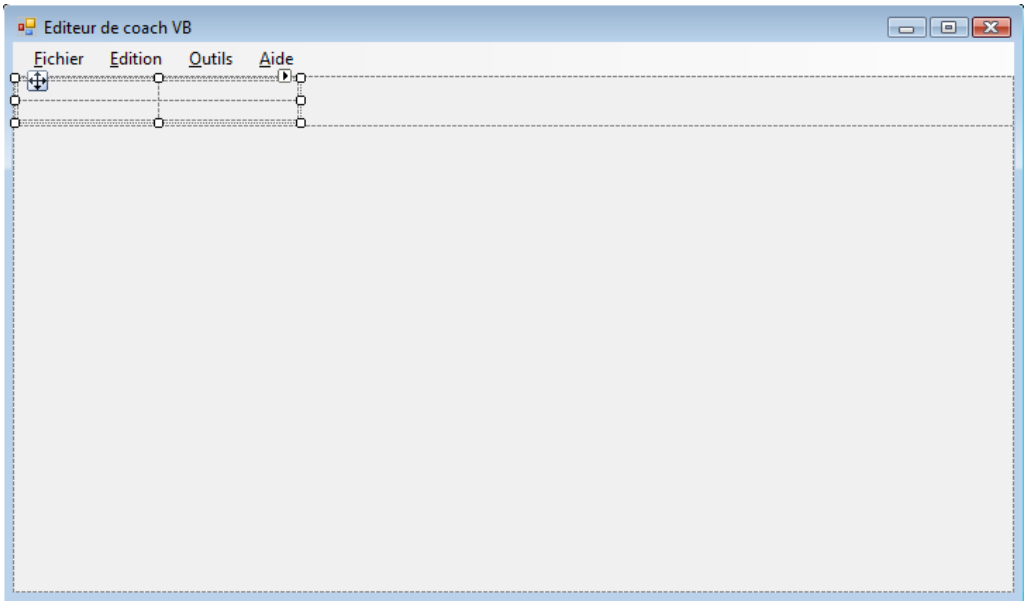


- **Cliquez** maintenant le bouton **Ok** pour fermer la boite de dialogue **Styles de ligne et de colonne**. Vous devez obtenir le tableau suivant :

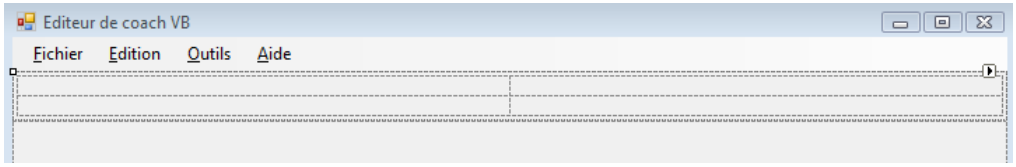


4. Ajoutez à l'intérieur de la première ligne du tableau un deuxième tableau (le violet):

- Faites un **glisser-déplacer** d'un deuxième contrôle **TableLayoutPanel** à l'intérieur de la première ligne du tableau précédent.

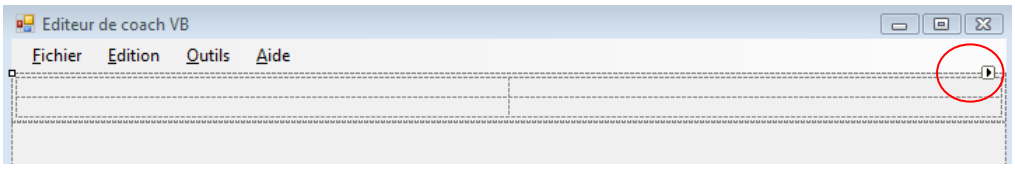


- Affichez la fenêtre de **Propriétés** du nouveau contrôle **tableLayoutPanel2** ;
- Configurez la propriété **Dock** du contrôle à **Fill** de façon à ce que le contrôle remplisse toute la surface disponible de son conteneur parent (c'est-à-dire de la ligne du premier tableau).

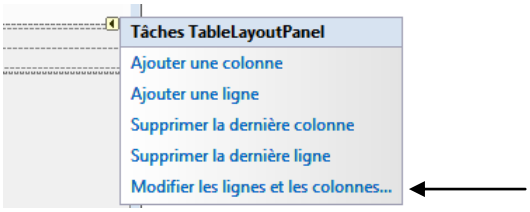


5. Configurez le tableau pour qu'il ait une ligne et trois colonnes :

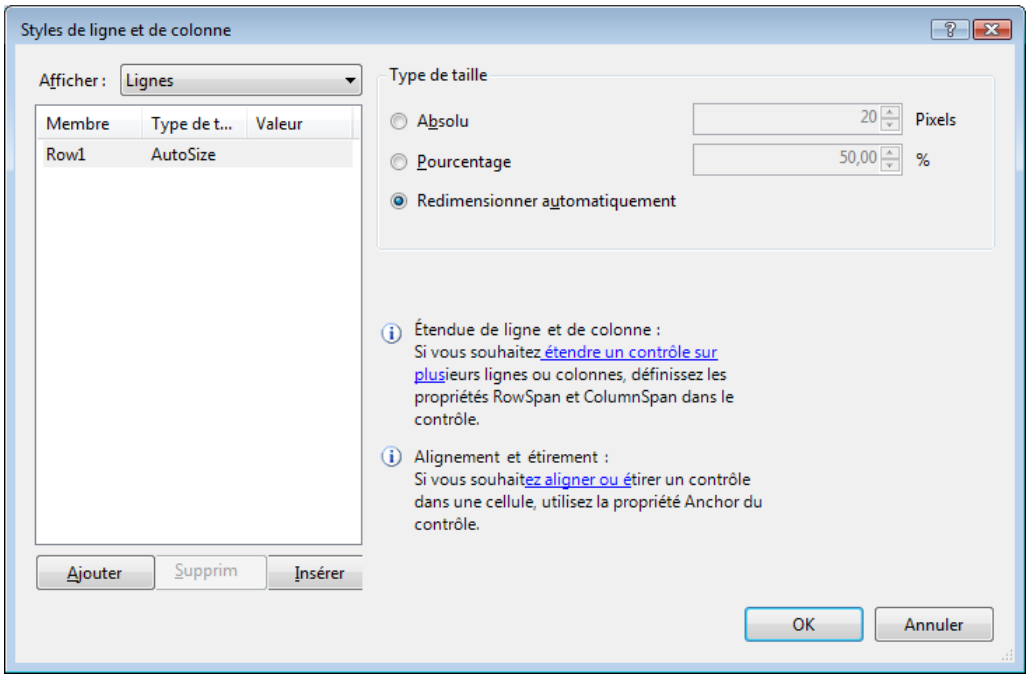
Il suffit, comme précédemment de configurer la propriété **Columns** (ou **Rows**) du contrôle pour obtenir les lignes et colonnes souhaitées.
Mais Visual Studio propose aussi un raccourci vers les principales actions possibles sur le contrôle en cliquant sa *balise active*. Il s'agit de l'icône représentant une petite flèche que l'on trouve en général en haut à droite du contrôle lorsqu'il est sélectionné.



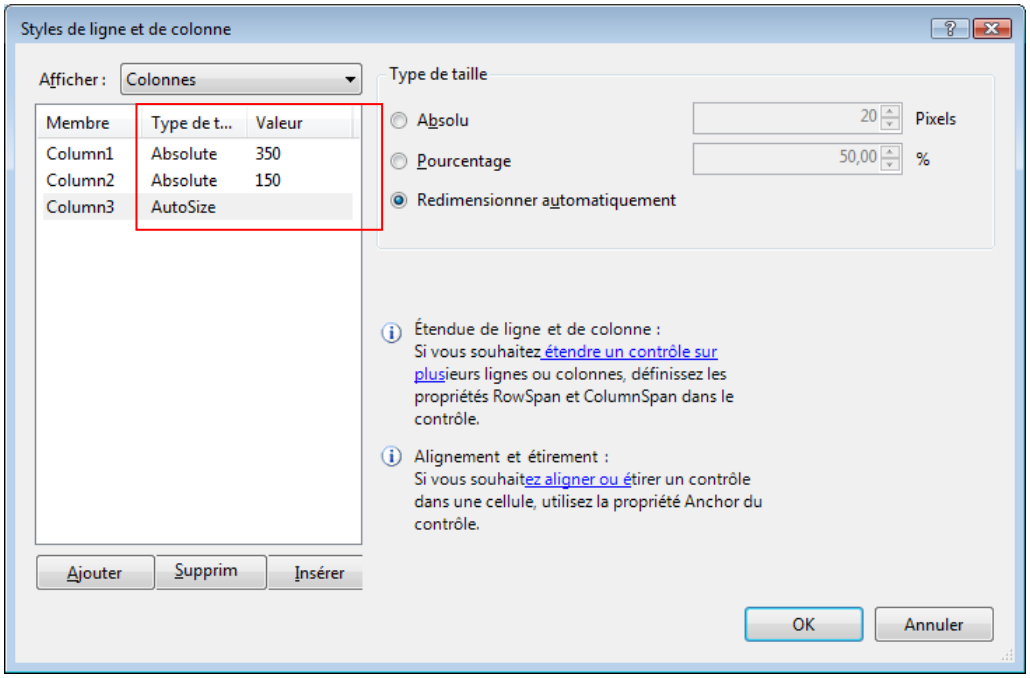
Par exemple , avec l'option **Modifier les lignes et les colonnes…**, vous retrouvez exactement la même boîte de dialogue de configuration que précédemment :



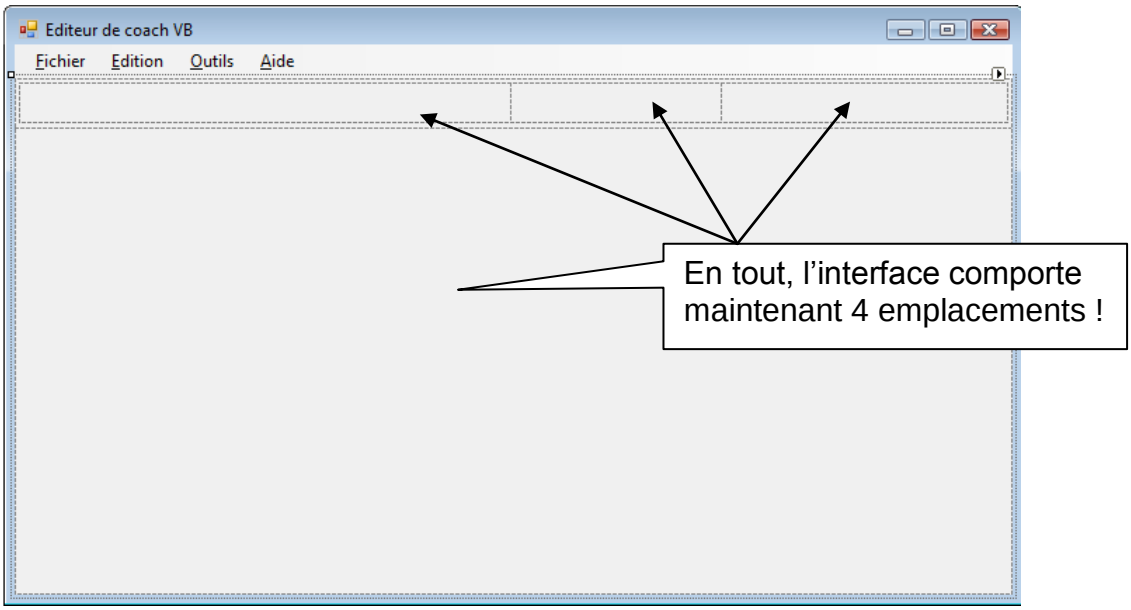
- Sélectionnez **Modifier les lignes et les colonnes…** pour afficher la boite de dialogue **Styles de ligne et de colonne**.
- Dans la liste déroulante **Afficher**, sélectionnez **Lignes**.
- Supprimez la ligne **Row2**.
- Indiquez un redimensionnement automatique pour la ligne **Row1** :



- Dans la liste déroulante **Afficher**, sélectionnez **Colonnes**.
- Ajoutez une nouvelle colonne an 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** pour fermer la boîte de dialogue **Styles de ligne et de colonne**. Vous obtenez la structure suivante :



Pour tout savoir sur le contrôle **TableLayoutPanel** :
http://msdn.microsoft.com/fr-fr/library/3a1tbfwd.aspx
Pour apprendre à disposer des contrôles dans les Windows Forms :
http://msdn.microsoft.com/fr-fr/library/ty26a068.aspx

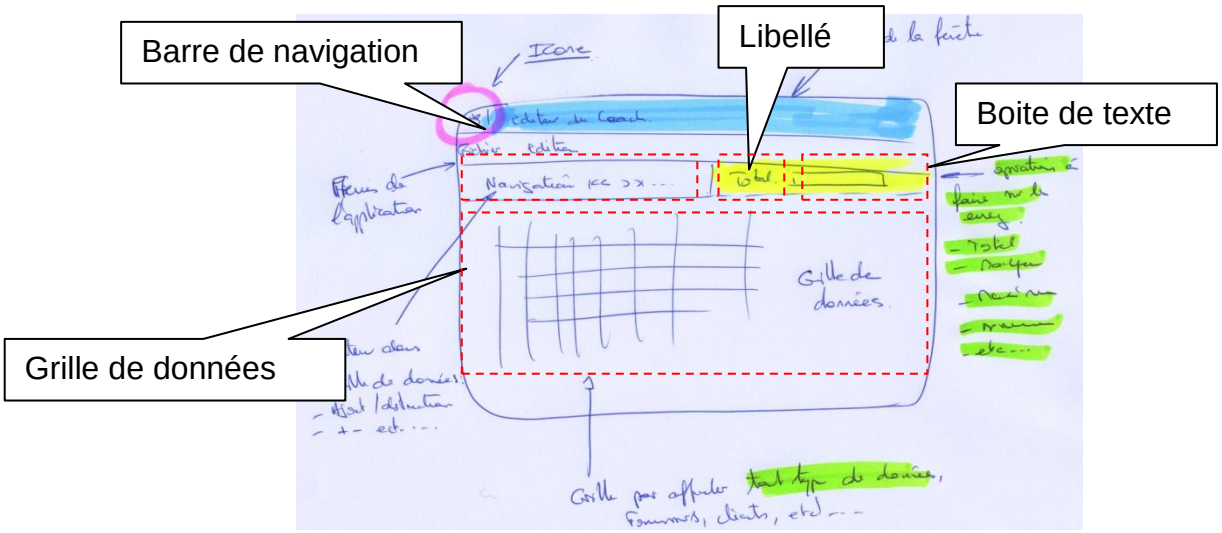
2.2 Dessiner les contrôles de données

Nous allons positionner dans les quatre cellules de tableau, quatre contrôles d'affichage que nous alimenterons avec les données d'un fichier CSV.

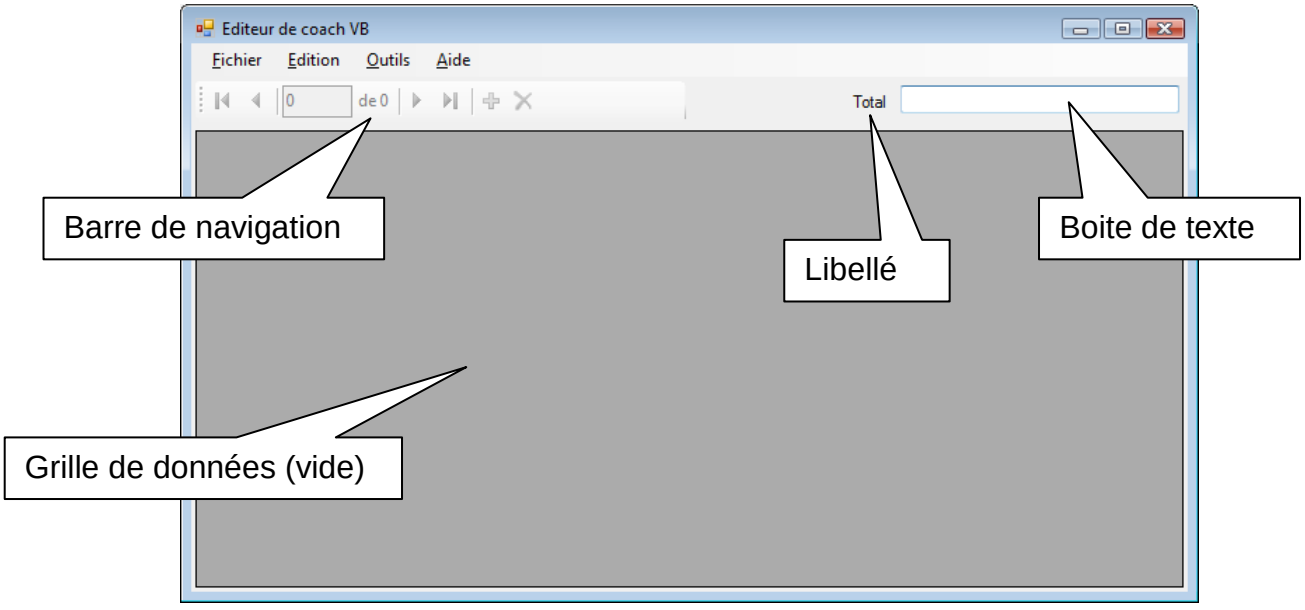
Contexte fonctionnel

Toujours en s'inspirant du document d'analyse (avec beaucoup d'imagination), il semble que l'interface utilisateur soit composée des contrôles suivants :

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



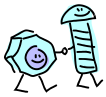
On devrait obtenir ceci :



Déroulement de l'exercice :

1. Ajoutez une grille d'affichage de données sur la deuxième ligne du tableau principal :

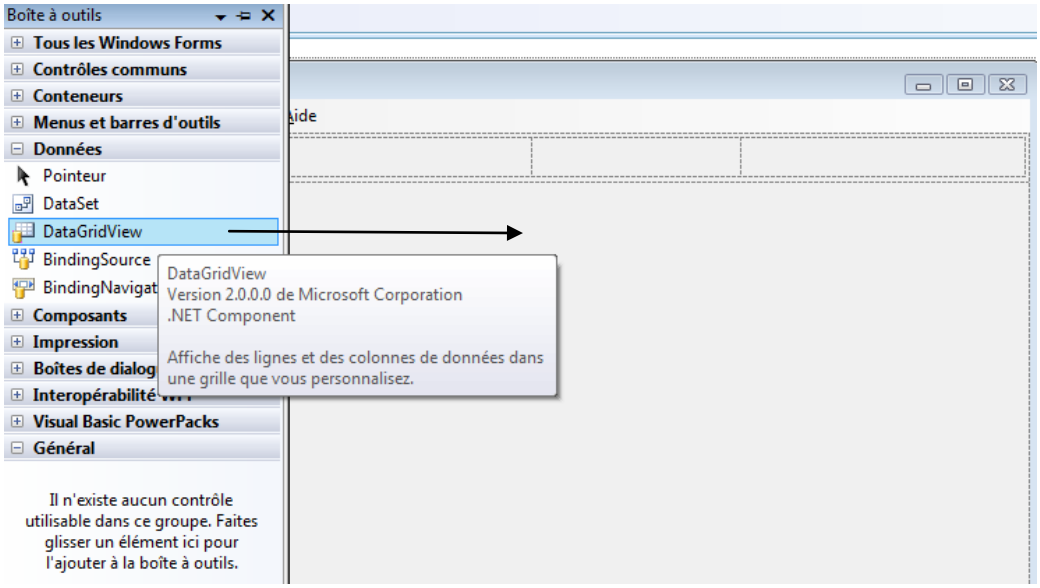
Le contrôle de grille de données standard des Windows Forms est le contrôle **DataGridView**. Il offre un moyen puissant et flexible pour afficher des données sous forme d'un tableau.



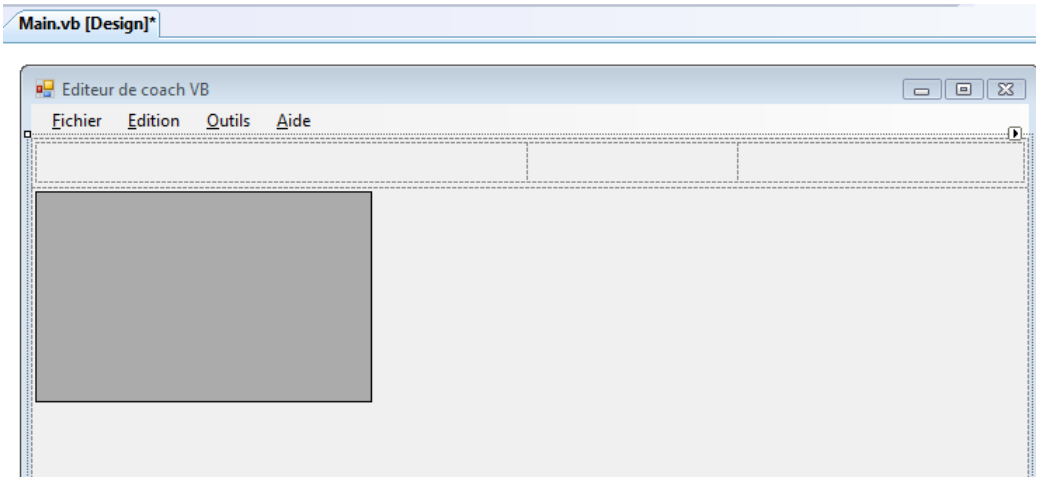
Nous verrons qu'avec les principes de programmation orienté objet tel que *l'héritage*, il est toujours possible d'étendre le comportement de base d'un objet pour le personnaliser en fonction de vos besoins.
Du coup, ne vous arrêtez pas au comportement standard d'un contrôle **DataGridView** qui ne remplit peut-être pas toutes les fonctionnalités dont vous avez besoin. Vous pouvez étendre par exemple son comportement en développant vos propres algorithmes de tri ou en confectionnant vos propres types de cellule.

Pour tout ce qui concerne l'utilisation du contrôle **DataGridView** :
http://msdn.microsoft.com/fr-fr/library/e0ywh3cz.aspx

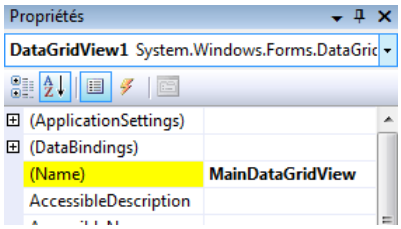
- Faites un glisser déplacer de la **Boîte à outils** > rubrique **Données** > du contrôle **DataGridView** sur la deuxième ligne du contrôle **TableLayoutPanel1** :



Vous obtenez :



- Affichez la fenêtre de **Propriétés** du contrôle.
- Configurez la propriété **(Name)** avec la valeur : **MainDataGridView**.



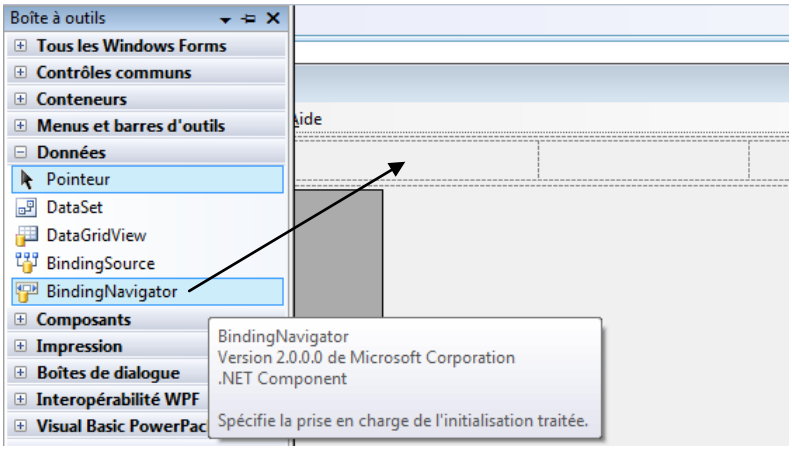
2. Ajoutez une barre de navigation de données dans la première cellule du second tableau :

Qu'est ce qu'on entend par barre de navigation ?
L'idée est d'assister l'utilisateur dans la manipulation du jeu de données qui est affiché dans l'Editeur. Comme son nom l'indique, une barre de navigation va lui permettre de *naviguer* d'un enregistrement à l'autre c'est-à-dire de faire avancer ou reculer le curseur dans la grille de données.

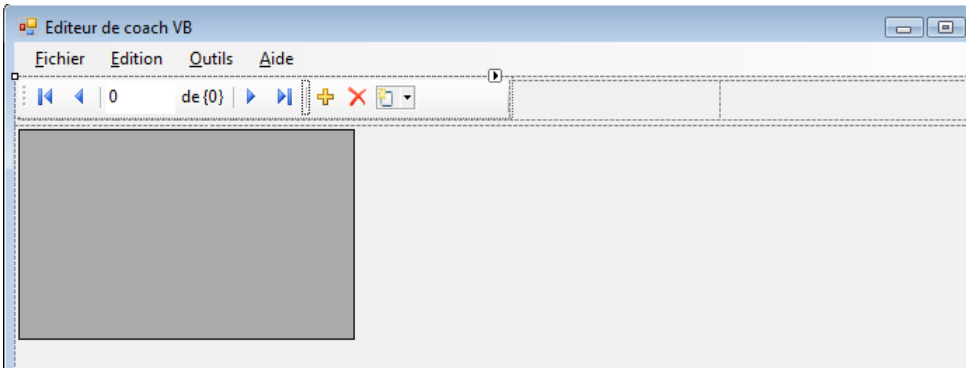
Les Windows Forms proposent un contrôle de navigation appelé **BindingNavigator**. Il permet non seulement de naviguer dans les données mais aussi d'interagir avec celles-ci. La barre de navigation peut par exemple afficher un bouton de création d'une nouvelle ligne de données, un bouton suppression etc…

Pour tout ce qui concerne l'utilisation du contrôle **BindingNavigator** :
http://msdn.microsoft.com/fr-fr/library/system.windows.forms.bindingnavigator.aspx

- Faites un glisser déplacer de la **Boîte à outils** > rubrique **Données** > du contrôle **BindingNavigator** sur la première cellule du contrôle **TableLayoutPanel2** :



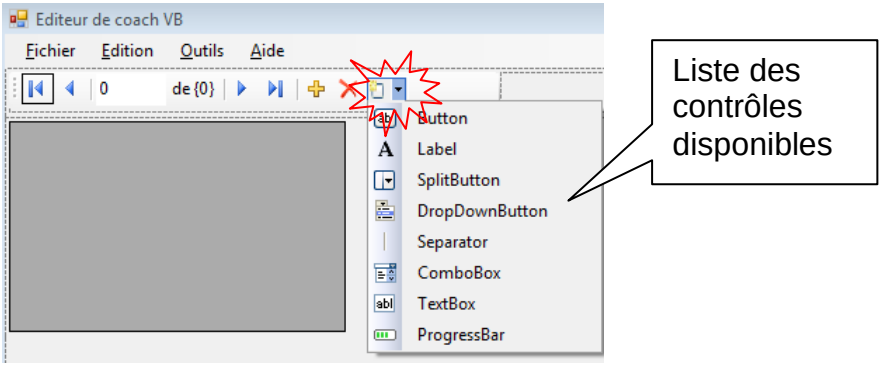
Vous obtenez :



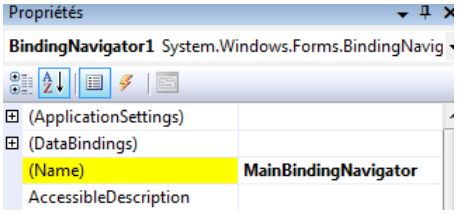
Notez les différents boutons que présente le contrôle :

- : pour la navigation classique,
- pour supprimer un enregistrement,
- pour créer un nouvel enregistrement.

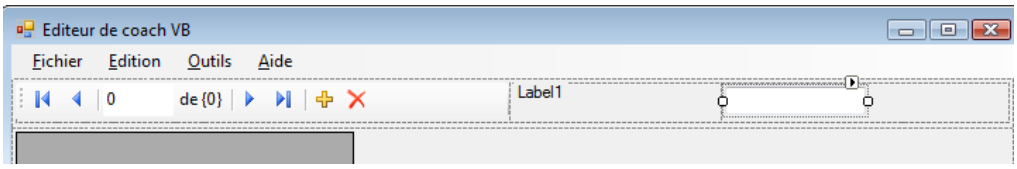
Vous pouvez agrémenter cette barre, exactement comme pour une barre d'outils, de nouveaux boutons ou autres contrôles personnalisés en cliquant (n'oubliez pas de sélectionner le contrôle BindingNavigator pour voir apparaître cet icône à droite) :



- Affichez la fenêtre de **Propriétés** du contrôle.
- Configurez la propriété **(Name)** avec la valeur : **MainBindingNavigator**.

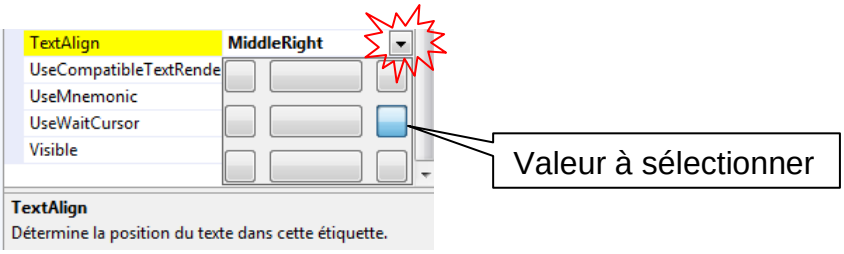


3. Ajoutez maintenant les deux derniers contrôles dans les deux autres cellules du second tableau :
   - Faites un glisser déplacer de la **Boîte à outils** > rubrique **Contrôles Communs** > d'un contrôle **Label** dans la deuxième cellule du tableau **TableLayoutPanel2**.
   - Faites un glisser déplacer de la **Boîte à outils** > rubrique **Contrôles Communs** > d'un contrôle **TextBox** dans la troisième et dernière cellule du tableau **TableLayoutPanel2** :

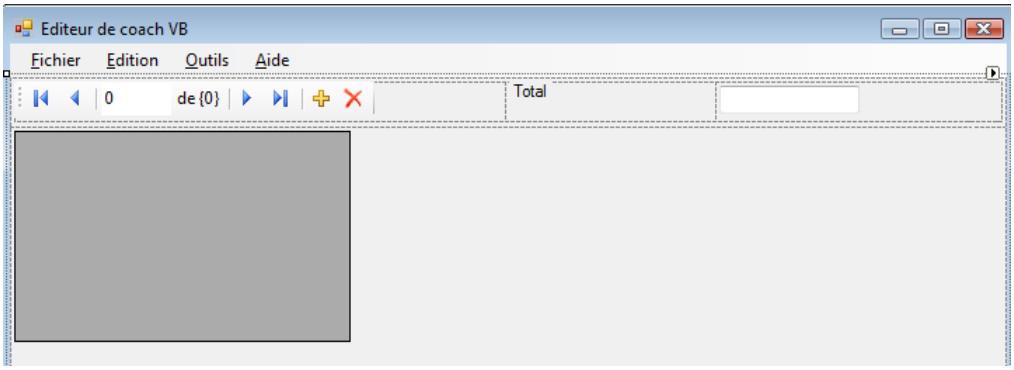


   - Affichez la fenêtre de **Propriétés** du contrôle **TextBox1**.
   - Configurez sa propriété **(Name)** avec la valeur : **TotalTextBox**.
   - Configurez sa propriété **ReadOnly** avec la valeur : **True** pour indiquer que la zone de texte sera en lecture uniquement.
   - Sélectionnez le contrôle **Label1**.
   - Configurez sa propriété **(Name)** avec la valeur : **TotalLabel**.
   - Configurez sa propriété **Text** avec la valeur : **Total**.
   - Configurez sa propriété **TextAlign** avec la valeur **MiddleRight** pour aligner le libellé à droite avec un centrage vertical.

Utilisez l'éditeur de propriété associé en cliquant sur la flèche à droite de la ligne de propriété.

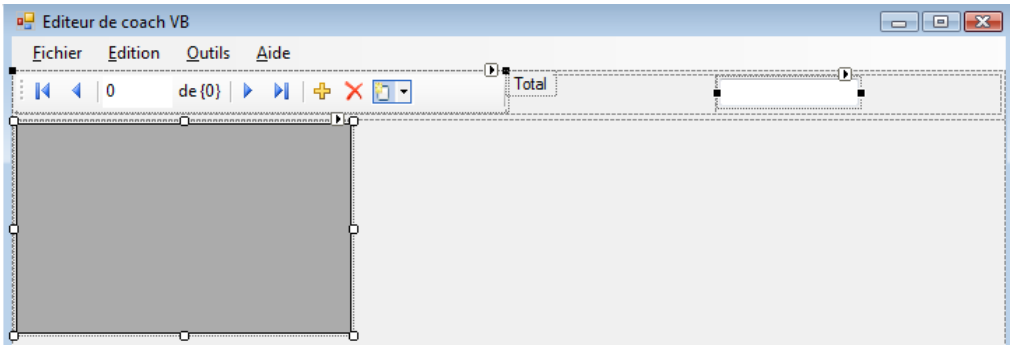


Vous obtenez :

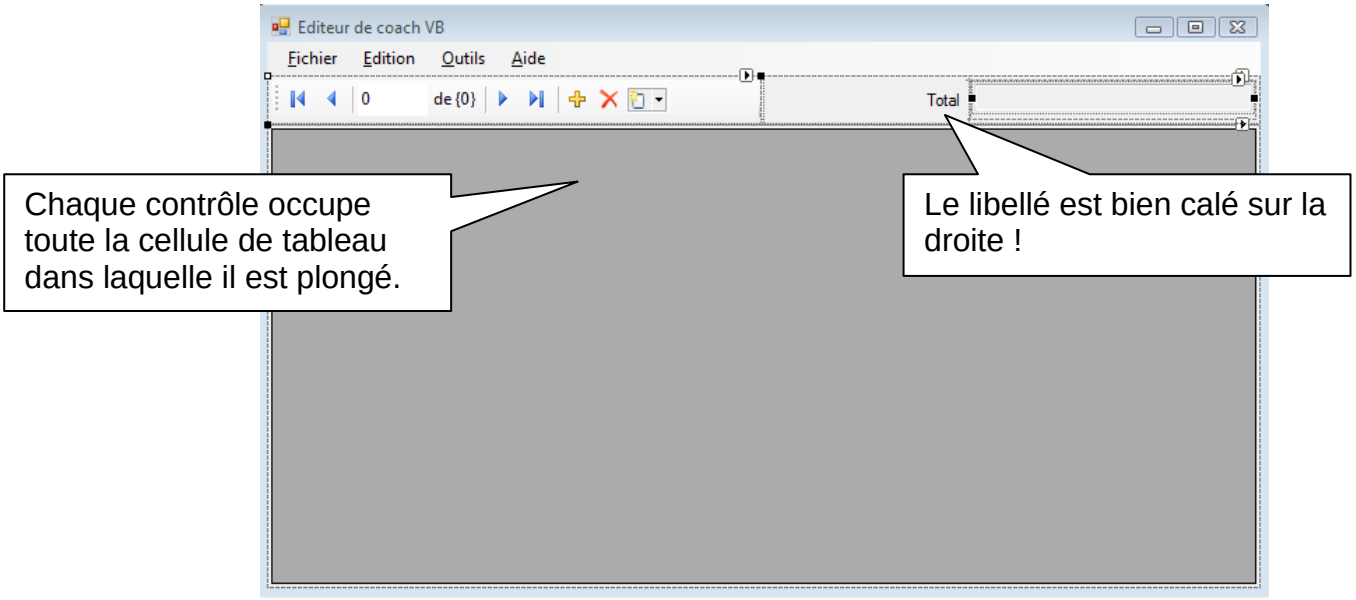


Le résultat n'est pas vraiment satisfaisant car chaque contrôle reste mal dimensionné. C'est là qu'il faut penser aux propriétés d'ancrage (**Dock**, **Anchor**) des contrôles vis-à-vis de leur conteneur parent.

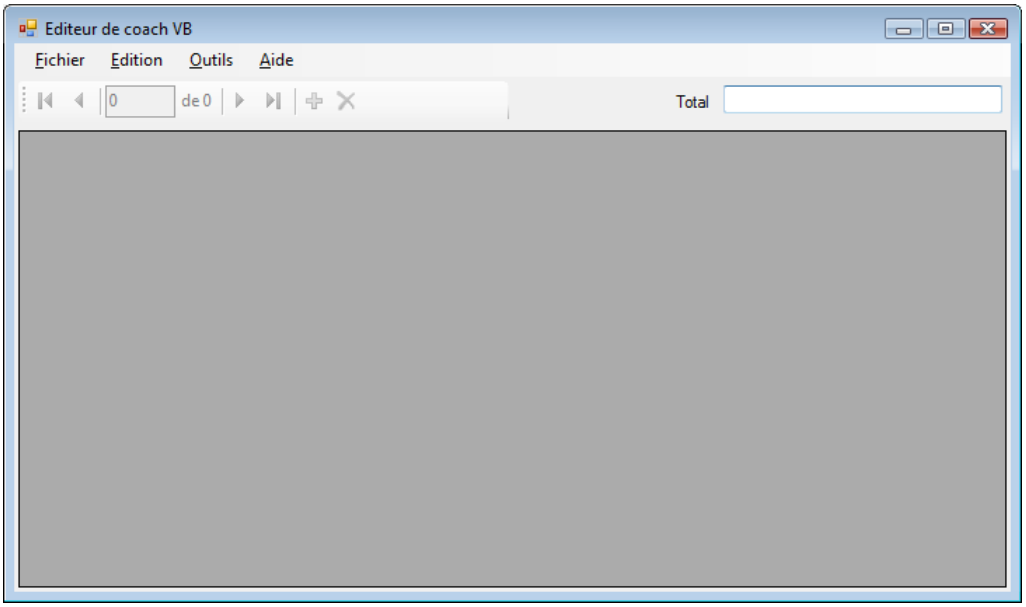
- Tout en maintenant la touche **Shift** appuyée, cliquez sur chacun des quatre contrôles, **TotalTextBox**, **TotalLabel**, **MainBindingNavigator** et **MainDataGridView** pour effectuer une sélection multiple de contrôles.
- Relâchez la touche **Shift**.



- Affichez la fenêtre de propriétés qui vous propose toutes les propriétés communes des contrôles de la sélection.
- Configurez la propriété **Dock** à la valeur **Fill** de façon à ce que chaque contrôle remplisse toute la surface disponible de son conteneur. Vous obtenez :



4. Testez l'affichage de notre interface à l'exécution :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).



Pour l'instant la barre de navigation est inactive car aucune donnée n'est affichée dans la grille mais l'interface fonctionne. Maintenant il ne reste plus qu'à programmer le code pour alimenter la grille avec le contenu d'un fichier CSV.

3 Alimenter les contrôles de données avec un fichier CSV

Dans cet exercice, vous allez apprendre à :

- Lire et enregistrer un fichier au format CSV,
- Alimenter des contrôles d'affichage avec des données,
- Utiliser une table mémoire,
- Mettre en œuvre le mécanisme de liaison de données (databinding).

Objectif
L'objectif de cet exercice est d'apprendre à manipuler des données en provenance d'un fichier texte délimité (*.csv).

La démarche que nous allons adopter consiste à utiliser un objet mémoire appelé **DataTable**, qui va servir d'intermédiaire entre le fichier et les contrôles d'affichage de notre application.

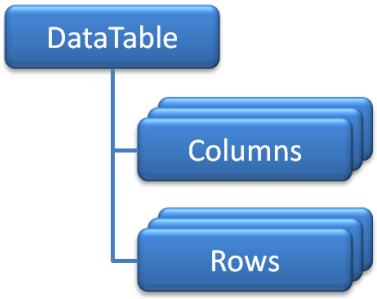
En effet, nous allons voir qu'il est très simple de faire interagir des contrôles d'affichage de données et une source de données en mémoire via le mécanisme dit de liaison de données (databinding). Un tel mécanisme n'existe pas directement sur une ressource de type fichier.

3.1 Créer une table mémoire (DataTable)

Dans ce premier exercice, nous allons préparer la structure de données mémoire dans laquelle nous chargerons les données de fichier.

Pour stocker les données en mémoire, nous vous proposons d'utiliser un objet de type **DataTable**.

C'est quoi exactement une **DataTable** ?
DataTable est une classe de l'espace de noms **System.Data** fourni par le Framework .NET, qui représente une table de données en mémoire. Qui dit table, dit **colonnes** et bien sûr **lignes**. La structure globale d'une **DataTable** est en effet la suivante :



- La propriété **Columns** d'un objet **DataTable** 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 la table. Les objets de cette collection sont de type **DataRow**.

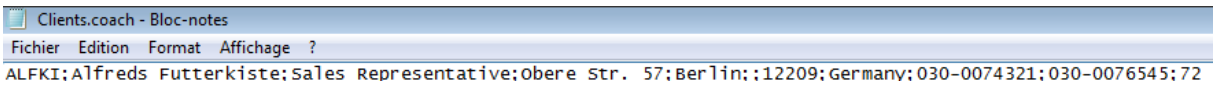
Autrement dit, ce que nous vous proposons de faire dans cet exercice, est de définir la structure de la table en mémoire en utilisant sa collection **Columns**. Nous manipulerons par la suite les données à l'aide de sa collection **Rows**.

Pour tout savoir sur l'espace de nom **System.Data** :
http://msdn2.microsoft.com/fr-fr/library/ax3wd0k9.aspx

Pour tout savoir sur la classe **DataTable** :
http://msdn2.microsoft.com/fr-fr/library/9186hy08.aspx

Contexte fonctionnel

Dans notre cas, les données sont stockées dans des fichiers texte délimités dont un enregistrement est composé de douze champs comme suit :



Nom	*Description*	*Type*
Id	Code d'indentification du client	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

La table de données en mémoire devra donc présenter la même structure soit douze colonnes.

Déroulement de l'exercice :

1. Définissez la structure de la table mémoire dans une fonction isolée de la classe Main appelée par exemple CreateDataTable :
   - Faites un clic droit dans l'**Explorateur de solutions** sur le fichier **Main.vb** > **Afficher le code** pour afficher le code de la classe **Main**.
   - Juste après la fin du constructeur **New** de la classe, ajoutez la définition d'une nouvelle fonction **CreateDataTable** comme suit :

Code VB

```
Public Class Main
   Public Sub New()
      …
   End Sub
   Function CreateDataTable() As DataTable
   End Function
   …
End Class
```

L'objectif de cette fonction est de créer un objet représentant la table en mémoire et de retourner une référence vers celui-ci à l'appelant. Le type de la valeur de retour est donnée par le mot clé **As** à la fin de la ligne de déclaration de la fonction.

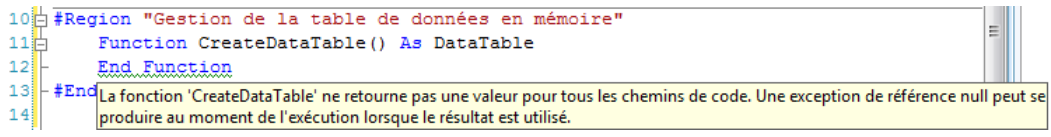
Function <NomDeLaFonction>(<Liste des paramètres>) As <TypeDeLaValeurRetour>

- Entourez la fonction d'une région nommée **Gestion de la table de données en mémoire** :

Code VB

```
#Region "Gestion de la table de données en mémoire"
   Function CreateDataTable() As DataTable
   End Function
#End Region
```

A priori, vous devriez voir apparaître un trait vert en dessous de **End Function**. Rappelez-vous que ces codes couleurs sont des indications précieuses de Visual Studio. Pour avoir le détail du problème dans une info bulle, il suffit de stopper la souris sur la ligne de couleur.
Visual Studio vous informe ici qu'il ne détecte pour l'instant aucune valeur de retour dans votre fonction et qu'il y a donc lieu de s'en préoccuper.

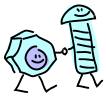


- Ajoutez la définition d'une variable de type **DataTable** puis préparer la ligne de retour de la fonction avec l'objet ainsi créé :

Code VB

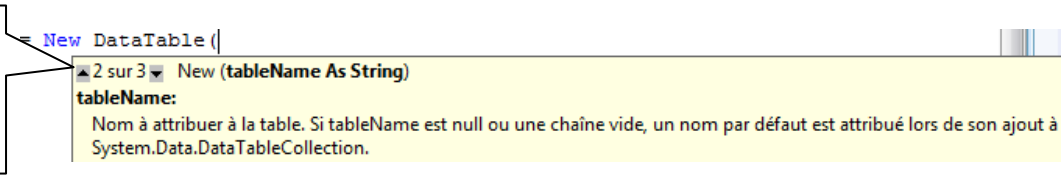
```
#Region "Gestion de la table de données en mémoire"
    Function CreateDataTable() As DataTable
        Dim result As DataTable = New DataTable("Coach")
        Return result
    End Function
#End Region
```

Avec ces deux lignes, le trait vert en dessous de **End Function** doit disparaître !

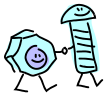


Quel est l'objet de la première ligne ?
Nous avons vu dans l'atelier précédent que le mot clé **New** permet de créer un nouvel objet d'un type donné et qu'il est suivi du *constructeur* de la classe concernée. Lorsque vous tapez la première parenthèse du constructeur, l'IntelliSense vous donne l'ensemble des constructeurs disponibles sur ce type d'élément. En fait il y a différents constructeurs d'un objet car il y a tout simplement plusieurs façons différentes d'initialiser l'objet. Par exemple, ici nous avons choisi le second qui permet d'initialiser le nom de la table que nous allons nommée : **Coach**.

Utiliser les flèches pour balayer les constructeurs disponibles



2. Ajoutez une première colonne à la table correspondant à l'Id d'un enregistrement de données dans notre fichier :



Une colonne est elle-même un objet, dont le type est donné par la classe **DataColumn**. Donc pour créer une colonne, il suffit d'utiliser le mot clé **New** suivi du constructeur de l'objet comme précédemment.

- Ajoutez la création d'un objet de type **DataColumn** comme suit :

Code VB

```
#Region "Gestion de la table de données en mémoire"
    Function CreateDataTable() As DataTable
        Dim result As DataTable = New DataTable("Coach")
        Dim idColumn As DataColumn = New DataColumn()
        Return result
    End Function
#End Region
```

Pour être défini précisément, vous devez indiquer le type de données qui seront contenues dans la colonne ainsi que son nom.

Pourquoi est-ce que c'est si important de définir le type des données ?
Parce que si vous imposez un maximum de restrictions à l'entrée de votre table, vous évitez d'autant les erreurs intempestives. Les types des données doivent par exemple correspondre avec ceux de la source de données, à savoir notre fichier plat.
Il faut savoir qu'un objet de type **DataColumn** peut être extrêmement riche et que son rôle est d'être capable de reproduire les mêmes contraintes qu'un schéma de base de données (contraintes d'unicité, colonne auto incrémentale, valeur nulle autorisée etc…)

Pour en savoir plus sur la classe **DataColumn** :
http://msdn.microsoft.com/fr-fr/library/system.data.datacolumn.aspx

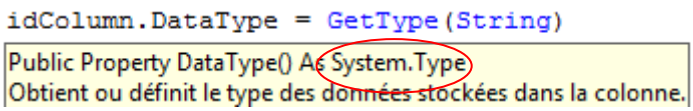
- Ajoutez les lignes suivantes pour configurer le nom et le type de la colonne :

Code VB

```
#Region "Gestion de la table de données en mémoire"
    Function CreateDataTable() As DataTable
        Dim result As DataTable = New DataTable("Coach")
        Dim idColumn As DataColumn = New DataColumn()
        idColumn.ColumnName = "Id"
        idColumn.DataType = GetType(String)
        Return result
    End Function
#End Region
```

Qu'est ce que *GetType* ?
GetType est un opérateur du langage Visual Basic. Il est utilisé pour récupérer l'objet correspondant à un type du langage.

Qu'est ce que c'est que l'objet correspondant à un type de données ?
En fait, la propriété **DataType** de notre objet **idColumn** doit référencer un objet qui définisse le type de données de la colonne. Ce n'est pas le type de données en lui-même tel que **String** ou **Integer**, non, non… Ce doit être un objet qui *décrive* le type. C'est un peu comme au restaurant. Pour commander, vous ne demandez pas les plats mais la carte du menu qui décrit chaque plat.



Chaque type du système de type communs du Framework .NET est décrit dans un objet de type **System.Type**.

Et comment est-ce qu'on fait pour dégoter un objet qui décrive par

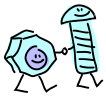
exemple le type de données **String** ?
C'est tout simple, il suffit de faire appel à l'opérateur **GetType** en lui donnant le type de données spécifié ☺ !

Pour en savoir plus sur l'opérateur **GetType** :
http://msdn.microsoft.com/fr-fr/library/tay4kywk.aspx

- Ajoutez une dernière ligne pour ajouter la colonne que vous avez créée à la collection des colonnes de la table **Coach** :

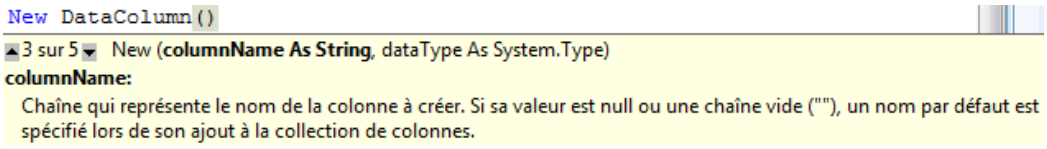
Code VB

```
#Region "Gestion de la table de données en mémoire"
    Function CreateDataTable() As DataTable
        Dim result As DataTable = New DataTable("Coach")
        Dim idColumn As DataColumn = New DataColumn()
        idColumn.ColumnName = "Id"
        idColumn.DataType = GetType(String)
        result.Columns.Add(idColumn)
        Return result
    End Function
#End Region
```



Au final, toutes ces lignes pour créer une seule colonne, cela fait pas mal de lignes en perspectives pour créer les dix colonnes restantes…
Heureusement il existe un moyen de réduire le code :

- D'abord en utilisant la notion de *constructeur* d'objet pour initialiser le nom et le type de la colonne en une seule ligne. Si vous utilisez l'IntelliSense, vous constaterez qu'en effet la classe **DataColumn** propose un constructeur qui permet d'initialiser les deux informations en une seule instruction :

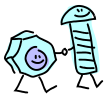


- Supprimez les deux lignes d'initialisation de l'objet et modifiez la ligne de déclaration et de construction de l'objet comme suit :

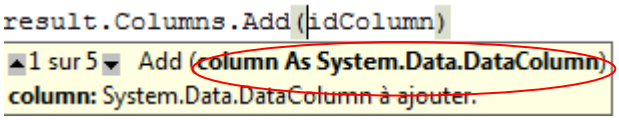
Code VB

```
#Region "Gestion de la table de données en mémoire"
    Function CreateDataTable() As DataTable
        Dim result As DataTable = New DataTable("Coach")
        Dim idColumn As DataColumn = New DataColumn("Id", GetType(String))
        idColumn.ColumnName = "Id"
        idColumn.DataType = GetType(String)
        result.Columns.Add(idColumn)
        Return result
    End Function
```

```
#End Region
```



- Ensuite il faut savoir que l'opérateur **New** peut être utilisé directement partout où l'on attend une référence d'objet.
  Par exemple, c'est le cas de la méthode **Add** qui ajoute un objet de type **DataColumn** à la collection **DataColumns** de l'objet **DataTable**. Il lui faut simplement en paramètre une référence vers l'objet à ajouter :



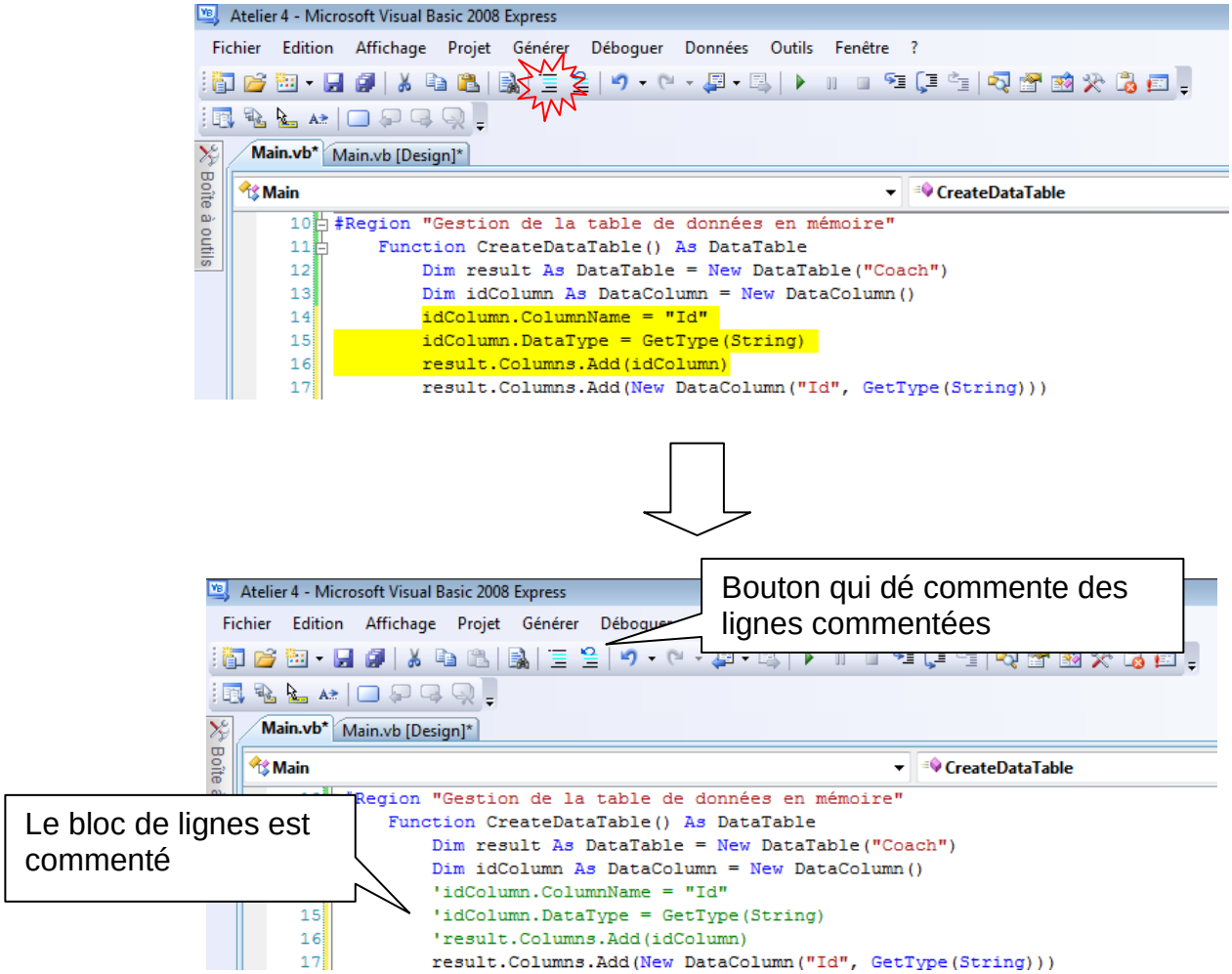
  Cette référence, c'est celle que nous avons appelée **idColumn** et définie comme étant de type **DataColumn**. On va pouvoir se passer de déclarer cette variable !

- Supprimez la déclaration de la variable **idColumn** et insérez directement le code de création de la colonne en paramètre de la méthode **Add** en utilisant l'opérateur **New** :

Code VB

```
#Region "Gestion de la table de données en mémoire"
    Function CreateDataTable() As DataTable
        Dim result As DataTable = New DataTable("Coach")
        Dim idColumn As DataColumn = New DataColumn("Id", GetType(String))
        result.Columns.Add(New DataColumn("Id", GetType(String)))
        Return result
    End Function
#End Region
```

Plutôt que de supprimer les lignes de code, vous pouvez également les mettre en commentaire. Pour mettre un bloc de plusieurs lignes en commentaire, il suffit de sélectionner (grossièrement) les lignes avec la souris puis de cliquer le bouton de la barre d'outils standard de Visual Studio.
A l'inverse, pour dé commenter ces même lignes, il suffit de cliquer le bouton .



3. Ajoutez maintenant les dix autres colonnes à la table :
   - A la suite dans la fonction **CreateDataTable**, ajoutez les autres colonnes qui structurent la table :

Code VB

```
#Region "Gestion de la table de données en mémoire"
    Function CreateDataTable() As DataTable
        Dim result As DataTable = New DataTable("Coach")
        result.Columns.Add(New DataColumn("Id", GetType(String)))
        result.Columns.Add(New DataColumn("Contact", GetType(String)))
        result.Columns.Add(New DataColumn("Titre", GetType(String)))
        result.Columns.Add(New DataColumn("Adresse", GetType(String)))
        result.Columns.Add(New DataColumn("Ville", GetType(String)))
        result.Columns.Add(New DataColumn("Region", GetType(String)))
        result.Columns.Add(New DataColumn("Code postal", GetType(String)))
        result.Columns.Add(New DataColumn("Pays", GetType(String)))
        result.Columns.Add(New DataColumn("Telephone", GetType(String)))
        result.Columns.Add(New DataColumn("Telecopie", GetType(String)))
        result.Columns.Add(New DataColumn("CA", GetType(Integer)))
        Return result
    End Function
```

Attention à la dernière colonne qui est de type Integer (et non String)

```
#End Region
```

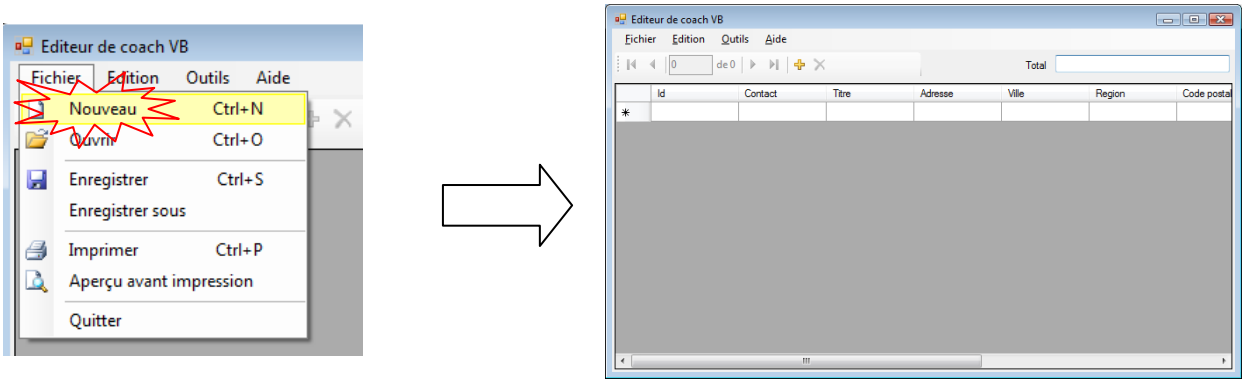
Parfait ! De cette façon, vous disposez maintenant d'une table en mémoire reflétant le schéma de votre source de données. Vous allez donc pouvoir la charger avec des données en provenance de fichier. Vous pouvez également l'utiliser dès maintenant comme une structure de données vierge que l'utilisateur peut enrichir avec de nouvelles données.
Dans les deux cas, nous avons besoin de lier la table mémoire avec des contrôles de données d'affichage qui vont permettre d'interagir avec l'utilisateur.

3.2 Comprendre la liaison de données (DataBinding)

L'objectif de cet exercice est de présenter le mécanisme de liaison de données (ou DataBinding en anglais).

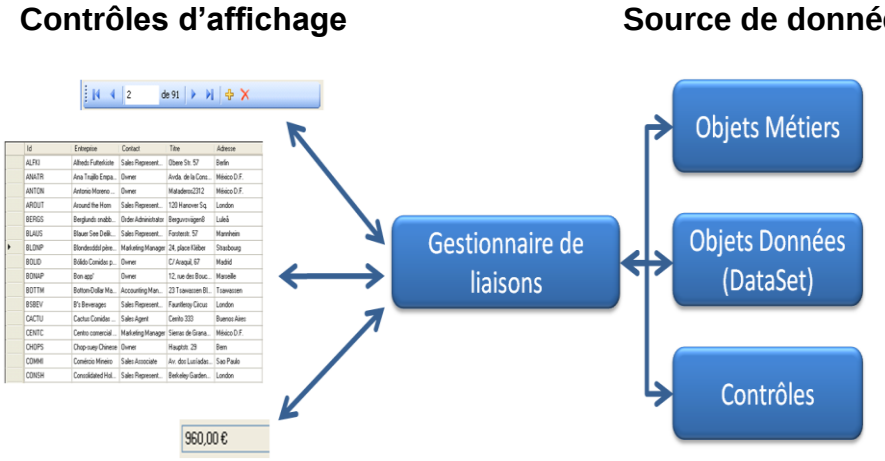
Contexte fonctionnel

Avant de se lancer dans la manipulation d'un fichier, nous vous proposons d'exploiter la table de données mémoire à l'état vierge pour créer de nouvelles données (en vue bien entendu de les sauvegarder par la suite sur une structure fichier).
L'utilisateur clique le menu **Fichier** > **Nouveau** de l'application. Une grille vide doit apparaître dans le formulaire de façon à ce qu'il puisse saisir des données.



Déroulement de l'exercice :

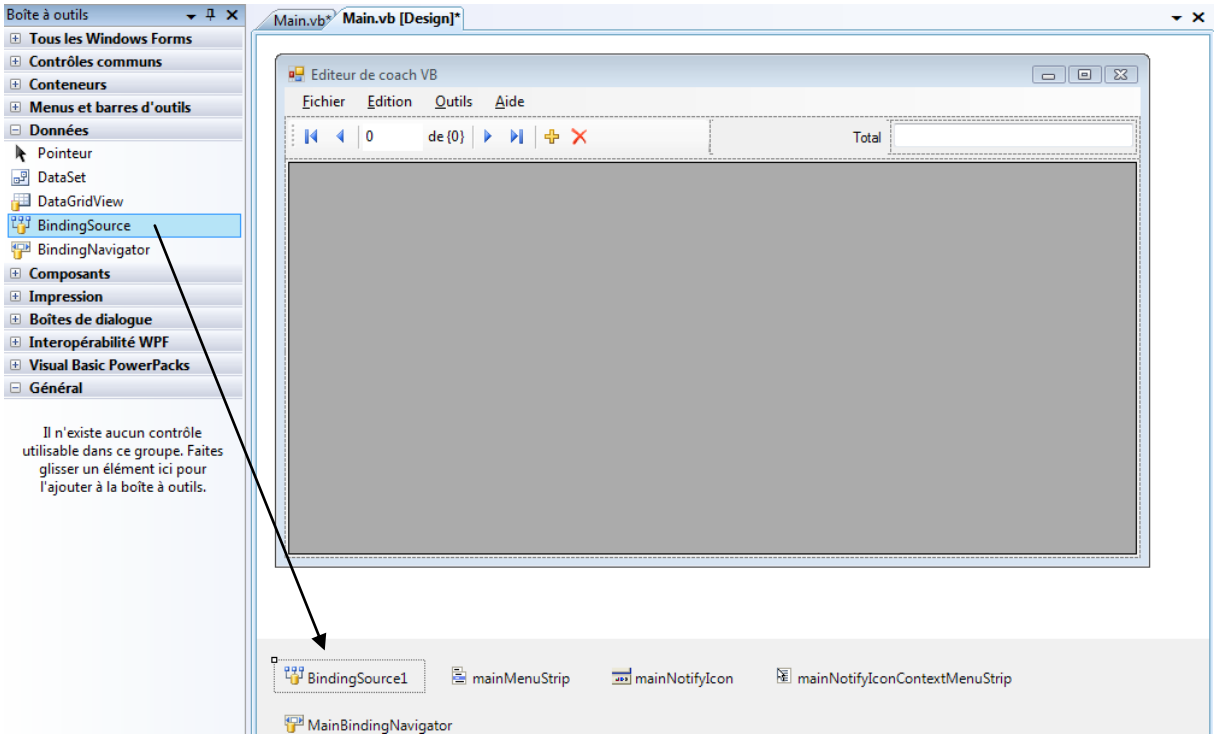
Pourquoi est-ce qu'on a besoin du *DataBinding* ?
Le **DataBinding** est le mécanisme par lequel la valeur d'une propriété d'un contrôle d'affichage est automatiquement maintenue à jour en fonction de la valeur de la propriété d'un objet fournissant des données. Le contrôle d'affichage et l'objet source de données sont étroitement liés (d'où le terme *binding*) si bien que la modification de l'un entraîne immédiatement la mise à jour en correspondance de l'autre et vice versa.
On peut schématiser le fonctionnement comme suit :



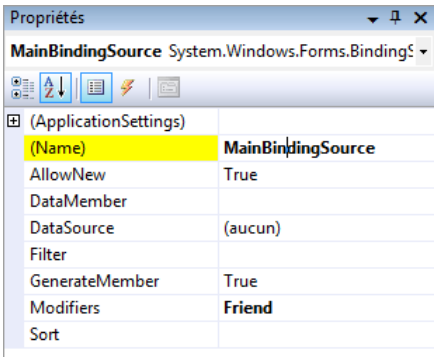
Qu'est ce qu'un *gestionnaire de liaisons* ?
Pour simplifier la liaison de données, le modèle des Windows Forms fournit un composant appelé **BindingSource** qui agit comme un intermédiaire entre la source de données et les contrôles d'affichage dépendants. On parle de *gestionnaire de liaisons* car il gère la liaison, telle que la notification des modifications de part et d'autre.

Pour tout savoir sur le modèle de liaison de données avec les Windows Forms :
http://msdn.microsoft.com/fr-fr/library/ef2xyb33.aspx

1. La première étape consiste donc à créer un gestionnaire de liaison pour gérer l'interconnexion entre nos contrôles d'affichage de données et la table mémoire :
   - Basculez sur le formulaire **Main.vb** en mode **Design**.
   - Faites un glisser-déplacer de la **Boîte à outils** > rubrique **Données** > du contrôle **BindingSource** sur la zone de dépôt de contrôles du formulaire :



- Affichez les propriétés du contrôle **BindingSource1** (F4).
- Configurez la propriété **(Name)** avec la valeur : **mainBindingSource** :



Pour tout savoir sur la classe **BindingSource** :
http://msdn.microsoft.com/fr-fr/library/h974h4y2.aspx

2. La seconde étape consiste à créer un gestionnaire d'évènement en réponse à l'évènement clic du menu Fichier > Nouveau de l'application :
   - Toujours sur le formulaire **Main.vb** en mode **Design**, sélectionnez le menu **Fichier** puis double cliquez sur l'option de menu **Nouveau** pour générer la procédure **NouveauToolStripMenuItem_Click** :

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click


End Sub
```

3. Créez la table mémoire à l'aide de la fonction CreateDataTable écrite dans l'exercice précédent :

- Dans la procédure **NouveauToolStripMenuItem _Click**, ajoutez le code de création d'une table mémoire comme suit :

Code VB

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click
    'Création d'une nouvelle table mémoire vide
    Dim newDataTable As DataTable
    newDataTable = CreateDataTable()
End Sub
```

4. Indiquez au gestionnaire de liaisons mainBindingSource que la source de données à prendre en charge est la table mémoire newDataTable :
    - Attribuez la nouvelle table à la propriété **DataSource** de l'objet **mainBindingSource** :

Code VB

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click
    'Création d'une nouvelle table mémoire vide
    Dim newDataTable As DataTable
    newDataTable = CreateDataTable()
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
End Sub
```

5. Liez les contrôles d'affichage de données avec le gestionnaire de liaisons :
    - D'abord le contrôle de grille de données **mainDataGridView** avec sa propriété **DataSource** :

Code VB

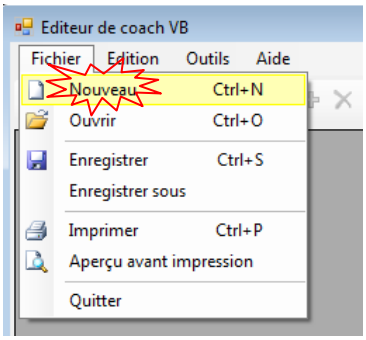
```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click
    'Création d'une nouvelle table mémoire vide
    Dim newDataTable As DataTable
    newDataTable = CreateDataTable()
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
End Sub
```

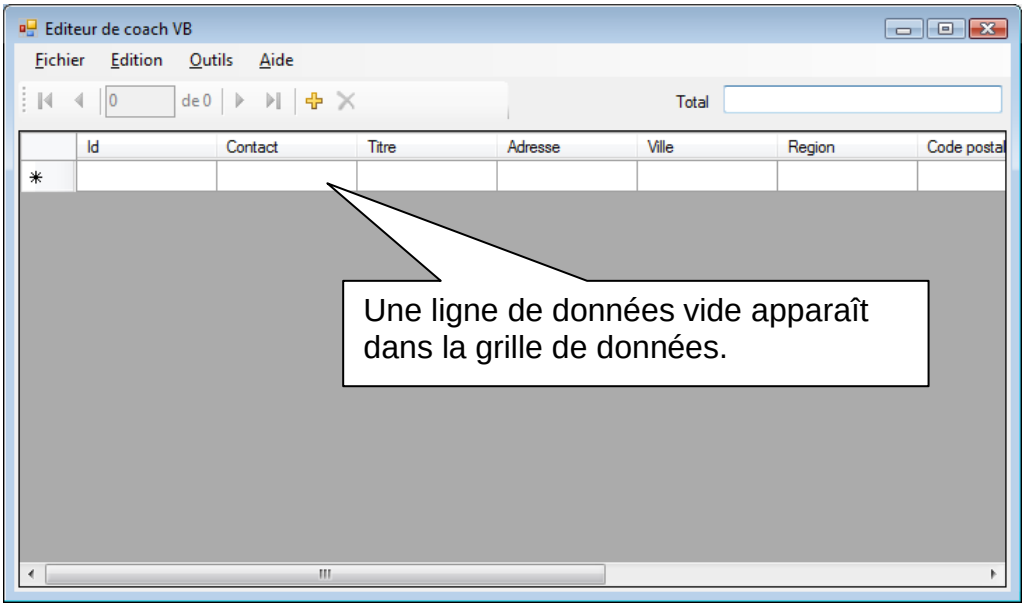
    - Ensuite le contrôle de navigation **mainBindingNavigator** avec sa propriété **BindingSource** :

Code VB

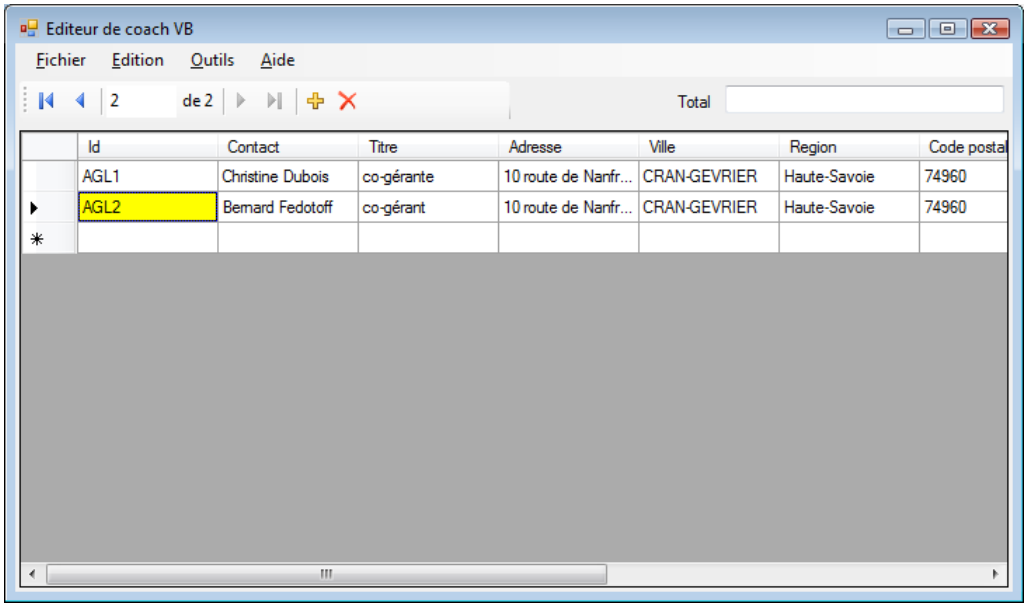
```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click
    'Création d'une nouvelle table mémoire vide
    Dim newDataTable As DataTable
    newDataTable = CreateDataTable()
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.BindingSource = MainBindingSource
End Sub
```

6. Testez le fonctionnement des contrôles et de la table mémoire :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier > Nouveau** :



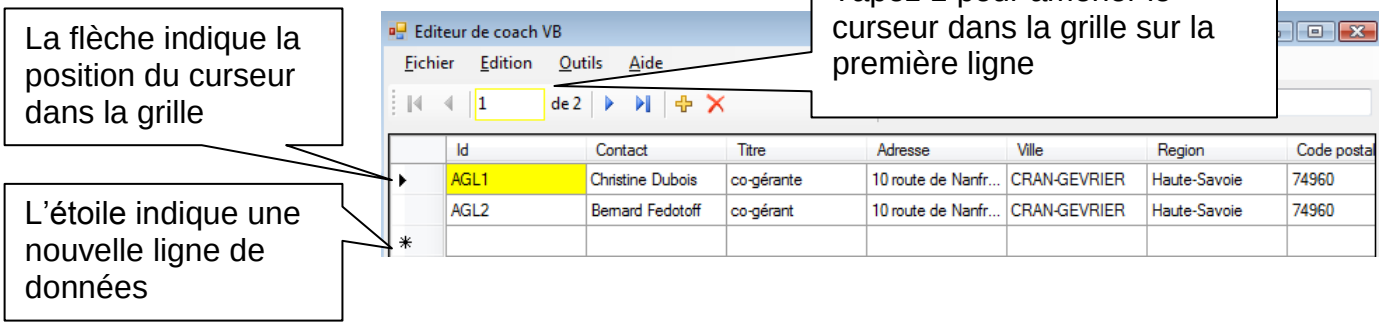


   - Amusez-vous à saisir d'autres informations.



Vérifiez que la barre de navigation fonctionne également. Par exemple :

- sélectionnez une ligne de données et naviguez d'une ligne à l'autre en cliquant sur les boutons ou ,
- ou entrez directement le numéro de la ligne souhaitée dans la zone de texte :



- Ajoutez une nouvelle ligne en cliquant ou en vous positionnant sur la grille en face de la ligne marquée d'une étoile (*),
- Supprimez une ligne en cliquant .

Bravo ! La grille fonctionne, liée à la table mémoire.

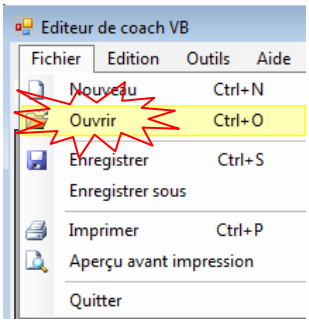
3.3 Manipuler le fichier CSV

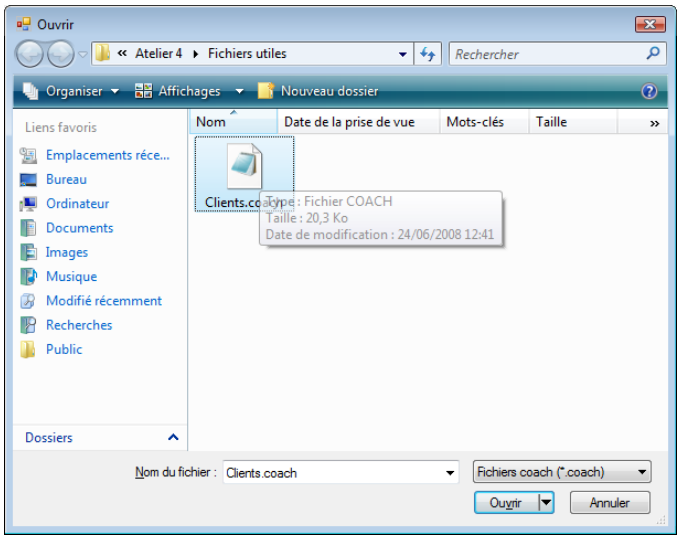
L'objectif de cet exercice est de lire les données de la grille à partir d'un fichier au format CSV (texte délimité), et bien sur de sauvegarder de nouveau ces données, après modification, dans le même fichier CSV (ou un autre d'ailleurs)

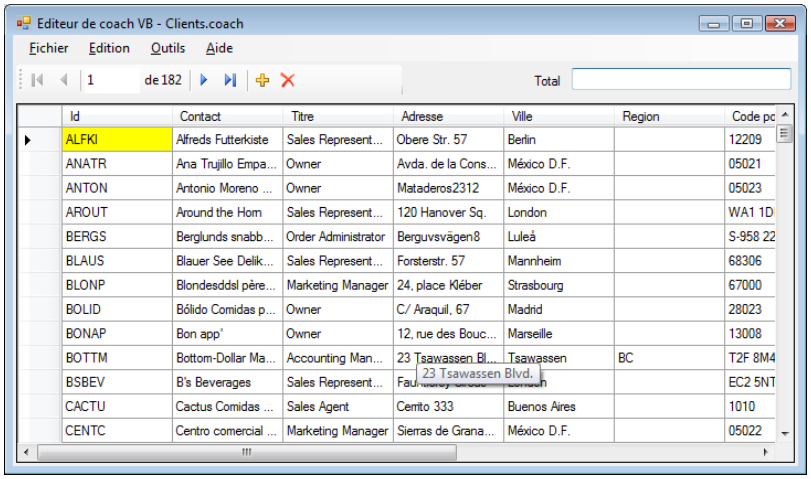
3.3.1 Lire le fichier

Contexte fonctionnel

Pour charger un fichier dans la grille de données de l'application, l'utilisateur clique le menu **Fichier** > **Ouvrir** puis sélectionne le fichier de données dans la structure de répertoires de la machine :







Il peut ensuite travailler sur les données : ajouter de nouveaux enregistrements, supprimer des enregistrements, mettre à jour des informations existantes etc…

Un fichier CSV est composé de ligne de valeurs, séparées par un séparateur qui sera dans notre cas 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 pour lire un tel fichier et le charger dans la grille de données du formulaire est composé de quatre étapes :

1. Sélectionner le fichier,
2. Ouvrir le fichier,
3. Lire le fichier CSV ligne à ligne,

4. Pour chaque ligne de texte lue, générer un enregistrement dans la table de données.

Etape 1 : La première étape consiste donc à écrire le code de sélection du fichier sur le disque.

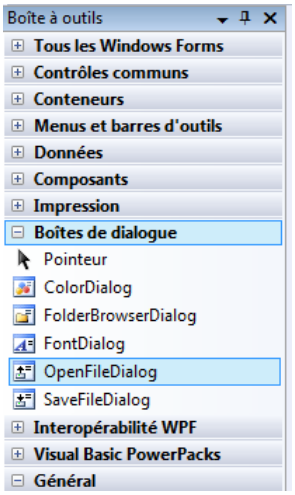
Nous allons coder l'ouverture du fichier sur le menu **Fichier** > **Ouvrir** du formulaire **Main**. Cela revient donc à créer un gestionnaire d'évènement associé à l'évènement **Click** de l'option de menu correspondante.

1. Créez un gestionnaire d'évènement associé au clic du menu Ouvrir :
   - Affichez le formulaire **Main** en mode **Design**.
   - Sélectionnez le menu **Fichier** puis double cliquez sur l'option de menu **Ouvrir** pour générer la procédure **OuvrirToolStripMenuItem_Click** :

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click

End Sub
```

2. Utilisez la boîte de dialogue standard d'ouverture de fichier de Windows pour permettre à l'utilisateur de faire sa sélection :

Vous vous souvenez des boîtes de dialogue communes fournies par le Framework .NET ? Nous en avons utilisé une dans l'atelier 3 de ce tutorial pour sélectionner un chemin de répertoire sur le disque. Nous allons utiliser ici la boîte **OpenFileDialog** pour sélectionner un fichier sur le disque.
Pour rappel, ces boîtes de dialogue se manipulent en tant qu'objets que Visual Studio peut créer pour vous via les composants de la **Boîte à outils** > rubrique **Boîtes de dialogue**.
Vous pouvez aussi créer vos propres objets dans le code directement à partir des classes du Framework .NET. C'est cette deuxième approche que nous allons adopter.



- Ajoutez à la procédure le code de déclaration et d'instanciation d'un nouvel objet de type **OpenFileDialog** :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
End Sub
```

Les principales propriétés à configurer sur cet objet sont :
- **Filter** : Cette propriété sert à filtrer la liste des fichiers de la boîte

de dialogue, typiquement sur l'extension de fichier pour orienter l'utilisateur directement sur le type de fichier autorisé. Nous pouvons donc filtrer tous les fichiers d'extension ***.coach**.

- **InitialDirectory** : Cette propriété configure le répertoire affiché par défaut lors de l'ouverture de la fenêtre.

- Complétez le code pour filtrer le type de fichier sur l'extension ***.coach** :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
End Sub
```

- Configurez le répertoire par défaut sur le dossier préférentiel de l'utilisateur configuré à l'aide de la fenêtre **Options** à l'atelier 3 dans la variable privée **SaveDirectoryPath** de la classe **Main** :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
End Sub
```

Reste à afficher la boîte de dialogue. Toujours dans le même atelier 3 vous avez appris à utiliser la méthode **ShowDialog**. Le code d'ouverture du fichier doit être exécuté si l'utilisateur clique **OK** pour fermer la boîte d'ouverture.

- Affichez la boîte de dialogue et testez le code de fermeture avec une condition **If…Then…End If** :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog =  _
            Windows.Forms.DialogResult.OK Then
       'Code d'ouverture du fichier sélectionné
    End If
End Sub
```

Etape 2 : La seconde étape consiste à ouvrir le fichier.

Comment retrouve-t-on le fichier sélectionné par l'utilisateur ?
C'est la propriété **FileName** de l'objet **openDataFileDialog** qui donne le nom du fichier sélectionné par l'utilisateur dans la boîte de dialogue.
En réalité le nom de cette propriété (FileName) est un faux ami car elle indique non seulement le nom du fichier mais aussi le chemin complet pour y accéder.

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, plutôt que d'écrire le code d'ouverture du fichier directement dans le gestionnaire d'évènement précédent, nous allons coder une nouvelle procédure. Et nous procèderons de même pour les autres étapes de notre algorithme.

3. Ajoutez une fonction dont l'objectif est d'ouvrir et lire le contenu d'un fichier :
   - Toujours dans le fichier de code **Main.vb**, juste après la fin du constructeur **New** de la classe **Main**, ajoutez la définition d'une nouvelle fonction **ReadFile** comme suit :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
End Function
```

Notez que la fonction a évidemment besoin du chemin du fichier qu'il lui faudra ouvrir. C'est l'objet du paramètre **FileName**.

Pourquoi prévoir une **valeur de retour** de type **DataTable** ?
Pour rappel, notre objectif est de lire le fichier pour charger son contenu dans la table mémoire. Voilà pourquoi le résultat de l'opération que nous vous proposons est un objet de type **DataTable** qu'il suffira de lier aux contrôles d'affichage de données comme nous l'avons vu dans l'exercice précédent.

- Codez aussitôt la déclaration d'une variable de retour et renvoyez-la à l'aide du mot clé **Return** :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
    Dim table As DataTable = Nothing
    'Renvoi de la valeur de retour
    Return table
End Function
```

Pensez à encadrer votre code à l'aide des régions !
Construisez par exemple une région pour retrouver facilement tous les codes en rapport avec le traitement de fichier !

- Entourez la fonction d'une région nommée **Traitement des fichiers de données** :

Code VB

```
#Region "Traitement des fichiers de données"
 Function ReadFile(ByVal FileName As String) As DataTable
     …
 End Function
#End Region
```

Nous allons avoir besoin de définir deux variables au sein de la fonction :

- **readRow**, de type **string,** dans laquelle nous récupérerons une à une les lignes de fichier lues. Chaque ligne devra être chargée dans la table mémoire.
- **isFirstRow**, de type **booléen**, pour différencier la lecture de la première ligne des autres lignes du fichier. En effet, au moment de la lecture de la première ligne il faudra prévoir de créer la table mémoire avant de penser à y charger la ligne lue.

- Ajoutez à la fonction **ReadFile** le code de définition des variables **readLine** et **isFirstLine** comme suit :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
    'Définition des variables locales à la fonction
    Dim table As DataTable = Nothing
    Dim readRow As String = String.Empty
    Dim isFirstRow As Boolean = Boolean.TrueString
    ‘Renvoi de la valeur de retour
    Return table
End Function
```

Pensez à initialiser vos variables pour éviter les erreurs (surprises) intempestives à l’exécution.

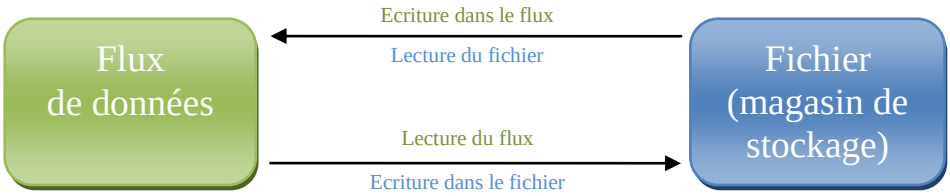
- Ajoutez à la suite le code suivant :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
    'Définition des variables locales à la fonction
    Dim table As DataTable = Nothing
    Dim readRow As String = String.Empty
    Dim isFirstRow As Boolean = Boolean.TrueString
    'Parcours du fichier à l’aide d’un StreamReader
    Dim sr As StreamReader = New StreamReader(FileName)

    ‘Renvoi de la valeur de retour
    Return table
End Function
```

C'est quoi un **StreamReader** ?
C'est un type d'objet qui lit des caractères à partir d'un flux d'octets dans un codage particulier. 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.

Qu'est ce qu'on entend par *flux* ?
Pour toute opération d'entrée/sortie de fichier de base, le principe consiste à passer par un flux. C'est lui qui gère la lecture ou l'écriture des octets stockés dans le fichier. Le fichier joue le rôle de magasin de stockage des données du flux.



C'est un peu comme si vous aviez besoin d'un *lecteur* pour lire le fichier à votre place, tout comme nous aurons besoin d'un *écrivain* pour écrire le fichier.

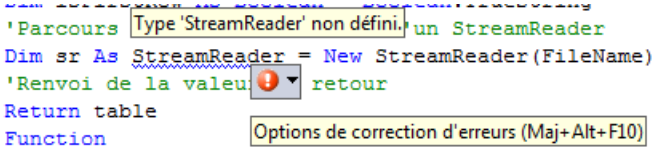
Pour tout savoir sur les flux de données :
http://msdn.microsoft.com/fr-fr/library/k3352a4t.aspx

Pour en savoir plus sur la classe **StreamReader** :
http://msdn2.microsoft.com/fr-fr/library/6aetdk20.aspx

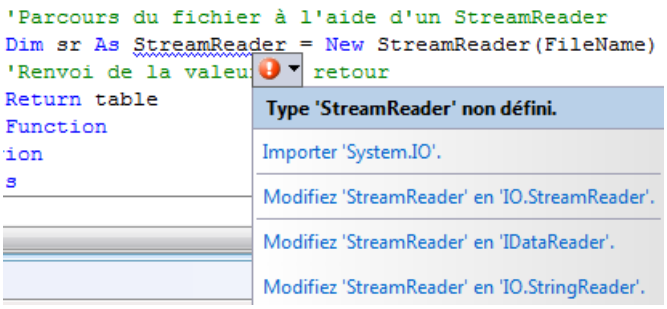
Avez-vous remarqué le surlignement bleu et le petit trait rouge en dessous de **StreamReader** ?

```
'Parcours du fichier à l'aide d'un StreamReader
Dim sr As StreamReader = New StreamReader(FileName)
```

Positionnez la souris sur le petit trait rouge en fin de mot pour faire apparaître la balise suivante :

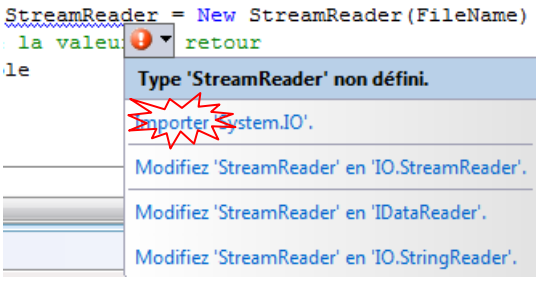


Visual Studio a un souci avec le type **StreamReader** qui ne lui dit rien du tout… Cliquez sur la petite flèche de la balise pour lister les corrections de l'erreur proposées :

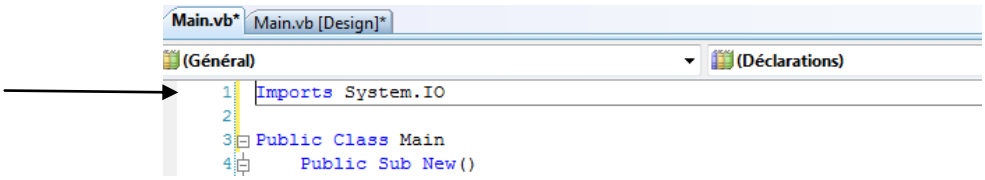


Visual Studio a bien trouvé une référence dans votre projet à la bibliothèque de classes du Framework .NET qui contient l'espace de nom **System.IO**. Il s'agit d'une référence automatiquement incluse par le modèle de projet que nous avons utilisé à la création du projet.
En revanche, il vous demande d'utiliser le nom qualifié complet de l'objet demandé pour être sûr de ne pas le confondre avec un objet de même nom d'une autre bibliothèque référencée.

Heureusement, il est possible d'éviter l'écriture du nom qualifié complet d'un type (par exemple System.IO.StreamReader) en définissant un **alias d'importation**.
C'est l'objet de la première suggestion de Visual Studio **Importer 'System.IO'**. Cliquez cette option parmi la liste des propositions.



Remontez au début du fichier **Main.vb** pour observer la ligne de code que Visual Studio a ajouté automatiquement :

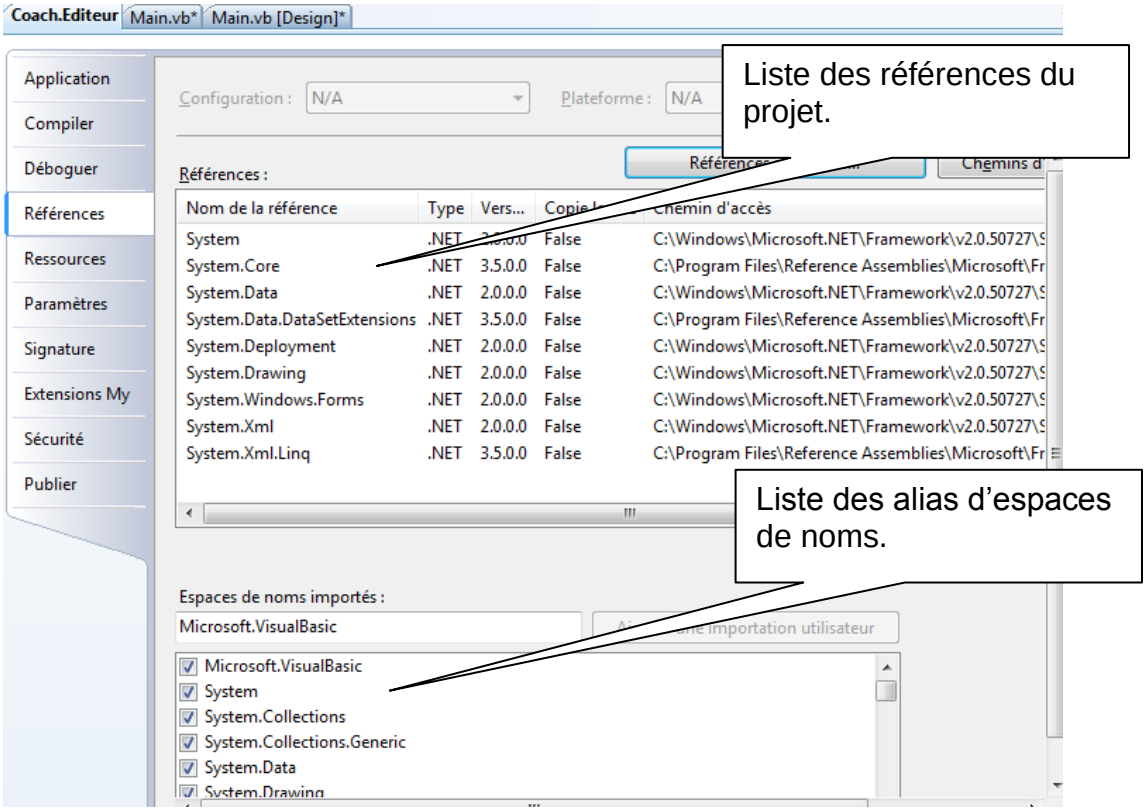


A partir de maintenant vous êtes dispensé d'écrire le nom complet des types de l'espace de noms **System.IO** au sein de ce fichier !

On aurait tout aussi bien pu définir cet alias au niveau du projet pour qu'il soit valable dans tous les fichiers de l'application. Les espaces de noms les plus courants sont d'ailleurs déjà enregistrés au niveau du projet pour vous faciliter l'utilisation des types standards du Framework .NET. C'est pourquoi nous ne nous étions encore jamais posé la question jusqu'à maintenant alors que nous avons déjà eu l'occasion d'utiliser à maintes reprises des classes du Framework .NET…

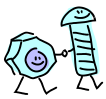
Pour retrouver les alias d'importation du projet :

- Cliquez **My Project** dans l'**Explorateur de solutions**.
- Sélectionnez l'onglet **Références**.



Il faut bien comprendre que la définition d'un alias ne remplace pas la définition de la référence. Ce sont deux étapes bien différentes. La définition de la référence est absolument nécessaire pour utiliser le type d'objet souhaité dans votre code alors que la définition d'un alias n'est pas du tout obligatoire. Il s'agit uniquement d'un confort d'écriture que vous pouvez décider de vous octroyer ou non.

Pour en savoir plus sur la définition d'un alias d'importation à l'aide du mot clé **Imports** :
http://msdn.microsoft.com/fr-fr/library/7f38zh8x.aspx



Fermons la parenthèse sur les alias et rouvrons une autre parenthèse ☺.
D'un point de vue objet, nous avions déjà évoqué le fait qu'il est très important de toujours veiller à supprimer les références d'objet que vous utilisez. Pour tout objet géré (*managé*) par le Framework.NET, vous savez que le GC (Garbage Collector) s'occupe de tout.
Mais ce n'est pas le cas pour les ressources non gérées (*non managées*), comme par exemple les ressources de type fichier.
Heureusement le langage Visual Basic fournit une structure de contrôle très pratique pour garantir la suppression de vos ressources à partir du moment où le code a fini de les utiliser. Le principe consiste à limiter la portée de la variable pointant sur la ressource à un bloc de code

déterminé par le bloc **Using…End Using**.

- Remplacez l'instruction **Dim** par **Using** pour gérer la portée de l'objet **StreamReader** comme suit :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
    'Définition des variables locales à la fonction
    Dim table As DataTable = Nothing
    Dim readRow As String = String.Empty
    Dim isFirstRow As Boolean = Boolean.TrueString
    'Parcours du fichier à l'aide d'un StreamReader
    ~~Dim~~ Using sr As StreamReader = New StreamReader(FileName)

    End Using
    'Renvoi de la valeur de retour
    Return table
End Function
```

La variable **sr** est utilisable à l'intérieur du bloc **Using…End Using**. Au-delà, elle sera détruite automatiquement.

msdn Pour tout savoir sur la structure de contrôle **Using** :
http://msdn.microsoft.com/fr-fr/library/ms172863.aspx

Etape 3 : La troisième étape consiste à lire les lignes du fichier une à une.

La méthode de la classe **StreamReader** que nous allons utiliser pour lire une ligne de donnée texte est **ReadLine()**. Pour traiter le fichier ligne à ligne nous allons également devoir utiliser une structure de boucle. Le bloc **Do…Loop** est idéal pour balayer un nombre d'élément dont on ne connait pas le nombre.

- Ajouter le code de lecture des lignes du fichier une à une :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
    'Définition des variables locales à la fonction
    Dim table As DataTable = Nothing
    Dim readRow As String = String.Empty
    Dim isFirstRow As Boolean = Boolean.TrueString
    'Parcours du fichier à l'aide d'un StreamReader
    Using sr As StreamReader = New StreamReader(FileName)
      'Lecture des lignes du fichier une à une jusqu'à la dernière ligne
      Do
        readRow = sr.ReadLine()
      Loop While readRow IsNot Nothing
    End Using
    'Renvoi de la valeur de retour
    Return table
End Function
```

Il existe plusieurs possibilités pour gérer la sortie d'une boucle **Do…Loop**, soit en début soit en fin de boucle.
La clause **While** positionnée en fin de boucle conditionne l'exécution d'au moins une fois du bloc de code. Si la référence **readRow** ne pointe sur aucun objet, alors la dernière ligne du fichier a été atteinte donc on peut arrêter le traitement.

msdn

Pour tout savoir sur la structure de boucle **Do…Loop** :
http://msdn.microsoft.com/fr-fr/library/eked04a7.aspx

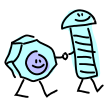
Il nous reste à programmer le traitement des lignes une à une. L'algorithme est le suivant :

- Dans le cas de la première ligne, créer la table mémoire puis ajouter la ligne à celle-ci.
- Dans le cas de toutes les autres lignes, ajouter la ligne à la table mémoire.

- Utiliser la variable **isFirstRow** (que nous avons initialisée à True) pour détecter la première ligne et créer la table mémoire dans la variable de retour **table** à l'aide de la fonction **CreateDataTable** :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
    'Définition des variables locales à la fonction
    Dim table As DataTable = Nothing
    Dim readRow As String = String.Empty
    Dim isFirstRow As Boolean = Boolean.TrueString
    'Parcours du fichier à l'aide d'un StreamReader
    Using sr As StreamReader = New StreamReader(FileName)
      'Lecture des lignes du fichier une à une jusqu'à la dernière ligne
      Do
        readRow = sr.ReadLine()
        If isFirstRow Then
            table = CreateDataTable()
            isFirstRow = False
        End If
        'Ajouter la ligne à la table mémoire
      Loop While readRow IsNot Nothing
    End Using
    'Renvoi de la valeur de retour
    Return table
End Function
```



Attention ! Vous vous souvenez que le runtime n'apprécie pas beaucoup que vous utilisiez une référence qui ne référencie aucun objet.
Que va-t-il se passer lorsque la dernière ligne est atteinte lors du dernier passage dans la boucle ?
PAF !!!!
Une bonne pratique est de toujours tester votre référence avant de vous

lancer à l'utiliser.

- Rajoutez le test de la référence **readRow** pour le cas où la dernière ligne aura été atteinte :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
        'Définition des variables locales à la fonction
        Dim table As DataTable = Nothing
        Dim readRow As String = String.Empty
        Dim isFirstRow As Boolean = Boolean.TrueString
        'Parcours du fichier à l'aide d'un StreamReader
        Using sr As StreamReader = New StreamReader(FileName)
            'Lecture des lignes du fichier une à une jusqu'à la dernière ligne
            Do
                readRow = sr.ReadLine()
                If Not readRow Is Nothing Then
                    If isFirstRow Then
                        table = CreateDataTable()
                        isFirstRow = False
                    End If
                    'Ajouter la ligne à la table mémoire
                End If
            Loop While readRow IsNot Nothing
        End Using
        ‘Renvoi de la valeur de retour
        Return table
End Function
```

Etape 4 : La dernière étape est l'ajout de chacune des lignes à la table mémoire.

Qui dit nouvelle grande étape dit nouvelle procédure…

4. Créez une nouvelle procédure nommée par exemple AddRowToDataTable pour ajouter une ligne de fichier lue à la table mémoire.
    - Retrouvez la région **« Gestion de la table de données en mémoire »** que vous avez créée au point 1 de l'exercice 3.1 dans le fichier de code **Main.vb** :

```
#Region "Gestion de la table de données en mémoire"
    Function CreateDataTable ...
#End Region
```

Elle doit contenir la fonction **CreateDataTable** qui crée la structure de la table mémoire.

- Ajoutez à la suite de la fonction **CreateDataTable**, une nouvelle procédure nommée **AddRowToDataTable** comme suit :

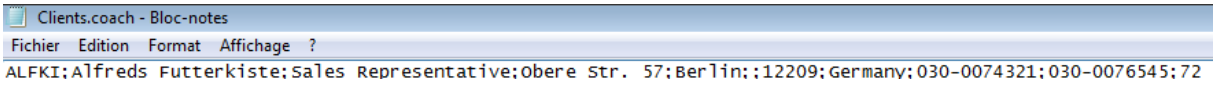
Code VB

```
#Region "Gestion de la table de données en mémoire" Function CreateDataTable() As
```

```
DataTable
…
End Function
Sub AddRowToDataTable(ByVal ReadRow As String, ByVal Table As DataTable)

End Sub
#End Region
```

Cette procédure ne retourne aucune valeur et doit récupérer en paramètres la ligne en cours lue dans le fichier ainsi que l'objet table mémoire dans lequel celle-ci doit être insérée.

Pour rappel, chaque ligne du fichier que nous avons récupérée est une chaîne de caractères comprenant un enregistrement de données où les valeurs des champs sont séparées par un point virgule.



Nous avions vu à l'atelier 3 qu'il existe de nombreuses fonctions pour traiter les chaînes de caractères, dont la fonction **Split** qui permet de découper une chaîne selon un séparateur donné et de ranger le résultat dans un tableau.

- Ajoutez le code pour découper la ligne de texte lue en tableau de chaînes de caractères en fonction du séparateur **« ; »** à l'aide de la fonction **Split** de la chaîne **ReadRow** :

Code VB

```
Sub AddRowToDataTable(ByVal ReadRow As String, ByVal Table As DataTable)
    'Récupération de toutes les valeurs de la ligne dans un tableau
    Dim readValues As String() = ReadRow.Split(SEPARATOR_SEMICOLON)
End Sub
```

Réutiliser la constante que nous avions créée dans l'atelier 3 pour définir le séparateur.

Pour ajouter une ligne à une table de type **DataTable**, vous pouvez utiliser la méthode **NewRow** qui crée un enregistrement vide. Il suffit ensuite de renseigner chaque champ avec la valeur correspondante en utilisant le format **<LigneDeLaTableMemoire>(<IndexDeLaColonne>) = <Valeur>** puis d'ajouter l'enregistrement à la collection des lignes de la table à l'aide de la méthode **Add** de la collection.

Pour comprendre le fonctionnement de la méthode **NewRow** :
http://msdn.microsoft.com/fr-fr/library/system.data.datatable.newrow.aspx

- Créez une nouvelle ligne vide dans la table mémoire à l'aide de la méthode **NewRow** de l'objet **Table** passé en paramètres de la procédure:

Code VB

```
Sub AddRowToDataTable(ByVal ReadRow As String, ByVal Table As DataTable)
    'Récupération de toutes les valeurs de la ligne dans un tableau
     Dim readValues As String() = ReadRow.Split(SEPARATOR_SEMICOLON)
     'Création d'un enregistrement vide dans la table mémoire
     Dim dataRow As DataRow = Table.NewRow()
End Sub
```

Comment est-ce qu'on peut parcourir le tableau des valeurs d'une ligne ? Il s'agit bien évidemment d'une structure de boucle. Lorsqu'on travaille sur la base d'un tableau ou d'une collection, le plus idéal est la structure **For Each…Next** que nous avons vu à l'atelier 3.

- Ajoutez le code de balayage du tableau de valeurs d'une ligne de données à l'aide de l'instruction **For Each** :

Code VB

```
Sub AddRowToDataTable(ByVal ReadRow As String, ByVal Table As DataTable)
    'Récupération de toutes les valeurs de la ligne dans un tableau
     Dim readValues As String() = ReadRow.Split(SEPARATOR_SEMICOLON)
     'Création d'un enregistrement vide dans la table mémoire
     Dim dataRow As DataRow = Table.NewRow()
     'Balayage du tableau de valeurs d'une ligne de données
     For Each Value As String In readValues

     Next
End Sub
```

Pour insérer la valeur dans la colonne correspondante, nous allons utiliser un **index** et la structure de décision multiple **Switch**.

- Ajoutez la définition d'un index en début de procédure et le code de remplissage de la ligne de données dans la table :

Code VB

```
Sub AddRowToDataTable(ByVal ReadRow As String, ByVal Table As DataTable)
     'Définition d'un index pour repérer les colonnes de valeurs
     Dim index As Integer = 0
     'Récupération de toutes les valeurs de la ligne dans un tableau
     Dim readValues As String() = ReadRow.Split(SEPARATOR_SEMICOLON)
     'Création d'un enregistrement vide dans la table mémoire
     Dim dataRow As DataRow = Table.NewRow()
     'Balayage du tableau de valeurs d'une ligne de données
     For Each Value As String In readValues
       Select Case index
         Case 0
           dataRow("Id") = Value.Trim()
         Case 1
           dataRow("Contact") = Value.Trim()
```

```
            Case 2
               dataRow("Titre") = Value.Trim()
            Case 3
               dataRow("Adresse") = Value.Trim()
            Case 4
               dataRow("Ville") = Value.Trim()
            Case 5
               dataRow("Region") = Value.Trim()
            Case 6
               dataRow("Code Postal") = Value.Trim()
            Case 7
               dataRow("Pays") = Value.Trim()
            Case 8
               dataRow("Telephone") = Value.Trim()
            Case 9
               dataRow("Telecopie") = Value.Trim()
            Case 10
               dataRow("CA") = Value.Trim()
         End Select
         index += 1
      Next
End Sub
```

A chaque valeur de la ligne on incrémente l'index pour passer à la colonne de données suivante.

Remarquez l'utilisation d'une autre fonction de traitement de chaîne de caractères, **Trim**, qui permet de supprimer tout espace blanc intempestif pouvant traîner au début et à la fin de la chaîne.

Pour en savoir plus sur la méthode **Trim** de la classe **String** :
http://msdn.microsoft.com/fr-fr/library/t97s7bs3.aspx

- Il ne vous reste plus qu'à ajouter l'enregistrement à la fin du chargement de toutes les colonnes dans la collection de lignes de données de la table **Table** :

Code VB

```
Sub AddRowToDataTable(ByVal ReadRow As String, ByVal Table As DataTable)
      'Définition d'un index pour repérer les colonnes de valeurs
      Dim index As Integer = 0
      'Récupération de toutes les valeurs de la ligne dans un tableau
      Dim readValues As String() = ReadRow.Split(SEPARATOR_SEMICOLON)
      'Création d'un enregistrement vide dans la table mémoire
      Dim dataRow As DataRow = Table.NewRow()
      'Balayage du tableau de valeurs d'une ligne de données
      For Each Value As String In readValues
         Select Case index
            …
         End Select
         index += 1
      Next
```

```
        Table.Rows.Add(dataRow)
End Sub
```

Dernière étape : Nous allons maintenant compléter chacune des procédures avec l'appel au traitement correspondant.

5. Revenez sur le code de la fonction **ReadFile** pour ajouter l'appel à la procédure **AddRowToDataTable** qui traite chaque ligne et l'ajoute à la table mémoire :

Code VB

```
Function ReadFile(ByVal FileName As String) As DataTable
    'Définition des variables locales à la fonction
    Dim table As DataTable = Nothing
    Dim readRow As String = String.Empty
    Dim isFirstRow As Boolean = Boolean.TrueString
    'Parcours du fichier à l'aide d'un StreamReader
    Using sr As StreamReader = New StreamReader(FileName)
      'Lecture des lignes du fichier une à une jusqu'à la dernière ligne
      Do
        readRow = sr.ReadLine()
        'si c'est la première ligne lue alors créer la table en mémoire
        If Not readRow Is Nothing Then
          If isFirstRow Then
            table = CreateDataTable()
            isFirstRow = False
          End If
          'Ajouter la ligne à la table mémoire
          AddRowToDataTable(readRow, table)
        End If
      Loop While readRow IsNot Nothing
    End Using
    'Renvoi de la valeur de retour
    Return table
End Function
```

6. Revenez sur le code du gestionnaire d'évènement **OuvrirToolStripMenuItem_Click** pour ajouter l'appel à la procédure **ReadFile** qui lit le fichier et renvoie la table mémoire chargée :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture de Windows
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
```

```
        'Code d'ouverture du fichier sélectionné
        Dim newDataTable As DataTable
        newDataTable = ReadFile(openDataFileDialog.FileName)
    End If
End Sub
```

N'oubliez pas de programmer la liaison de données entre la table mémoire et les contrôles d'affichage comme dans l'exercice précédent en utilisant le gestionnaire de liaison **mainBindingSource**.

- Ajoutez le code de mise en place de la liaison de données :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture de Windows
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
        'Code d'ouverture du fichier sélectionné
        Dim newDataTable As DataTable
        newDataTable = ReadFile(openDataFileDialog.FileName)
        'Configuration du gestionnaire de liaison sur la source de données
        MainBindingSource.DataSource = newDataTable
        'Liaison du gestionnaire avec les contrôles d'affichage de données
        MainDataGridView.DataSource = MainBindingSource
        MainBindingNavigator.BindingSource = MainBindingSource
    End If
End Sub
```

Une dernière petite chose avant de tester le fonctionnement de l'ouverture d'un fichier de donnée : il est fort probable que l'utilisateur veuille réenregistrer ses modifications de données dans le même fichier au moment de la sauvegarde sur disque.
Dans cette optique, il faut donc prévoir de conserver le nom du fichier à l'ouverture de celui-ci. Nous allons donc rajouter une variable dont la portée est la classe **Main** pour y stocker le chemin complet du fichier.

- Ajouter la définition d'une variable **DataFilePath** de type **String** dans les déclarations de variable privée de la classe **Main** :

Code VB

```
Public Class Main

#Region "Déclaration des variables privées de la classe"
    'Déclaration des variables de fichier
    Dim DataFilePath As String = String.Empty
```

```
    'Déclaration des propriétés de la boîte des options
    Dim RecentFileListNumber As Short = 0
    Dim ConfirmBeforeSave As Boolean = Boolean.FalseString
    Dim SaveDirectoryPath As String = String.Empty
    Dim TraceEnabled As Boolean = Boolean.FalseString
    Dim SaveDirectoryType As DirectoryType = DirectoryType.MyDocuments
    Dim AuthorInfo() As String
    'Déclaration des énumérations
    Enum DirectoryType
        MyDocuments = 0
        Other = 1
    End Enum
    'Déclaration des constantes
    Const SEPARATOR_SEMICOLON As String = ";"
#End Region
…
End Class
```

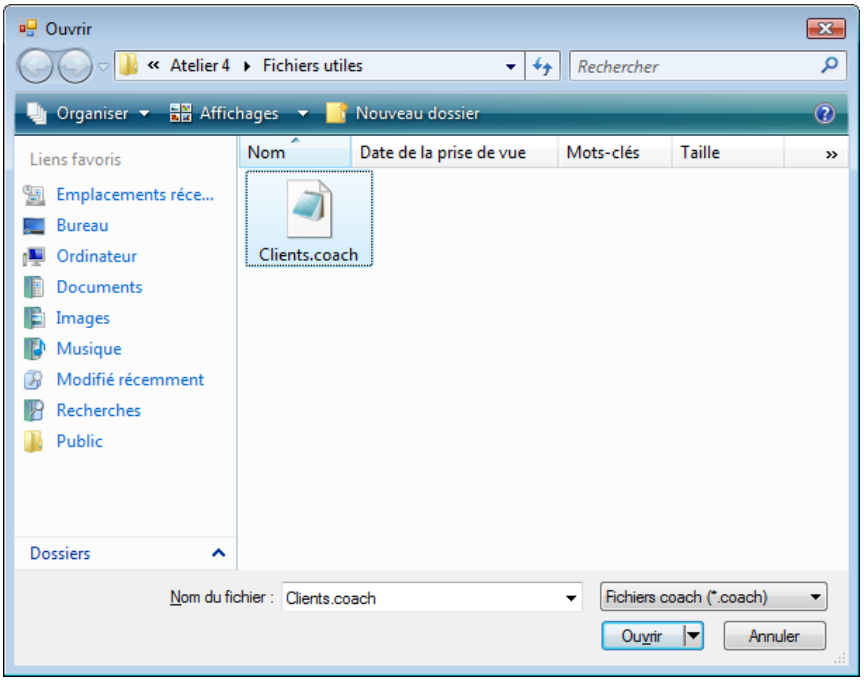
- Sauvegardez le chemin du fichier sélectionné par l'utilisateur, donné par la propriété **FileName** de la boîte de dialogue **openDataFileDialog**, dans la variable **DataFilePath** à la suite du gestionnaire d'évènement **OuvrirToolStripMenuItem_Click** :

Code VB

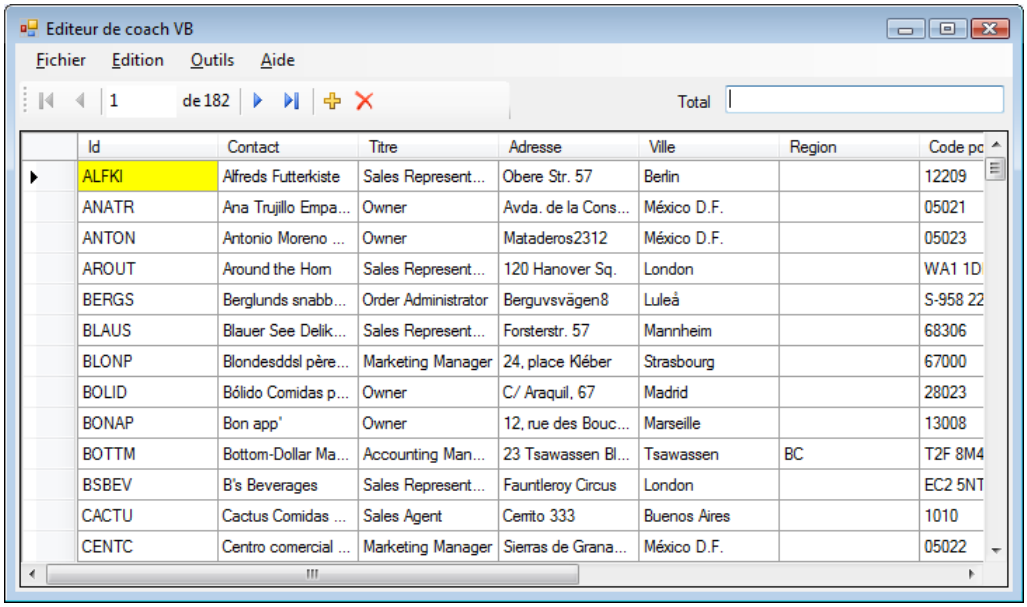
```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture de Windows
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
        'Code d'ouverture du fichier sélectionné
        Dim newDataTable As DataTable
        newDataTable = ReadFile(openDataFileDialog.FileName)
        'Configuration du gestionnaire de liaison sur la source de données
        MainBindingSource.DataSource = newDataTable
        'Liaison du gestionnaire avec les contrôles d'affichage de données
        MainDataGridView.DataSource = MainBindingSource
        MainBindingNavigator.BindingSource = MainBindingSource
        'Mémorisation du chemin du fichier pour une éventuelle sauvegarde
        DataFilePath = openDataFileDialog.FileName
    End If
End Sub
```

7. Testez le fonctionnement de l'ouverture d'un fichier de données d'extension *.coach :
    - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.

- Exécutez l'application en cliquant sur (ou touche F5).
- Cliquez le menu **Fichier** > **Ouvrir**.
- Sélectionnez le fichier **Clients.coach** fourni dans le dossier **…Atelier 4\Fichiers Utiles** de cet atelier.



- Cliquez **Ouvrir**. Vous devez obtenir la grille chargée avec le jeu de données contenu dans le fichier proposé.



Id	Contact	Titre	Adresse	Ville	Region	Code pc
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 Hom	Sales Represent...	120 Hanover Sq.	London		WA1 1D
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	Bólido 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

- Quitter l'application.

Bravo ! Encore un petit exercice pour apprendre à faire le processus inverse de sauvegarde des données de la table mémoire sur disque et l'éditeur fonctionne !

Une dernière bricole…
Et si vous affichiez le nom du fichier dans l'intitulé de la fenêtre de l'application (un peu comme le fait Word) ?
Pour cela, vous pouvez utiliser une classe de l'espace de noms **System.IO** très utile appelée **FileInfo** qui, comme son nom l'indique, permet de gérer des informations de fichier (dont son nom qui nous intéresse ici, le chemin du répertoire dans lequel se trouve le fichier, le chemin complet etc…). C'est une classe qui contient également des méthodes intéressantes pour copier, supprimer, déplacer un fichier etc…

- A la suite du gestionnaire d'évènement **OuvrirToolStripMenuItem_Click**, ajoutez le code de définition d'un objet de type **FileInfo** sur la base du chemin enregistré dans **DataFilePath** :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture de Windows
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
        …
        'Mémorisation du chemin du fichier pour une éventuelle sauvegarde
        DataFilePath = openDataFileDialog.FileName
        'Affichage du nom du fichier dans la barre de titre du formulaire
        Dim fileInformation As IO.FileInfo = New IO.FileInfo(DataFilePath)
    End If
End Sub
```

- Utiliser la méthode **Concat** de la classe **String** pour concaténer le nom du fichier avec le titre initialement configuré pour le formulaire :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture de Windows
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
        …
        'Mémorisation du chemin du fichier pour une éventuelle sauvegarde
        DataFilePath = openDataFileDialog.FileName
        'Affichage du nom du fichier dans la barre de titre du formulaire
        Dim fileInformation As IO.FileInfo = New IO.FileInfo(DataFilePath)
```

```
        Me.Text = String.Concat("Editeur de coach VB", _
                                " - ", fileInformation.Name)
    End If
End Sub
```

Comme nous allons certainement avoir besoin de mettre à jour ce titre ultérieurement, par exemple lorsque l'utilisateur décide d'enregistrer ses données sous un nouveau nom de fichier, nous pouvons extraire ce code et le factoriser dans une procédure séparée.

- Ajoutez une nouvelle méthode **UpdateFormTitle** à la classe **Main** comme suit :

Code VB

```
Private Sub UpdateFormTitle(ByVal FilePath As String)
        Dim fileInformation As IO.FileInfo = New IO.FileInfo(FilePath)
        Me.Text = String.Concat("Editeur de coach VB", _
                                " - ", fileInformation.Name)
End Sub
```

Attention ! Le constructeur de l'objet **FileInfo** ne traite pas le cas d'une chaîne vide en paramètre. Or nous allons en avoir besoin pour traiter le cas d'un nouveau fichier pour lequel la barre de titre ne comprend pas de nom de fichier.

- Ajoutez un test pour le cas où le chemin de fichier n'est pas connu en utilisant la méthode **IsNullOrEmpty** de la classe **String** :

Code VB

```
Private Sub UpdateFormTitle(ByVal FilePath As String)
    If String.IsNullOrEmpty(FilePath) Then
        Me.Text = String.Concat("Editeur de coach VB")
    Else
        Dim fileInformation As IO.FileInfo = New IO.FileInfo(FilePath)
        Me.Text = String.Concat("Editeur de coach VB", _
                                " - ", fileInformation.Name)
    End If
End Sub
```

- Remplacez les lignes de la procédure **OuvrirToolStripMenuItem_Click** avec l'appel à la méthode **UpdateFormTitle** :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture de Windows
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
```

```
        openDataFileDialog.Filter = "Fichiers coach|*.coach"
        openDataFileDialog.InitialDirectory = SaveDirectoryPath
        If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
            …
            'Mémorisation du chemin du fichier pour une éventuelle sauvegarde
            DataFilePath = openDataFileDialog.FileName
            'Affichage du nom du fichier dans la barre de titre du formulaire
            Dim fileInformation As IO.FileInfo = New IO.FileInfo(DataFilePath)
            Me.Text = String.Concat(Me.Text, " - ", fileInformation.Name)
            UpdateFormTitle(DataFilePath)
        End If
    End Sub
```

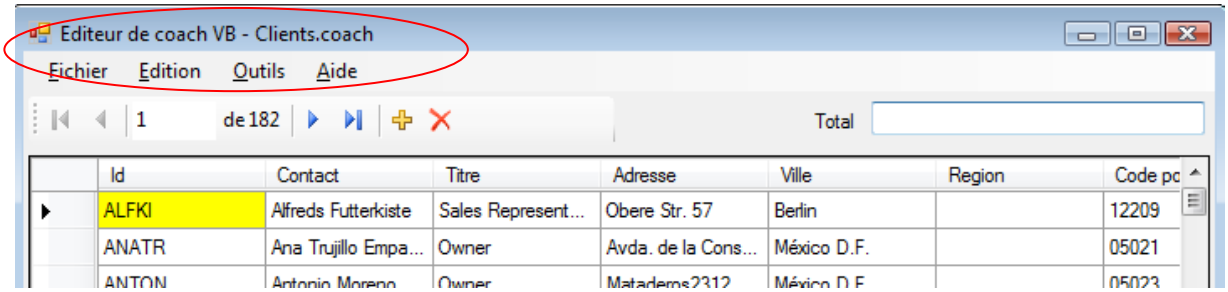
Profitons en pour tout de suite remettre à jour le titre du formulaire lorsque l'utilisateur créé un nouveau fichier pour lequel il n'existe encore aucun nom :

- Retrouvez la procédure **NouveauToolStripMenuItem_Click.**
- Ajoutez la mise à jour du chemin du fichier et du titre du formulaire comme suit :

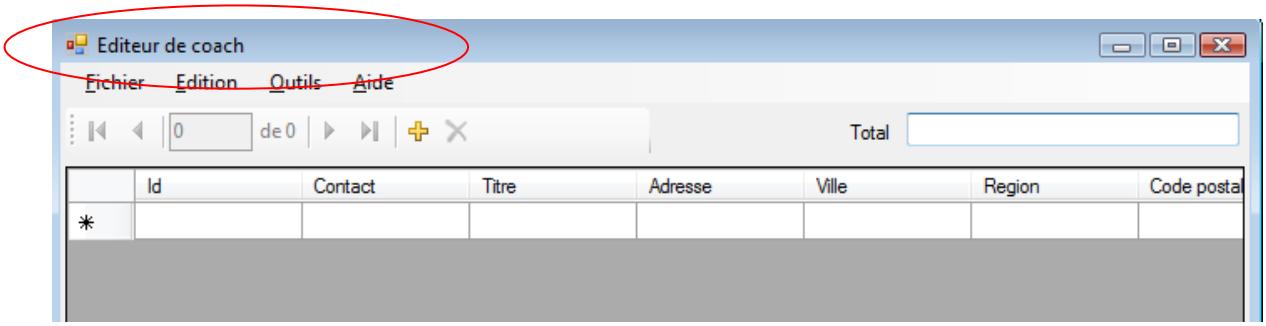
Code VB

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click
    'Création d'une nouvelle table mémoire vide
    Dim newDataTable As DataTable
    newDataTable = CreateDataTable()
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.BindingSource = MainBindingSource
    'Mettre à jour le chemin de fichier à vide
    DataFilePath = String.Empty
    'Remettre à jour le titre du formulaire
    UpdateFormTitle(DataFilePath)
  End Sub
```

8. Testez l'affichage du titre :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir**.
   - Sélectionnez le fichier **Clients.coach** fourni dans le dossier **…Atelier 4\Fichiers Utiles** de cet atelier.
   - Cliquez **Ouvrir**. Vérifiez que le nom du fichier est concaténé au titre du formulaire :



- Cliquez le menu **Fichier** > **Nouveau**. Vérifiez que la barre de titre est remise à jour :

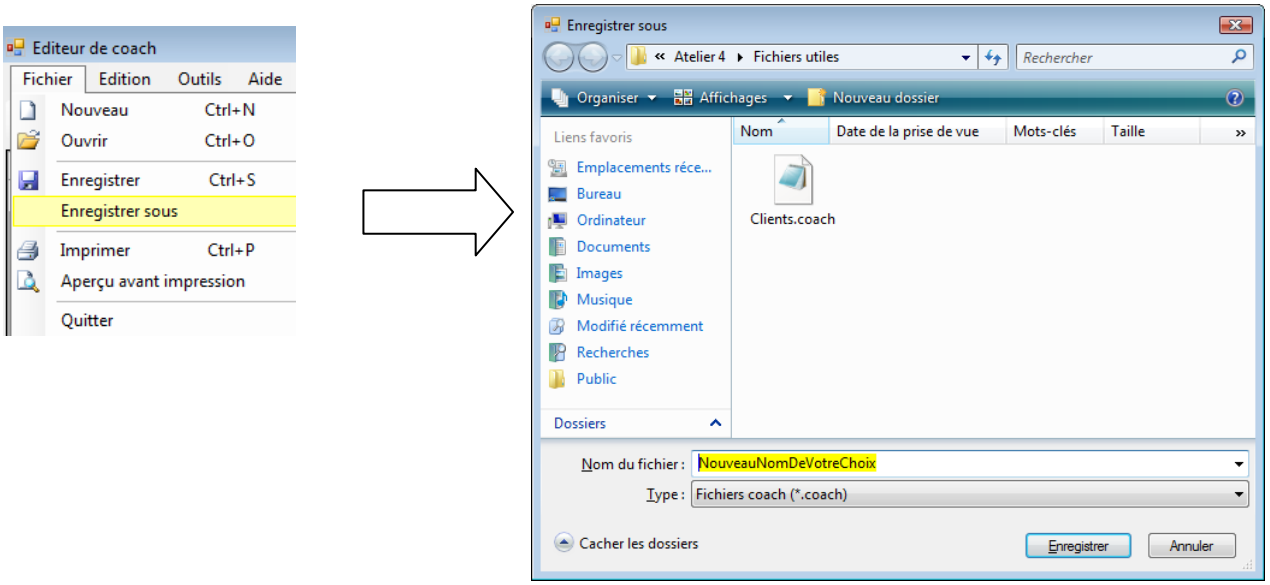


- Quitter l'application.

3.3.2 Ecrire dans le fichier

Contexte fonctionnel
L'application comporte les deux menus traditionnels **Enregistrer** et **Enregistrer sous** pour enregistrer les données :

- si l'utilisateur sélectionne le menu **Fichier** > **Enregistrer sous**, la boîte de dialogue standard **Enregistrer sous** de Windows apparaît pour qu'il choisisse un nom de fichier et le chemin de sauvegarde.



- si l'utilisateur sélectionne le menu **Fichier** > **Enregistrer**.

  Soit le nom du fichier est connu : par exemple parce que l'utilisateur travaille sur un fichier existant. Dans ce cas, la sauvegarde est réalisé sur le nom du fichier existant.

  Soit il ne l'est pas : par exemple pour un nouveau fichier. Dans ce deuxième cas, la boîte de dialogue **Enregistrer sous** de Windows doit s'afficher pour que l'utilisateur puisse déterminer le nom et le chemin du fichier de sauvegarde (on retombe sur le premier scénario).

L'algorithme à développer pour écrire dans un fichier est composé de trois étapes :

1. Sélectionner le chemin de sauvegarde et le nom du fichier (uniquement dans le cas de l'option de menu **Enregistrer sous**),
2. Balayer les enregistrements de la table mémoire un à un,
3. Ecrire chaque enregistrement sur une ligne de fichier.

Etape 1 : La première étape consiste donc à écrire le code de sélection du chemin et du fichier sur le disque.

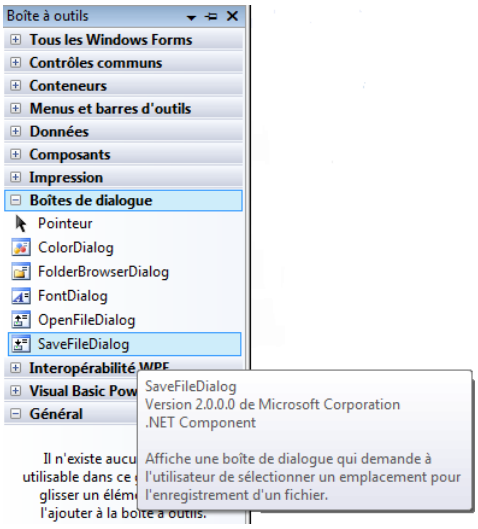
Cette étape concerne l'option de menu **Fichier** > **Enregistrer sous** du formulaire **Main**.

1. Créez un gestionnaire d'évènement associé au clic du menu Enregistrer sous :
   - Affichez le formulaire **Main** en mode **Design**.
   - Sélectionnez le menu **Fichier** puis double cliquez sur l'option de menu **Enregistrer sous** pour générer la procédure **EnregistrersousToolStripMenuItem_Click** :

```
247     Private Sub EnregistrersousToolStripMenuItem_Click(ByVal sender As System.Object, _
248                                                        ByVal e As System.EventArgs) _
249                                                        Handles EnregistrersousToolStripMenuItem.Click
250
251     End Sub
252 End Class
```

2. Utilisez la boîte de dialogue standard de sauvegarde de fichier de Windows pour permettre à l'utilisateur de faire sa sélection :

La boîte de dialogue qui nous intéresse cette fois-ci est **SaveFileDialog**. Elle est utilisable comme précédemment à partir de la Boîte à outils ou directement via la classe correspondante du Framework .NET.



- Ajoutez à la procédure le code de déclaration et d'instanciation d'un nouvel objet de type **SaveFileDialog** :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog As SaveFileDialog = New SaveFileDialog()
End Sub
```

Les principales propriétés à configurer sur cet objet sont les mêmes que celles de la boîte **OpenFileDialog**, à savoir : **Filter** et **InitialDirectory**.

- Complétez le code pour filtrer le type de fichier sur l'extension ***.coach** :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog As SaveFileDialog = New SaveFileDialog()
    saveDataFileDialog.Filter = "Fichiers coach|*.coach"
End Sub
```

- Configurez le répertoire par défaut sur le dossier préférentiel de l'utilisateur configuré à l'aide de la fenêtre **Options** à l'atelier 3 dans la variable privée **SaveDirectoryPath** de la classe **Main** :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog As SaveFileDialog = New SaveFileDialog()
    saveDataFileDialog.Filter = "Fichiers coach|*.coach"
```

```
        saveDataFileDialog.InitialDirectory = SaveDirectoryPath
End Sub
```

Pour éviter de taper à chaque ligne le nom relativement long de notre objet de type **SaveFileDialog**, une bonne pratique est d'utiliser la structure **With…End With** que nous avons déjà rencontrée dans ce tutorial.

- Ajoutez un bloc **With…End With** pour simplifier l'écriture du code :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                    ByVal sender As System.Object, _
                                    ByVal e As System.EventArgs) _
                                    Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog As SaveFileDialog = New SaveFileDialog()
    With saveDataFileDialog
        saveDataFileDialog.Filter = "Fichiers coach|*.coach"
        saveDataFileDialog.InitialDirectory = SaveDirectoryPath
    End With

  End Sub
End Class
```

Avec la toute dernière version du langage Visual Basic fournie avec Visual Studio 2008, il existe même une écriture encore plus concise qui permet de déclarer, instancier et initialiser les propriétés de l'objet en une seule instruction.

- Retransformez le code de la façon suivante :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                    ByVal sender As System.Object, _
                                    ByVal e As System.EventArgs) _
                                    Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog = New SaveFileDialog() With _
                {.Filter = "Fichiers coach|*.coach", _
                 .InitialDirectory = SaveDirectoryPath}
  End Sub
End Class
```

Deux remarques :

- Le mot clé **As** qui indique le type de données de la variable a disparu ! Comment le système va-t-il connaître le type de l'objet si on omet de le préciser ?
  C'est une nouveauté du langage ☺. On parle d'**Inférence de type avec les types nommés**. Le type de donnée de la variable est déduit du type de l'objet créé par l'assignation (c'est-à-dire par ce qui se trouve à droite du signe **=**. Ici, il s'agit du constructeur de la classe).

- Pour cette nouvelle écriture plus concise utilisant le mot clé **With**, on parle d'**Initialiseur d'objet**.

Pour tout savoir sur les **initialiseurs d'objets** de Visual Basic 2008: http://msdn.microsoft.com/fr-fr/library/bb385125.aspx

- Affichez maintenant la boîte de dialogue avec la méthode **ShowDialog** et testez le code de fermeture avec une condition **If…Then…End If** :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog = New SaveFileDialog() With _
                                        {.Filter = "Fichiers coach|*.coach", _
                                         .InitialDirectory = SaveDirectoryPath}
    If saveDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then

    End If
End Sub
```

- Récupérez le chemin du fichier à partir de la propriété **FileName** de l'objet **saveDataFileDialog** et stockez-le dans la variable **DataFilePath** :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog = New SaveFileDialog() With _
                                        {.Filter = "Fichiers coach|*.coach", _
                                         .InitialDirectory = SaveDirectoryPath}
    If saveDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
        'Sauvegarde du chemin du fichier sélectionné
        DataFilePath = saveDataFileDialog.FileName
    End If
End Sub
```

- Mettez à jour le nouveau nom de fichier dans l'intitulé du formulaire à l'aide de la méthode **UpdateFormTitle** créé à l'exercice précédent :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog = New SaveFileDialog() With _
                                        {.Filter = "Fichiers coach|*.coach", _
                                         .InitialDirectory = SaveDirectoryPath}
    If saveDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
```

```
            'Sauvegarde du chemin du fichier sélectionné
            DataFilePath = saveDataFileDialog.FileName
            'Mise-à-jour du titre du formulaire avec le nouveau nom de fichier
            UpdateFormTitle(DataFilePath)
        End If
End Sub
```

Etapes 2 et 3 : Les deux autres étapes consistent à sauvegarder les données de la grille dans le fichier.

3. Ajoutez une méthode à la classe Main dont l'objectif est de sauvegarder les données dans un fichier :
   - Retrouvez la région de code **Traitement des fichiers de données** contenant déjà la fonction **ReadFile** :

```
#Region "Traitement des fichiers de données"
    Function ReadFile ...
#End Region
```

   - Juste après la fonction **ReadFile**, ajoutez la définition d'une nouvelle procédure **WriteFile** comme suit :

Code VB

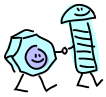
```
Sub WriteFile(ByVal FileName As String)
End Sub
```

Notez que la procédure a évidemment besoin du chemin du fichier qu'il lui faudra ouvrir. C'est l'objet du paramètre **FileName**.
En revanche, nous n'avons pas besoin de valeur de retour ici.

Pour lire le fichier, nous avons eu besoin d'un objet **StreamReader** qui faisait office de *lecteur*. Pour écrire dans le fichier nous allons avoir besoin à l'inverse d'un *écrivain*, c'est-à-dire un objet ... ? **StreamWriter** bien sûr !

C'est quoi un **StreamWriter** ?
C'est un type d'objet qui *écrit* des caractères à partir d'un flux d'octets dans un codage particulier. Tout comme le **StreamReader**, il appartient à l'espace de nommage **System.IO**. La méthode que nous allons utilisée sur cet objet pour écrire une ligne de données texte est **WriteLine** (on s'en serait douté puisque c'était **ReadLine** pour l'opération inverse avec le **StreamReader**).

Pour en savoir plus sur la classe **StreamWriter** :
http://msdn.microsoft.com/fr-fr/library/system.io.streamwriter.aspx



N'oublions pas la structure de contrôle **Using…End Using** pour maitriser la durée de vie de notre objet !

- Ajoutez la déclaration d'un objet de type **StreamWriter** comme suit :

Code VB

```
Sub WriteFile(ByVal FileName As String)
    Using sw = New System.IO.StreamWriter(FileName, False)

    End Using
End Sub
```

Plus besoin de s'embêter avec la clause **As** ☺ !

Maintenant il suffit de récupérer les données et de les charger enregistrement par enregistrement dans le fichier.

Mais où sont les données au fait ?
Tout simplement dans la propriété **DataSouce** de l'objet gestionnaire de liaison **mainBindingSource**. C'est lui qui connait la source de données.

En effet, aussi bien lors de la création d'un nouveau fichier que lors de l'ouverture d'un fichier existant, nous avons écrit respectivement les lignes :

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click
    'Création d'une nouvelle table mémoire vide
    Dim newDataTable As DataTable
    newDataTable = CreateDataTable()
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
```

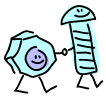
Création d'une table vide

Et :

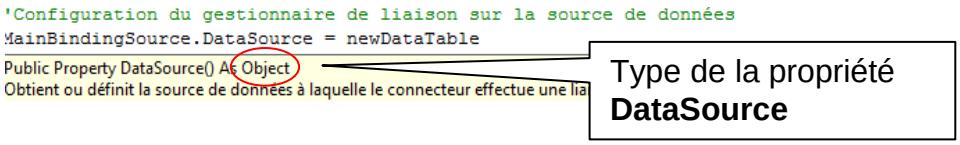
```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture standard de Windows
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPa
    If openDataFileDialog.ShowDialog = Windows.Forms.Dial
        'Code d'ouverture du fichier sélectionné
        Dim newDataTable As DataTable
        newDataTable = ReadFile(openDataFileDialog.FileName)
        'Configuration du gestionnaire de liaison sur la source de données
        MainBindingSource.DataSource = newDataTable
```

Création d'une table à partir d'un fichier existant

DataSource contient donc un objet de type **DataTable**, qui a une propriété **Rows** pointant sur la collection de ligne de la table de données. C'est ce qu'il nous faut sauvegarder !



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



DataSource est donc de type **Object**.

Que représente le type de données **Object** ?
Il s'agit du type de données universel dans le Framework .NET et VB, ce qui signifie que tous les autres types de données sont dérivés de lui. Il est donc très utile pour typer une variable destinée à recevoir des données de différents types.

La propriété **DataSource** est de type **Object** car elle est destinée à configurer ou obtenir non seulement des tables de données (**DataTable**), mais aussi tout objet pouvant fournir des données (par exemple un **DataSet** que nous aurons l'occasion d'utiliser dans le prochain atelier). Donc le Framework ne sait pas à l'avance le type de données qu'il devra stocker.

Pour tout savoir sur le type **Object** :
http://msdn.microsoft.com/fr-fr/library/3cf69sdx.aspx

- Ajouter le code de balayage de la collection **Rows** des lignes de la table **DataTable** stockée dans la propriété **DataSource** à l'aide d'une structure de bouclage **For Each**:

Code VB

```
Sub WriteFile(ByVal FileName As String)
    Using sw = New System.IO.StreamWriter(FileName, False)
        For Each Row In MainBindingSource.DataSource.Rows

        Next
    End Using
End Sub
```

Avez-vous remarqué que l'on a omis de définir le type de la variable **Row** avec le mot clé **As** ?
C'est toujours cette même fonctionnalité d'**inférence de type** du langage que l'on utilise. Le compilateur va déduire le type de la variable **Row** à partir du type de la collection d'objets définie après **In**.

Pour tout savoir sur l'**inférence de type local** :
http://msdn.microsoft.com/fr-fr/library/bb384937.aspx

Avez-vous remarqué également que l'IntelliSense de Visual Studio ne vous propose pas la propriété **Rows** dans la liste des membres de l'objet **DataSource** ?

```
Sub WriteFile(ByVal FileName As String)
    Using sw = New System.IO.StreamWriter(FileName, False)
        For Each Row In MainBindingSource.DataSource.

        Next
    End Using
End Sub
d Region
```

Equals
GetHashCode
GetType
ReferenceEquals
ToString

En même temps, pourquoi le ferait-elle puisque Visual Studio ne connaît pas encore le type d'objet stocké dans **DataSource** ? Seul le type **DataTable** contient un membre **Rows**, pas le type **Object**…

Mais alors comment indiquer qu'un objet a un type précis ?
Visual Basic propose des instructions et des méthodes de conversion d'un type en un autre. Nous en avons vu quelques unes dans l'atelier 3 précédent. Si la conversion est possible, alors la valeur retournée est du type attendu.

Comment indiquer une conversion (explicite) dans le cas d'un *objet* ?
La fonction **CType** de Visual Basic effectue la conversion d'un type objet en un autre type de données. Il suffit de lui indiquer l'objet à convertir et le type de données cible en paramètres, et elle renvoie la donnée convertie.

Pour convertir un objet en un autre type dans Visual Basic :
http://msdn.microsoft.com/fr-fr/library/x53d8wxa.aspx
Pour vous documenter sur la conversion de types de données en Visual Basic :
http://msdn.microsoft.com/fr-fr/library/hcb26cc8.aspx

- Ajoutez la conversion de la propriété **DataSource** de type **Object** en type **DataTable** à l'aide de la fonction **CType**, avant de procéder au balayage de la collection **Rows** :

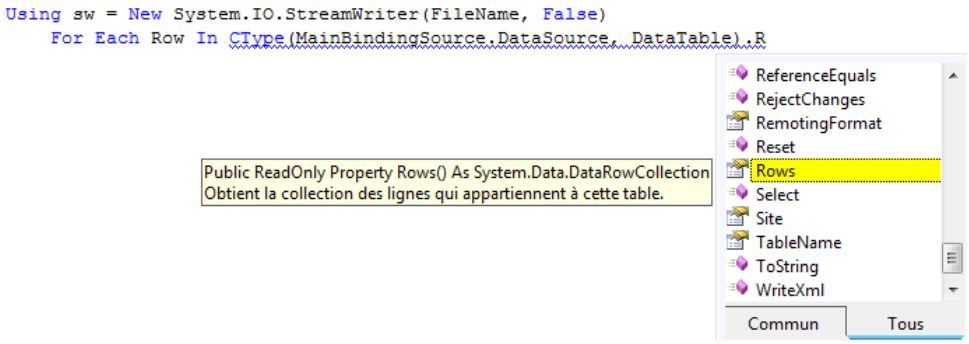
Code VB

```
Sub WriteFile(ByVal FileName As String)
    Using sw = New System.IO.StreamWriter(FileName, False)
        For Each Row In CType(MainBindingSource.DataSource,DataTable).Rows

        Next
    End Using
End Sub
```

Elément à convertir

Type de données ciblé

L'IntelliSense s'y retrouve tout de suite beaucoup mieux lorsque vous cherchez à utiliser une propriété **Rows** sur l'objet retourné par la fonction **CType** !



En même temps, vous n'étiez pas obligé de faire une conversion. Se produit alors ce qu'on appelle une *liaison tardive* c'est-à-dire que le typage se produit au moment de l'exécution. Le processus est évidemment dangereux et plus couteux.

Pour comparer **la liaison tardive** à **la liaison anticipée** :
http://msdn.microsoft.com/fr-fr/library/0tcf61s1.aspx

Pour construire une ligne du fichier à partir de la ligne de données de la table mémoire, il faut concaténer à la suite les valeurs de chaque colonne de la ligne séparées par le séparateur (point virgule).

Comment allons-nous accéder à la valeur des colonnes d'une ligne de la table de données ?
La propriété **Rows** du type **DataTable** est une collection de ligne de données (**DataRow**). Pour accéder à la valeur d'une colonne (champ) de la ligne, il faut indiquer entre parenthèses, le nom du champ de données voulu. Par exemple, **Row("Id")** renvoie la valeur de l'**Id** de la ligne courante.

- Utilisez la méthode **WriteLine** de l'objet **sw** pour écrire un enregistrement de données dans le fichier. Utilisez la méthode **Concat** de la classe **String** pour concaténer chacune des colonnes de la ligne séparées par le point virgule donné par la constante **SEPARATOR_SEMICOLON** :

Code VB

```
Sub WriteFile(ByVal FileName As String)
    Using sw = New System.IO.StreamWriter(FileName, False)
        For Each Row In CType(MainBindingSource.DataSource,DataTable).Rows
            sw.WriteLine(String.Concat( _
                Row("Id"), SEPARATOR_SEMICOLON _
                , Row("Contact"), SEPARATOR_SEMICOLON _
                , Row("Titre"), SEPARATOR_SEMICOLON _
                , Row("Adresse"), SEPARATOR_SEMICOLON _
                , Row("Ville"), SEPARATOR_SEMICOLON _
                , Row("Region"), SEPARATOR_SEMICOLON _
                , Row("Code Postal"), SEPARATOR_SEMICOLON _
                , Row("Pays"), SEPARATOR_SEMICOLON _
```

```
                , Row("Telephone"), SEPARATOR_SEMICOLON _
                , Row("Telecopie"), SEPARATOR_SEMICOLON _
                , Row("CA")))
        Next
    End Using
End Sub
```

Cette fois, on tient le bon bout !
Il reste une toute petite précaution à prendre avec le type de la colonne **CA** qui est **Integer** (alors que toutes les autres colonnes sont de type **String**).
En effet, si la valeur du CA est nulle dans la grille de données, c'est-à-dire qu'aucune valeur n'a été saisie par l'utilisateur, la donnée insérée dans le fichier est une chaîne vide.
Rien de bien gênant à l'écriture dans le fichier, sauf qu'à la relecture du fichier dans notre Editeur, les choses se gâtent et (aucune conversion n'ayant été prévu dans le sens inverse) on récupère une erreur d'exécution.

Comment repère-t-on une donnée de valeur nulle ?
Grâce au Framework .NET (comme d'hab), qui fournit une classe appelée **DBNull** dans l'espace de noms **System** pour indiquer l'absence de valeur d'une information d'une source de données.

- Modifiez le code pour traiter le cas où aucune valeur n'a été saisi dans la colonne de données chiffre d'affaires (CA) :

Code VB

```
Sub WriteFile(ByVal FileName As String)
    Using sw = New System.IO.StreamWriter(FileName, False)
        For Each Row In CType(MainBindingSource.DataSource,DataTable).Rows
            Dim chiffreAffaire As String
            If Row("CA") Is System.DBNull.Value Then
                chiffreAffaire = "0"
            Else
                chiffreAffaire = Row("CA").ToString()
            End If
            sw.WriteLine(String.Concat( _
                    Row("Id"), SEPARATOR_SEMICOLON _
                    , Row("Contact"), SEPARATOR_SEMICOLON _
                    , Row("Titre"), SEPARATOR_SEMICOLON _
                    , Row("Adresse"), SEPARATOR_SEMICOLON _
                    , Row("Ville"), SEPARATOR_SEMICOLON _
                    , Row("Region"), SEPARATOR_SEMICOLON _
                    , Row("Code Postal"), SEPARATOR_SEMICOLON _
                    , Row("Pays"), SEPARATOR_SEMICOLON _
                    , Row("Telephone"), SEPARATOR_SEMICOLON _
                    , Row("Telecopie"), SEPARATOR_SEMICOLON _
                    , chiffreAffaire))
```

```
        Next
    End Using
End Sub
```

Dernière étape : Nous allons maintenant compléter le gestionnaire d'évènement de l'option **Enregistrer sous** avec l'appel au traitement correspondant.

4. Revenez sur le code de la procédure **EnregistrersousToolStripMenuItem_Click** pour ajouter l'appel à la procédure **WriteFile** qui traite chaque ligne de la table mémoire et l'ajoute au fichier :

Code VB

```
Private Sub EnregistrersousToolStripMenuItem_Click( _
                                    ByVal sender As System.Object, _
                                    ByVal e As System.EventArgs) _
                                    Handles EnregistrersousToolStripMenuItem.Click
    Dim saveDataFileDialog = New SaveFileDialog() With _
                                        {.Filter = "Fichiers coach|*.coach", _
                                         .InitialDirectory = SaveDirectoryPath}
    If saveDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
        'Sauvegarde du chemin du fichier sélectionné par l'utilisateur
        DataFilePath = saveDataFileDialog.FileName
        'Mise-à-jour du titre du formulaire avec le nouveau nom de fichier
        UpdateFormTitle(DataFilePath)
        'Ecriture des données dans le fichier
        WriteFile(DataFilePath)
    End If
End Sub
```

Quel est l'algorithme derrière l'autre option de menu **Fichier** > **Enregistrer** ?
Rigoureusement le même que celui que nous venons de suivre, à ceci près qu'il faut vérifier qu'il ne s'agit pas d'un nouveau fichier auquel cas vous ne connaissez pas le nom ni le chemin du fichier de sauvegarde. Mais qu'à cela ne tienne, il suffit alors de re router l'utilisateur sur la procédure **Enregistrer Sous** précédente.

5. Coder l'option de menu Enregistrer :
   - Affichez le formulaire **Main** en mode **Design**.
   - Sélectionnez le menu **Fichier** puis double cliquez sur l'option de menu **Enregistrer** pour générer la procédure **EnregistrerToolStripMenuItem_Click** :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( _
                                    ByVal sender As System.Object, _
                                    ByVal e As System.EventArgs) _
                                    Handles EnregistrerToolStripMenuItem.Click

End Sub
```

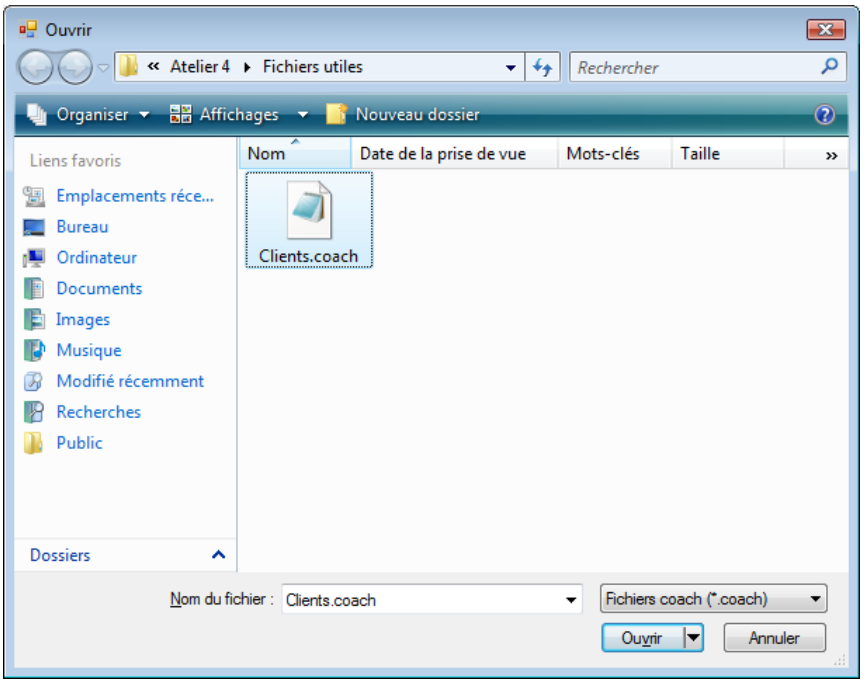
- Ajoutez le test du nom de fichier. S'il est vide, invoquer le gestionnaire d'évènement **EnregistrerSousToolStripMenuItem_Click** pour exécuter la procédure précédente :

Code VB

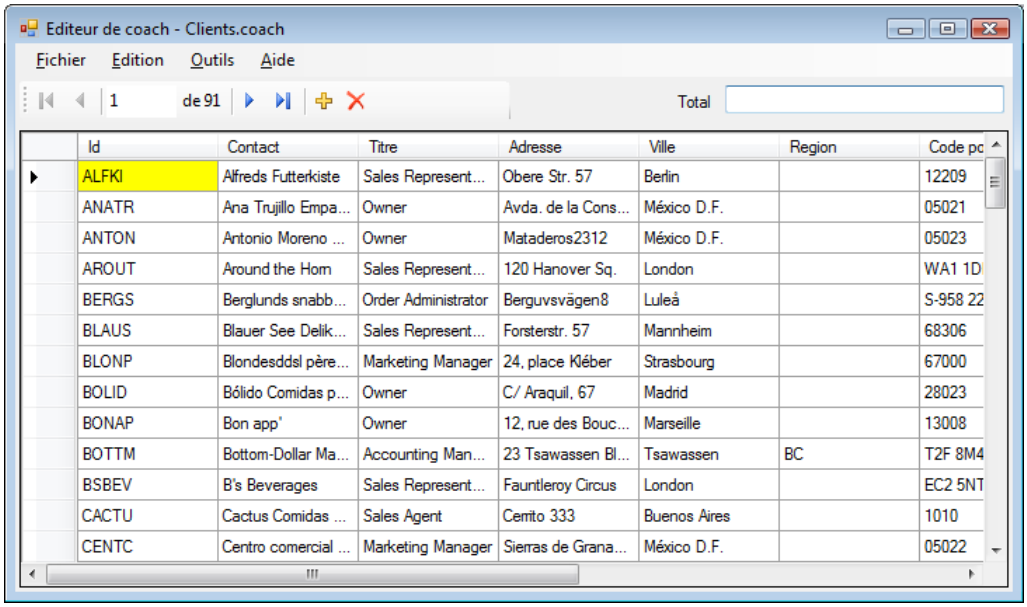
```
Private Sub EnregistrerToolStripMenuItem_Click( _
                                    ByVal sender As System.Object, _
                                    ByVal e As System.EventArgs) _
                                    Handles EnregistrerToolStripMenuItem.Click
    If String.IsNullOrEmpty(DataFilePath) Then
       'Il s'agit d'un nouveau fichier. Déclencher la proc. Enregistrer Sous
       EnregistrersousToolStripMenuItem_Click(sender, e)
    Else
       'Le nom du fichier est connu. Sauvegarder les donne
       WriteFile(DataFilePath)
    End If
End Sub
```

Retransmettez au gestionnaire d'évènement les mêmes paramètres que le gestionnaire en cours.

6. Testez le fonctionnement de l'enregistrement des données dans un fichier de données d'extension *.coach :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir**.
   - Sélectionnez le fichier **Clients.coach** fourni dans le dossier **…Atelier 4\Fichiers Utiles** de cet atelier.



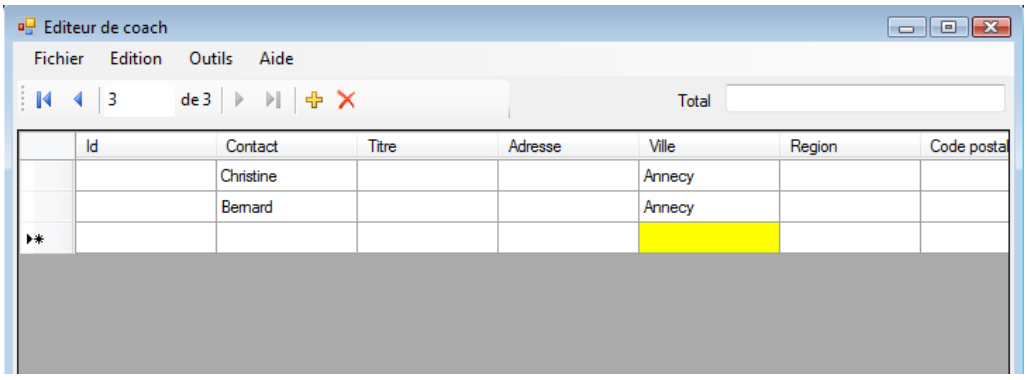
- Cliquez **Ouvrir**.



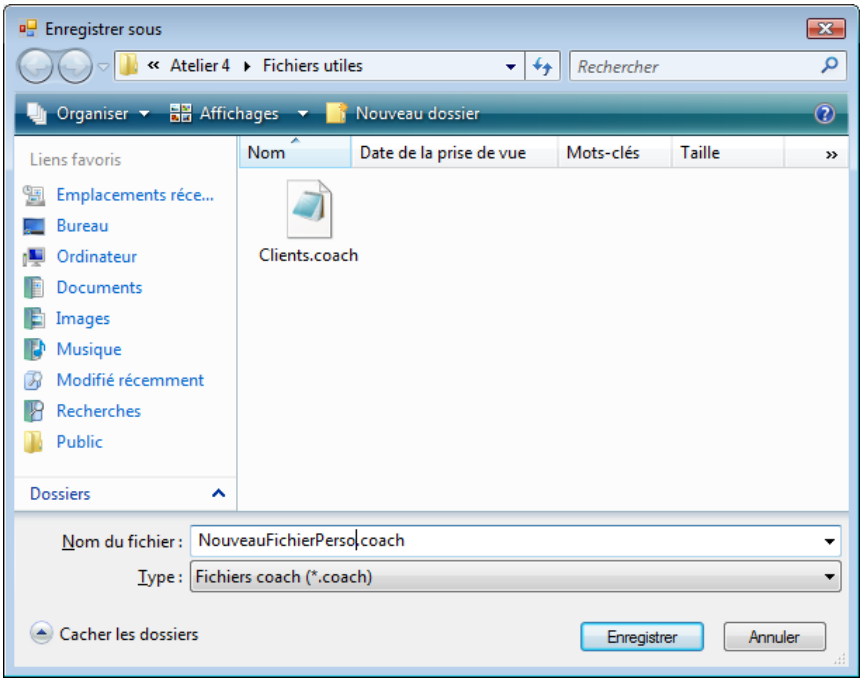
- Modifiez une ou plusieurs données dans la grille de données.
- Cliquez le menu **Fichier** > **Enregistrer**.
- Cliquez ensuite **Fichier** > **Nouveau** puis ré ouvrez le même fichier en cliquant le menu **Fichier** > **Ouvrir** comme précédemment.

Vérifiez que le fichier s'ouvre avec les valeurs enregistrées précédemment !

- Créez un nouveau fichier en cliquant **Fichier** > **Nouveau**.
- Saisissez une ou deux lignes de données.



- Cliquez **Fichier** > **Enregistrer**. Comme il s'agit d'un nouveau fichier, la boîte **Enregistrer sous** doit apparaître :



- Enregistrer le fichier avec un nouveau nom puis ré ouvrez-le pour vérifier que la sauvegarde s'est bien passée.
- Quitter l'application.

msdn

Comme pour la lecture, vous pouvez aussi utiliser les extraits de code et travailler sur la base de l'objet **My.Computer.FileSystem**.
Pour vous documenter sur l'écriture dans un fichier en Visual Basic en utilisant l'objet **My.Computer.FileSystem** :
http://msdn.microsoft.com/fr-fr/library/c520bdhb.aspx

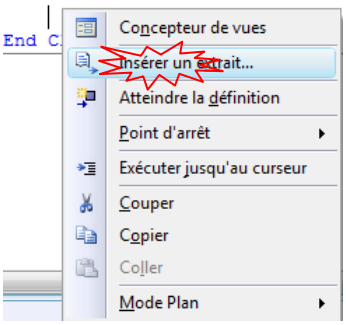
4 Pour aller plus loin…

4.1 L'objet My.Computer.FileSystem

Vous vous souvenez des objets **My.Application**, **My.Computer**, **My.Log** etc. de Visual Basic que nous avons déjà rencontrés à plusieurs reprises ?
Il se trouve que l'objet **My.Computer** comporte un objet **FileSytem** qui propose des raccourcis super simples pour manipuler des fichiers.
Plus d'histoire de flux de données et tout le tremblement… Par exemple, pour lire un fichier il suffit d'invoquer la méthode **ReadAllText** de l'objet et le tour est joué !

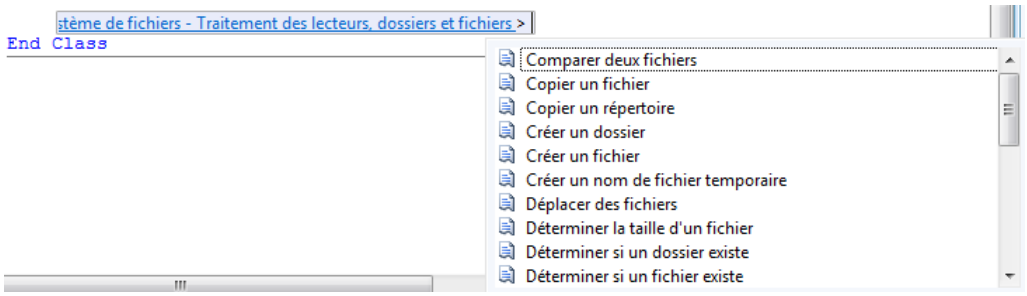
C'est d'ailleurs ce même objet **My.Computer.FileSystem** que vous retrouvez si vous utilisez les extraits de code proposés par Visual Studio :

- Faites un **clic droit** juste au dessus de **End Class** dans la classe **Main**.
- Cliquez **Insérer un extrait**.



- Double cliquez **Notions de base – Collections, types de données, système de fichiers, mathématiques** puis **Système de fichiers – Traitement des lecteurs, dossiers et fichiers**.

Vous tombez sur une batterie de codes d'exemple de manipulation des fichiers en tout genre, basé sur l'objet **My.Computer.FileSystem**.



Attention toutefois ! L'objet **My** est propre au langage Visual Basic alors que la méthode qui utilise les flux de données **Stream** du Framework.NET est applicable dans les autres langages tels que le C#.

Pour vous documenter sur la lecture à partir d'un fichier en Visual Basic en utilisant l'objet **My.Computer.FileSystem** :
http://msdn.microsoft.com/fr-fr/library/wz100x8w.aspx
Pour vous documenter sur l'écriture dans un fichier en Visual Basic en utilisant l'objet **My.Computer.FileSystem** :
http://msdn.microsoft.com/fr-fr/library/c520bdhb.aspx

4.2 Evènements de fichier

Il existe un composant nommé **FileSystemWatcher** qui permet de surveiller les modifications effectuées dans les fichiers et répertoires du système ou d'un ordinateur. Lorsque des modifications ont lieu, un ou plusieurs évènements sont déclenchés et transmis au composant. En écrivant des gestionnaires pour ces évènements, imaginez les scénarios auxquels vous pouvez répondre…

Pour consulter une introduction sur le sujet :
http://msdn.microsoft.com/fr-fr/library/ch2s8yd7.aspx
Pour tout savoir sur le sujet :
http://msdn.microsoft.com/fr-fr/library/342da5th.aspx

Manipuler des données de SQL Express et Access

Sommaire

1 INTRODUCTION 3

1.1 CONTEXTE FONCTIONNEL 3
1.2 CONTEXTE TECHNIQUE 4

2 TRAVAILLER AVEC DES DONNEES DE MICROSOFT OFFICE ACCESS 2007 5

2.1 DEFINIR UNE SOURCE DE DONNEES 6
2.2 ANALYSER LA SOURCE DE DONNEES 16
2.3 AFFICHER LES DONNEES DANS UNE GRILLE 28
2.4 METTRE A JOUR LES DONNEES 39

3 TRAVAILLER AVEC DES DONNEES DE SQL SERVER 2005 EXPRESS EDITION 55

3.1 DEFINIR LA SOURCE DE DONNEES 55
3.2 MANIPULER LES DONNEES 63

4 POUR ALLER PLUS LOIN… 66

4.1 DEFINIR DE NOUVELLES REQUETES UTILISANT DES PROCEDURES STOCKEES 66
4.2 AJOUTER UNE REQUETE PARAMETREE A UN FORMULAIRE DANS UNE APPLICATION WINDOWS 71
4.3 DEVELOPPER AVEC ADO.NET 72

1 Introduction

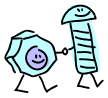
1.1 Contexte fonctionnel

Rappel du contexte fonctionnel du tutorial du coach VB

L'objectif du tutorial du Coach VB est d'accompagner les développeurs à la découverte et la prise en main du langage Visual Basic (VB) pour la construction d'applications avec une approche orientée objet.

Pour rappel, vous pouvez repérer facilement deux caractéristiques importantes du langage à l'aide des logos suivants en marge :

Ce logo marque une fonctionnalité de VB ou de Visual Studio qui permet de développer vite (et juste ☺).

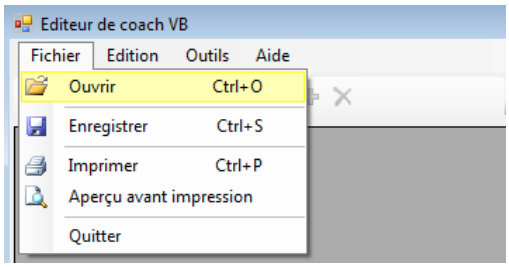


Ce logo met en évidence une caractéristique de la programmation orientée objet.

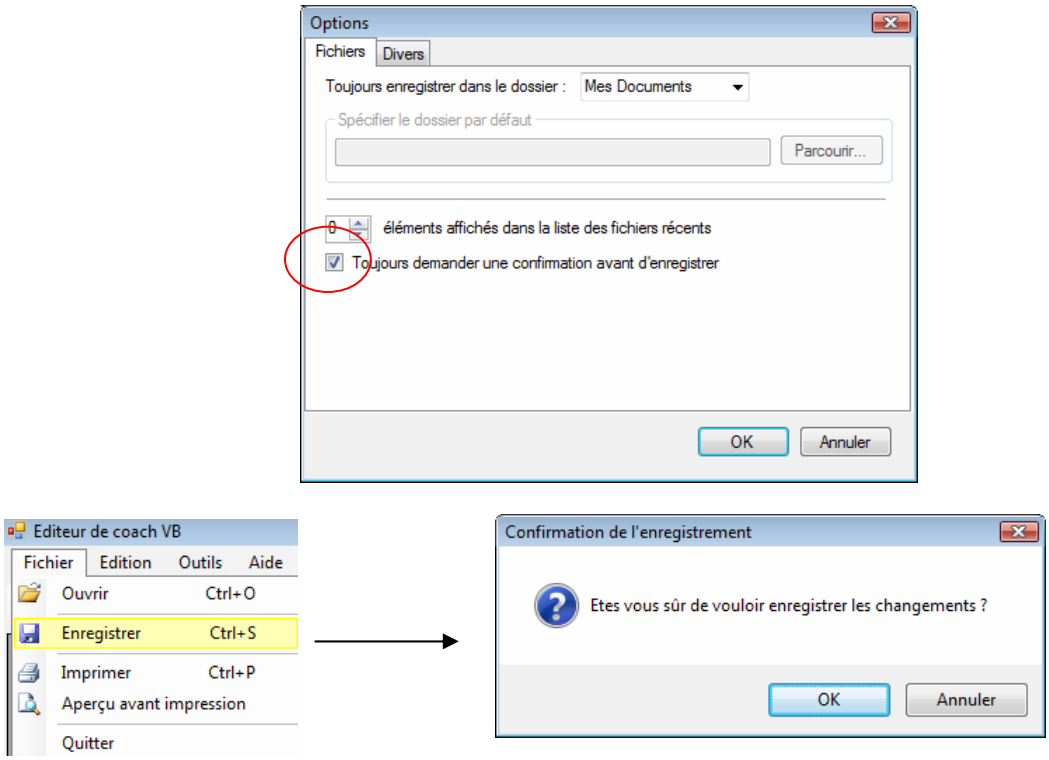
Contexte fonctionnel du cinquième atelier

Dans ce cinquième atelier, l'objectif fonctionnel est rigoureusement le même que dans l'atelier précédent à ceci près que les données manipulées dans l'Editeur ne sont plus extraites d'un fichier texte (d'extension *.coach) mais d'une base de données type **Microsoft Office Access** et **Microsoft SQL Server Express Edition**.

L'interface de l'application va donc rester rigoureusement identique à celle de l'atelier précédent moyennant un léger aménagement du menu **Fichier** de l'application qui ne comportera plus les options **Nouveau** et **Enregistrer sous** puisqu'elles ne sont plus applicables dans le cas de données en bases de données :



Combiné à l'option de menu **Fichier** > **Enregistrer**, nous réutiliserons l'option **Toujours demander une confirmation avant d'enregistrer** de la boîte d'**Options** de l'application pour, si elle est cochée, proposer un message de confirmation à l'utilisateur lui permettant, avant d'enregistrer définitivement ses modifications en base, de les annuler s'il le désire.



Si l'utilisateur clique **Annuler**, les changements ne seront pas enregistrés dans la base de données et la grille sera réinitialisée avec les valeurs initiales.

1.2 Contexte technique

Dans cet atelier, l'objectif est bien sûr d'apprendre à se familiariser avec les concepts fondamentaux de l'accès aux données avec le Framework .NET.

A la fin de cet atelier, vous saurez :

- Ce qu'est **ADO.NET**,
- Ce qu'est un **Fournisseur d'accès aux données**,
- Configurer une **Chaîne de connexion** à une source de données,
- Ce qu'est le **Fichier de configuration** de l'application,
- Manipuler les objets **DataSet** et **TableAdapter** d'ADO.NET pour lire et modifier des données.

La solution de cet atelier est disponible dans le répertoire **..\Atelier 5\Solution** sous la forme de trois répertoires correspondant aux trois grands exercices :

- **1 - Access** : contient la solution de l'exercice 2 : *Travailler avec des données de Microsoft Office Access*.
- **2 - SQL Server** : contient la solution de l'exercice 3 : *Travailler avec des données de Microsoft SQL Server Express Edition.*
- **3 - Pour aller plus loin** : contient le code de solution de la partie *Pour aller plus loin*.

Les fichiers utiles (dont les fichiers de bases de données) auxquels font référence les exercices sont disponibles dans le répertoire **..Atelier 5\Fichiers utiles**.

2 Travailler avec des données de Microsoft Office Access 2007

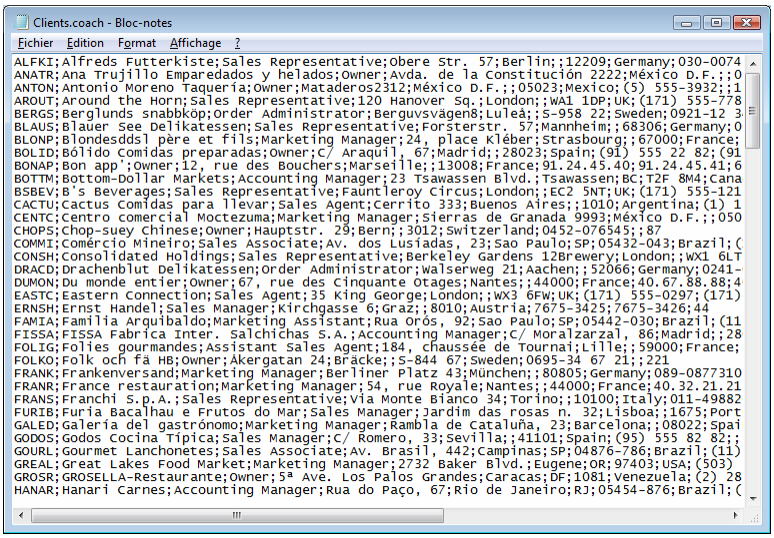
Dans cet exercice, vous allez apprendre à :

- Définir et utiliser une source de données avec Visual Studio,
- Lire des données en provenance d'une base de données,
- Manipuler des données d'une base de données.

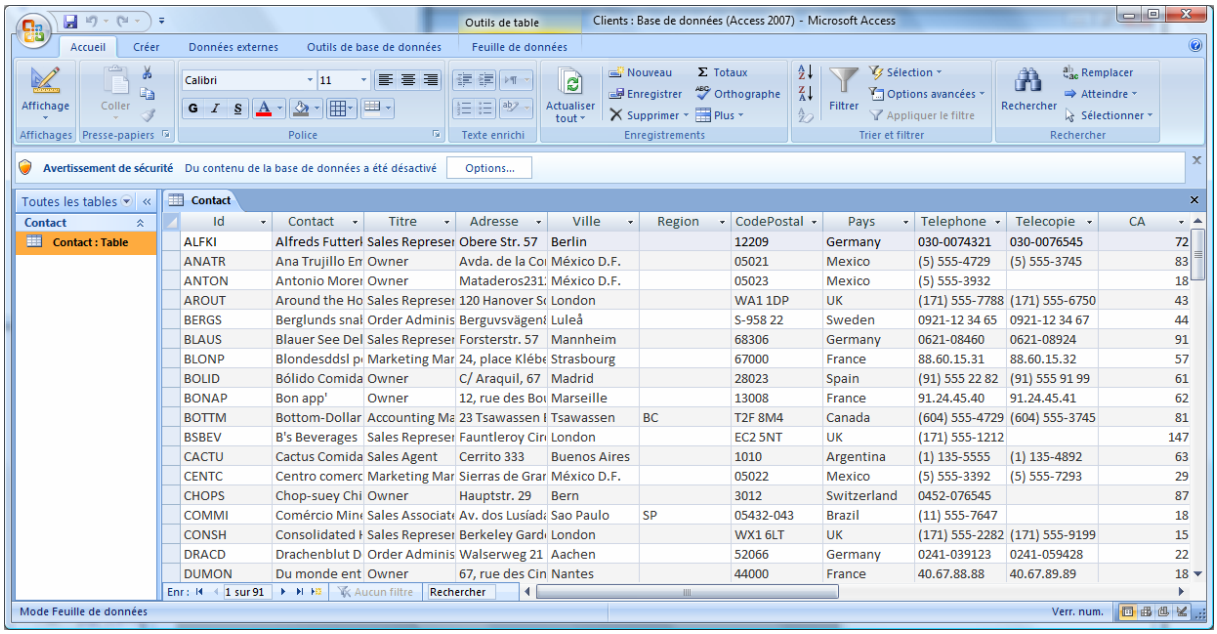
Objectif

L'objectif de ce premier exercice est de découvrir les principes élémentaires d'accès à des données stockées dans une base de données avec le Framework .NET. Dans cette première partie, le moteur de base de données est Microsoft Office Access 2007.

Nous vous proposons de travailler sur le même jeu de données « clients » que vous avez manipulé au travers de fichiers d'extension ***.coach** à l'atelier 4 précédent :



Mais cette fois-ci, les données sont stockées dans une base **Microsoft Office Access 2007** plutôt que sur un fichier.



Pour vous permettre de vous concentrer sur l'essentiel, un exemple de cette base vous est fourni avec le code de l'atelier dans le dossier **Atelier 5\Fichiers utiles** dans le fichier **Clients.accdb**.
Pour pouvoir utiliser cette base de données, il vous faut évidemment installer le moteur de la base sur votre poste de développement.
Si vous n'avez pas la possibilité d'installer **Microsoft Office Access 2007**, n'ayez pas d'inquiétude. Il vous suffit de télécharger et d'installer les **composants de connectivité des données** fournis gratuitement par **Office System**.

msdn

Téléchargez le **pilote d'Office System** ici :
http://www.microsoft.com/downloads/details.aspx?displaylang=fr&FamilyID=7554f536-8c28-4598-9b72-ef94e038c891

2.1 Définir une source de données

Dans cet exercice, vous allez apprendre à :

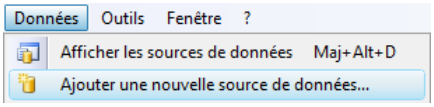
- Définir une source de données.
- Utiliser l'Assistant Configuration de sources de données,
- Définir une chaîne de connexion,
- Choisir un fournisseur d'accès,
- Extraire un paramètre dans un fichier de configuration,
- Utiliser un fichier de données.

Déroulement de l'exercice :

1. Ouvrez le projet précédent réalisé lors de l'atelier 4 :
   - Lancez Visual Studio à partir du menu **Démarrer** > **Tous les programmes** > **Microsoft Visual Basic 2008 Express Edition**.
   - Menu **Fichier** > **Ouvrir un projet**.

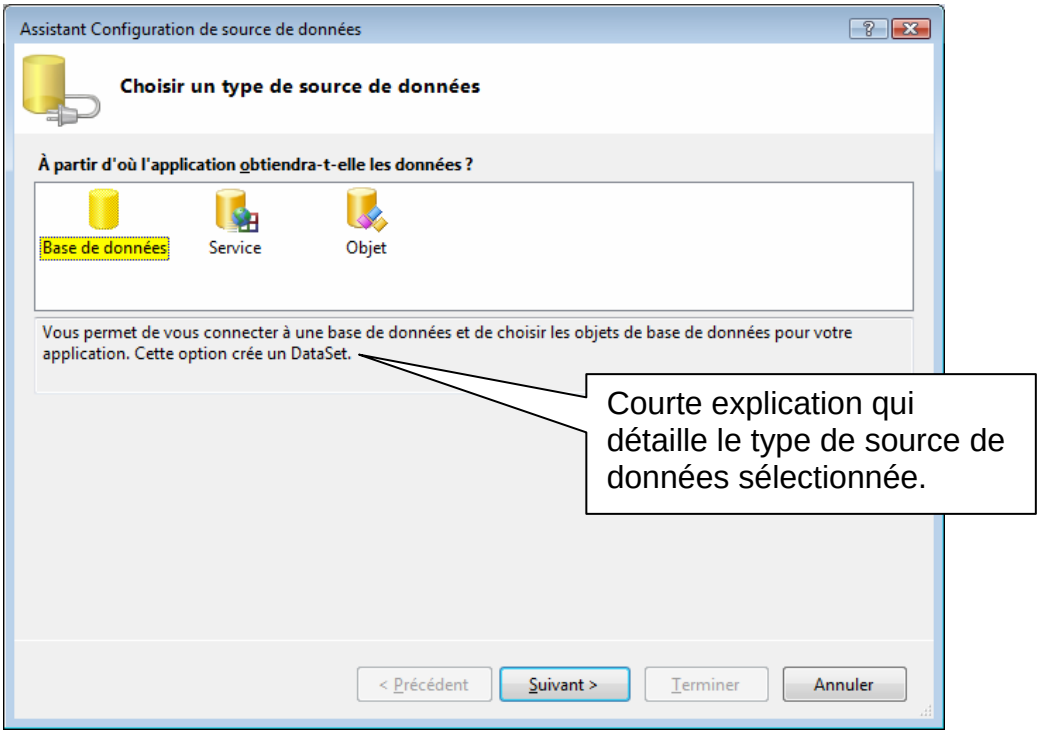
- Retrouvez le fichier **Atelier 4.sln** que vous avez créé lors de l'atelier 4 ou, si vous n'avez pas fait l'atelier précédent, récupérez le projet de solution dans le répertoire : **..\Atelier 5\Démarrage\Atelier 5.sln**.

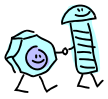
2. Créez une source de données pointant sur le fichier Access fourni en exemple :
   - Sélectionnez le menu **Données** > **Ajouter une nouvelle source de données** dans la barre de menu **Standard** de Visual Studio.



Une fois de plus, nous allons nous laisser guider par Visual Studio. Son **Assistant Configuration de sources de données** suit l'approche la plus standard et la plus simple pour accéder aux données avec le Framework .NET.

- Dans l'écran **Choisir un type de source de données**, sélectionnez **Base de données**.





Vous constatez au passage que Visual Studio nous propose de travailler avec d'autres sources de données qu'une base, telle qu'un objet ou un service. Ces deux solutions correspondent à des scénarios d'application plus complexes dans lesquels les données ne proviennent pas simplement d'une base de données locale mais par exemple d'autres applications ou de sources hétérogènes. Dans ce cas, l'accès aux données fait intervenir des couches de composants ou services supplémentaires qui facilitent l'interaction avec les données.

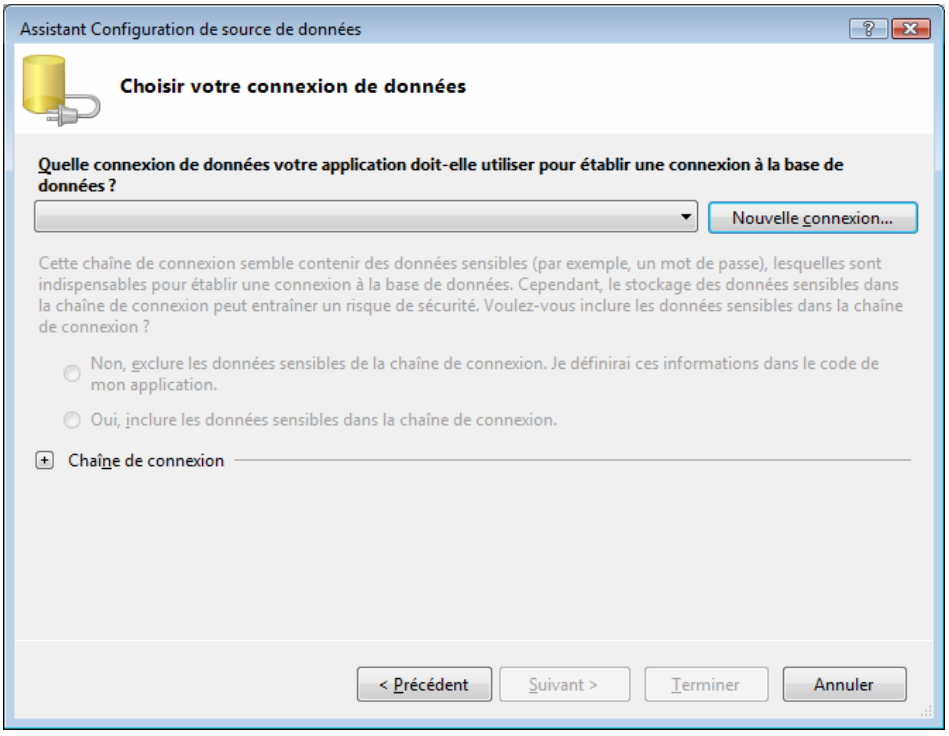
La description de la source de données indique que l'option choisie va donner lieu à la création d'un **DataSet**.

C'est quoi un **DataSet** ?
Il s'agit d'une structure mémoire (donc un objet) très riche qui permet de travailler avec des données de manière indépendante de leur provenance. Nous allons nous servir de cet objet pour stocker en mémoire les données que nous allons extraire de la base de données Microsoft Access.
Il faut savoir qu'il constitue l'un des deux principaux composants d'ADO.NET.

Qu'est ce que **ADO.NET** ?
Vous voulez vous connecter à une source de données ? En extraire des données puis les manipuler et les réinjecter dans la source de données initiale ? Comme d'habitude, il suffit de se reposer sur le Framework .NET qui fournit tous les composants nécessaires à la gestion de données. Cet ensemble de classes s'intitule **ADO.NET**.

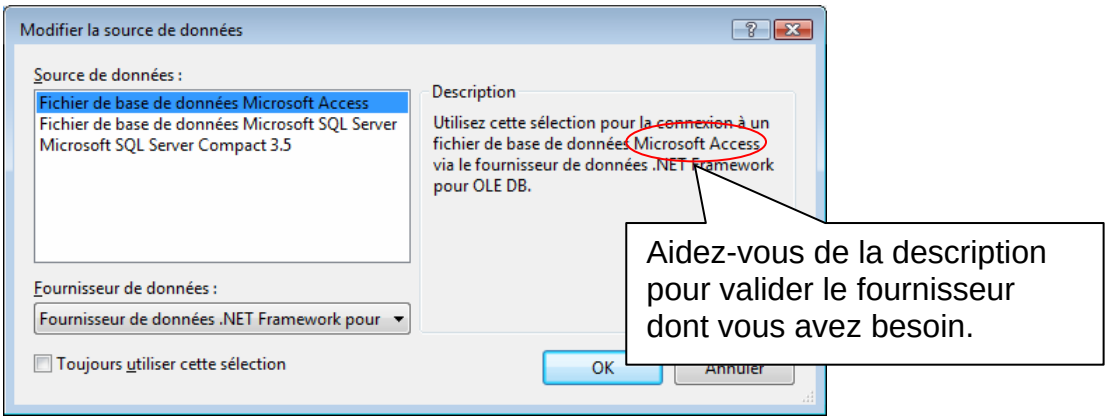
Pour avoir une vue d'ensemble d'ADO.NET :
http://msdn.microsoft.com/fr-fr/library/h43ks021.aspx

- Cliquez **Suivant**.
- Dans l'écran **Choisir votre connexion de données**, cliquez **Nouvelle connexion**.



- Dans la boîte **Ajouter une connexion**, cliquez sur **Modifiez** pour sélectionner le fournisseur de données pour accéder à une base Access.

- Sélectionnez **Fichier de base de données Microsoft Access** dans la liste des sources de données proposées :

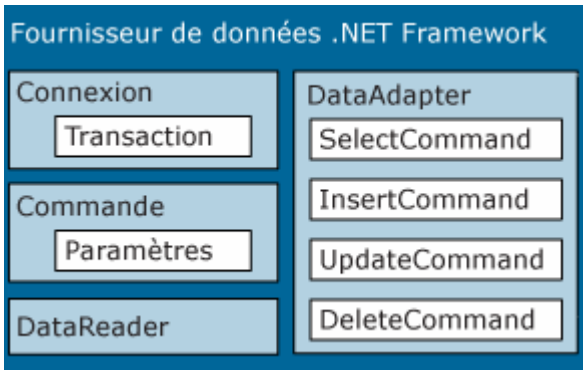


Qu'est ce qu'on entend par *fournisseur de données* ?
Après le **DataSet**, le **Fournisseur de données** constitue le deuxième composant essentiel pour la manipulation de données d'ADO.NET.

En gros de quoi avons-nous besoin pour discuter avec une base de données ? C'est un peu comme si vous commandiez un thé glacé à une terrasse de café.

- D'abord, Il faut établir ce qu'on appelle une ***connexion*** avec le serveur de base de données pour montrer patte blanche. Cela revient à interpeller un serveur du café qui vous identifier comme un client potentiel.
- Il faut ensuite pouvoir communiquer et exécuter des ordres de traitement sur la base de données, c'est-à-dire que vous devez formuler explicitement la ***commande*** d'un thé glacé au serveur.
- Et enfin, il faut pouvoir ***stocker*** le résultat de vos investigations sur la base de données quelque part en mémoire. Typiquement, pour lire des données, il vous faut un ***curseur*** qui permet de balayer les enregistrements un à un. C'est le verre et sa paille…

Ces trois types de composants, ADO.NET nous les fournit via des fournisseurs de données. Par exemple, un objet de type **Connection** permet d'établir une connexion avec la base, un objet **Command** d'exécuter une requête etc… :

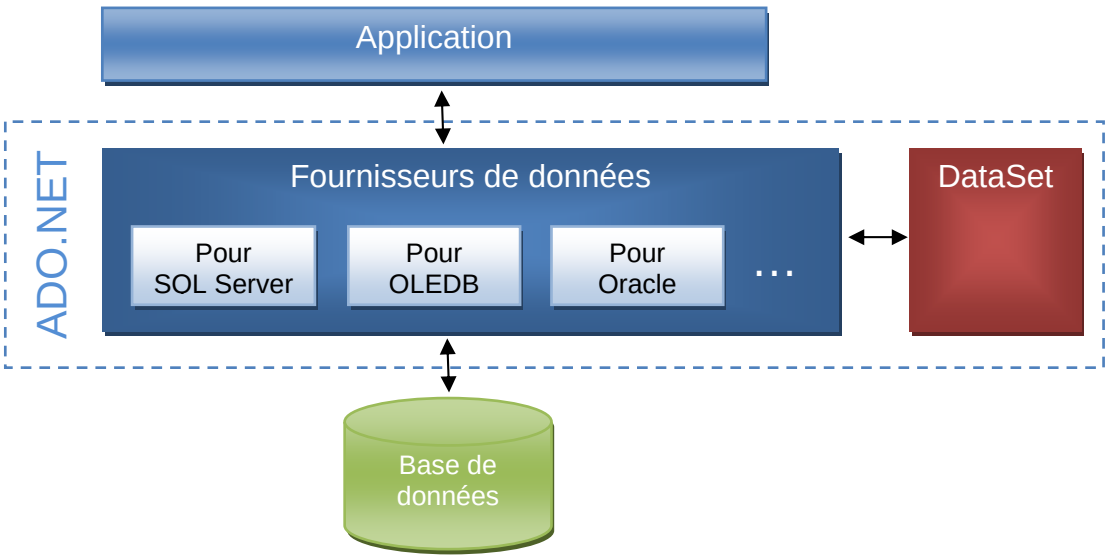


Principaux objets d'un fournisseur

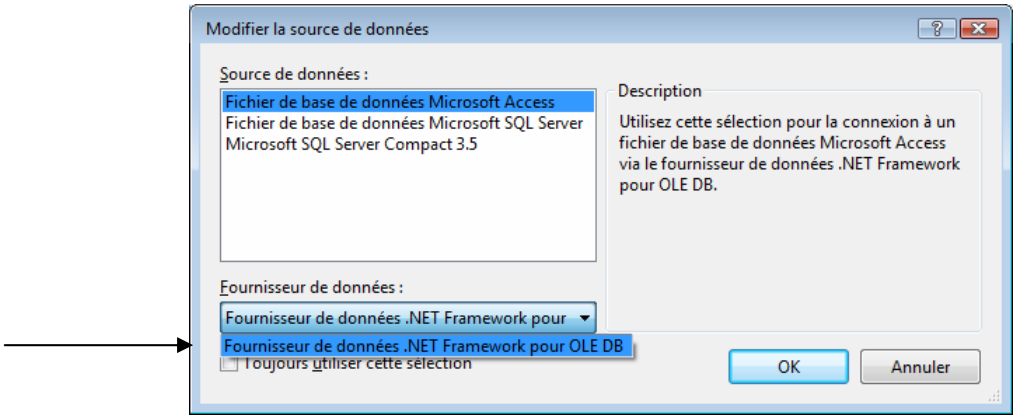
Pourquoi *des* fournisseurs ? Lequel choisir ?
Le souci, c'est que vous ne vous connectez pas à une base de données SQL Server exactement de la même manière qu'à une base Access… De même que vous ne commandez pas l'une avec les mêmes ordres SQL que l'autre. En gros pour commander une pizza et vous la faire livrer, vous allez devoir établir un autre style de communication que pour un simple thé glacé à une terrasse de café, par exemple par Internet ou par téléphone.
Donc les objets que nous allons manipuler ne peuvent pas être génériques ! Leur caractéristique va dépendre de la base de données avec laquelle vous voulez travailler. C'est pour cela que le Framework.Net ne comprend pas un fournisseur de données mais plusieurs, qui permettent chacun de s'interfacer avec une source de données spécifique.

Pour connaître tous les fournisseurs de données inclus dans le Framework .NET et pour les comparer :
http://msdn.microsoft.com/fr-fr/library/a6cd7c08.aspx

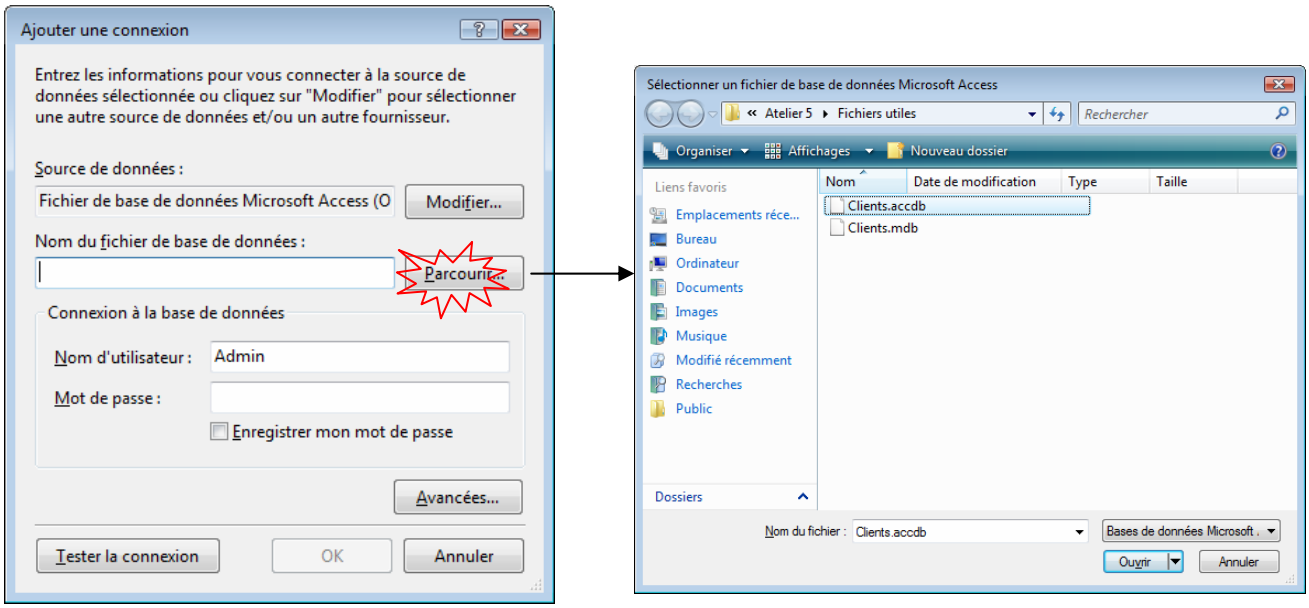
Et le **DataSet** dans tout ça ?
Par opposition aux objets qui sont fournis par les fournisseurs de données, le **DataSet** est un objet complètement indépendant de toute source de données. Donc lui est unique et n'est pas inclus dans les fournisseurs de données (il est fourni par l'espace de noms **System.Data**). C'est un peu la glacière dans laquelle vous pouvez transporter aussi bien une pizza qu'un thé glacé. Peu importe d'où ça vient, les données sont là !
Cet objet est extrêmement intéressant car il permet de fonctionner sur un mode dit **déconnecté** (par opposition à connecté) de la base de données. Après tout, vous n'avez plus besoin du serveur ni de la cuisine pour consommer votre thé ou votre pizza. Nous aurons l'occasion de revenir sur ce sujet ultérieurement dans cet atelier.



Tout ça pour dire que la première chose que Visual Studio vous demande de faire est donc de déterminer le fournisseur d'accès le plus adapté à votre source de données. Il déterminera les objets dont vous disposerez pour programmer l'accès à celle-ci. Pour Microsoft Office Access, le fournisseur correspondant est **Fournisseur de données .NET Framework pour OLE DB**.

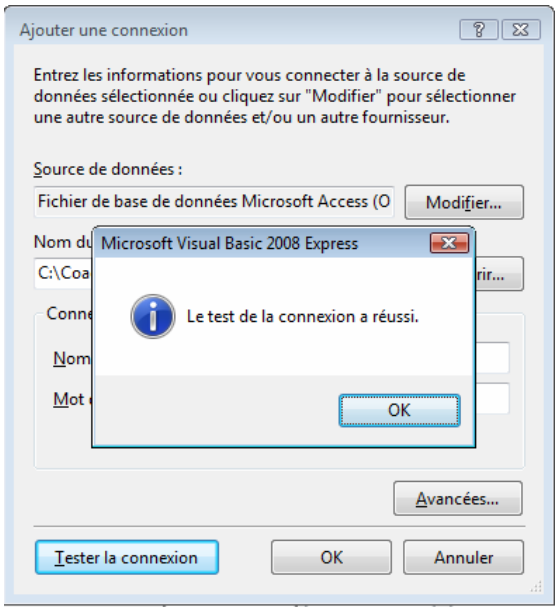


- Validez la sélection du fournisseur en cliquant **OK**.
- De retour dans l'écran **Ajouter une connexion**, cliquez **Parcourir** à droite de la zone de nom de fichier de base de données.
- Sélectionnez le fichier de base de données Microsoft Office Access 2007 **Clients.accdb** fourni en exemple dans le dossier **..\Atelier 5\Fichiers utiles** :



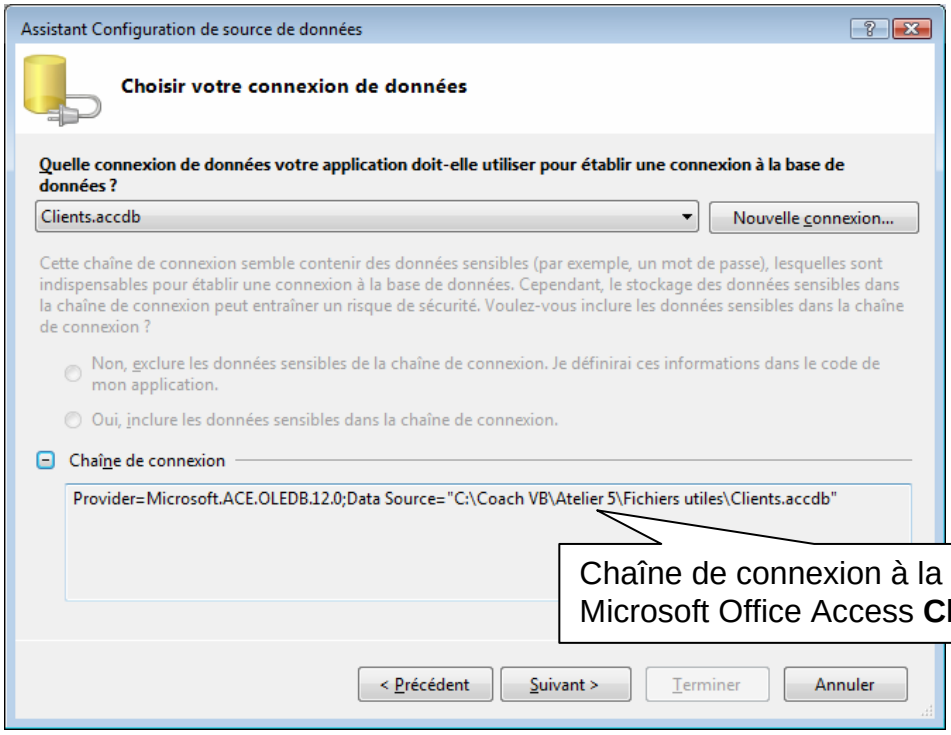
- Cliquez **Ouvrir**.

Avant de valider la connexion établie, utilisez le bouton **Tester la connexion** de la boîte **Ajouter une connexion** pour vérifier que Visual Studio est à même d'établir la connexion sur la base de données avec les éléments que vous lui avez fournis :



Un message vous informe si la connexion a pu être établie ou non.

- Une fois la connexion vérifiée, cliquez sur **OK** pour quitter la boîte **Ajouter une connexion**.
- De retour dans l'écran **Choisir votre connexion de données**, cliquez le sigle ⊞ devant **Chaîne de connexion** pour afficher le détail de la connexion que vous venez de configurer :



Qu'est ce qu'une *chaîne de connexion* ?
C'est une chaîne de texte qui regroupe toutes les informations nécessaires pour l'établissement d'une connexion avec la base de

données. Chaque information est inscrite sous la forme :

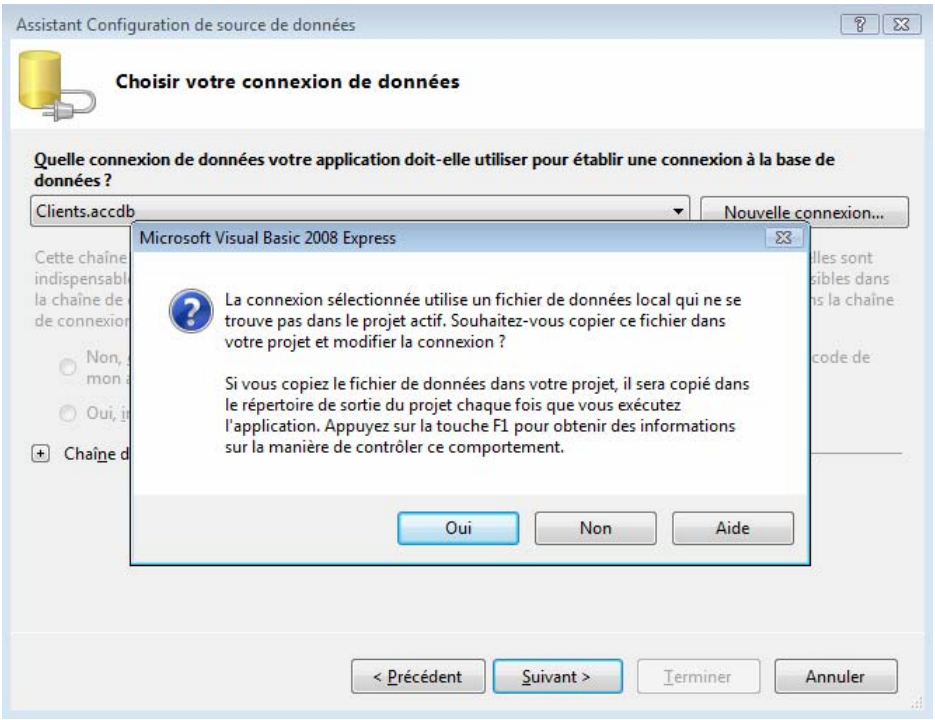
Clé = Valeur

En somme, pour communiquer avec le moteur de base de données d'Access, vous devez spécifier deux clés :

- **Provider** dans laquelle est spécifié le fournisseur de données, ici celui pour Access,
- **Data Source** dans laquelle est définie la source de données proprement dite, ici le fichier de données d'extension *.accdb.

- Validez en cliquant **Suivant**.

Visual Studio détecte que vous utilisez un fichier de données et vous suggère de l'inclure *en tant que fichier* dans le projet. Cela revient à le copier à la racine du projet. Le chemin inscrit dans la chaîne de connexion précédente est alors aussitôt modifié en conséquence.
Si vous déclinez la proposition, le fichier reste à son emplacement initial, un peu comme une base de données distante.

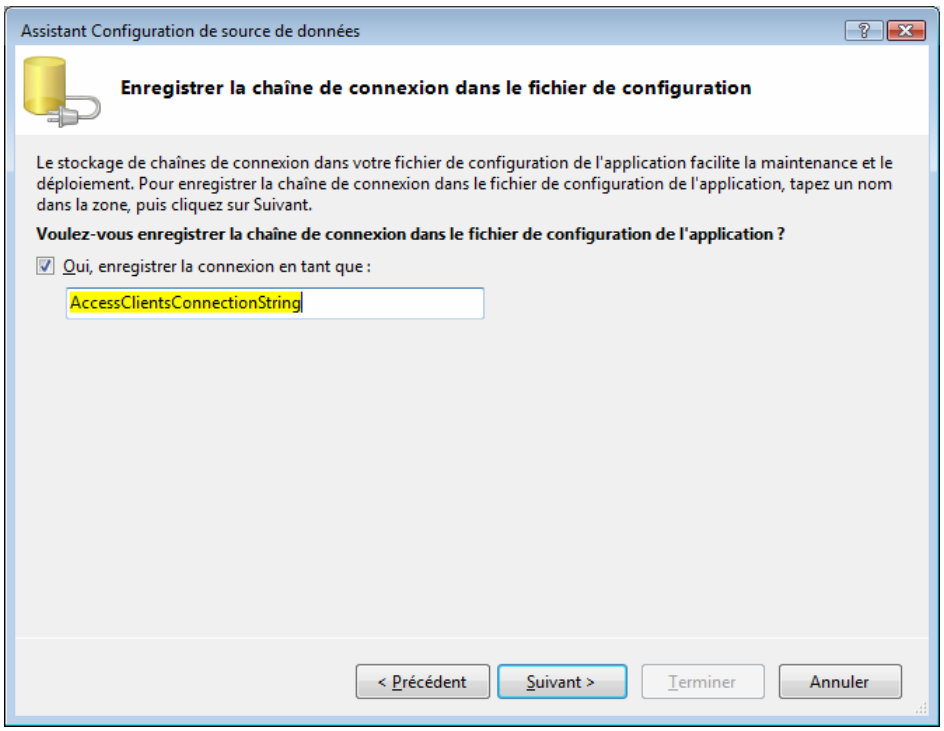


Quel est l'intérêt de copier le fichier dans votre projet ?
Il repose sur le comportement de Visual Studio à l'encontre du fichier au moment de la compilation du projet.
En effet, pour vous aider à déboguer, Visual Studio se propose de copier le fichier utilisé à l'exécution dans le répertoire de sortie **\bin** de l'application, en plus de la copie présente dans le répertoire racine du projet. Jusque là rien de transcendant puisque cela revient à placer le fichier au même endroit que l'exécutable du projet. C'est plutôt logique.
Sauf que l'idée est que la copie à partir du fichier dans le répertoire racine peut avoir lieu à chaque compilation de sortes que le fichier est

systématiquement réinitialisé à chaque fois. Du coup, c'est super utile pour travailler sur une base propre malgré les nombreux tests qu'impose le débogage du programme à chaque exécution. Toute modification apportée au fichier de données est réinitialisée à la prochaine exécution.

Pour en savoir plus sur comment gérer des fichiers de données locaux dans un projet :
http://msdn.microsoft.com/fr-fr/library/ms246989.aspx

- Répondez **Oui** à la question dans le contexte de l'atelier.
- Dans l'écran **Enregistrer la chaîne de connexion dans le fichier de configuration**, modifiez le nom de la chaîne en : **AccessClientsConnectionString**.



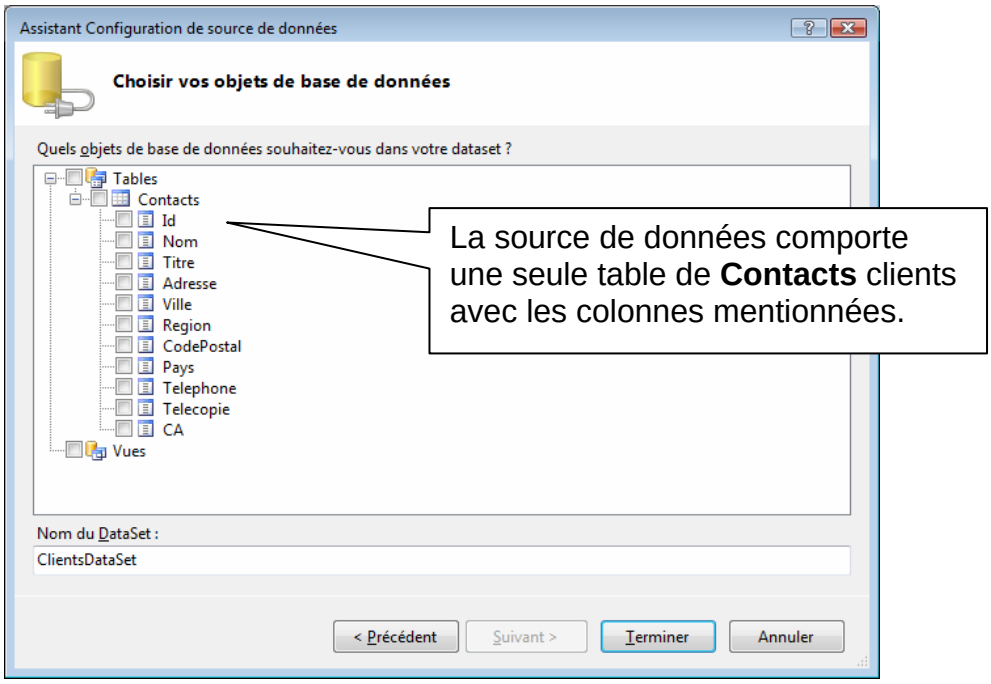
Dans cette étape, Visual Studio vous propose d'enregistrer la chaîne de connexion dans le fichier de configuration de l'application.

C'est quoi un *fichier de configuration* ?
Il s'agit d'un fichier XML dans lequel sont stockés tous les paramètres spécifiques de l'application. Le runtime lit les informations qu'il contient au moment de l'exécution.
L'intérêt d'externaliser des informations dans un tel fichier (plutôt que de les programmer en dur dans le code du programme) est bien sûr d'en faciliter la maintenance puisqu'au format texte, ces données n'ont pas besoin d'être recompilées en cas de modification.

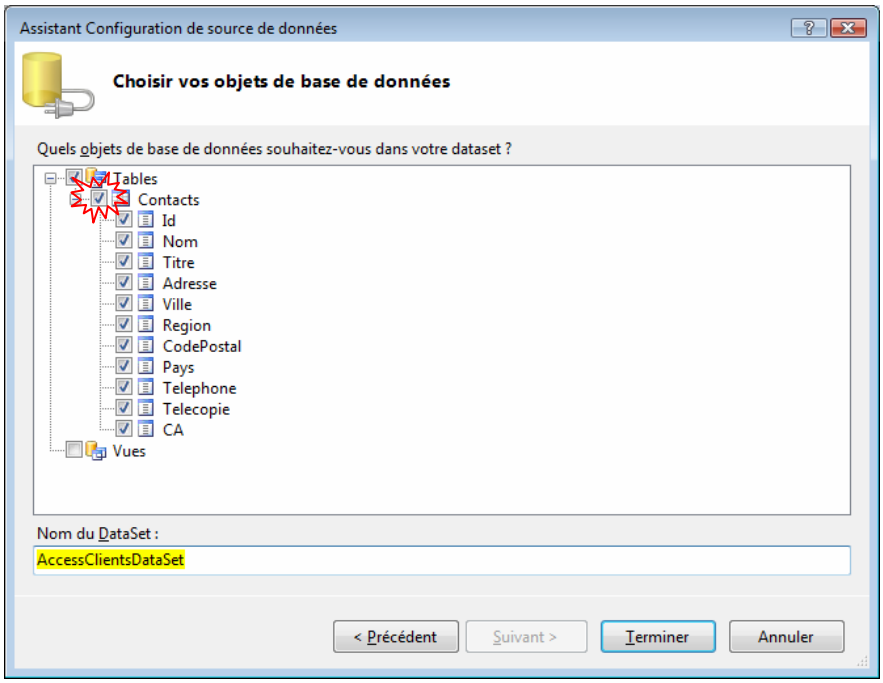
En externalisant la chaîne de connexion dans le fichier de configuration de votre application, vous vous simplifiez la vie le jour où une quelconque modification s'impose, telle que le changement de nom du

fichier de données ou une modification de son emplacement.

- Cliquez sur **Suivant** pour accepter l'enregistrement de la connexion
- Dans l'étape **Choisir vos objets de base de données**, étendez les différents objets en cliquant sur ⊞ pour voir les objets disponibles dans la source Access.



- Cochez la table **Contacts**. Toutes les colonnes sont alors automatiquement sélectionnées.
- Modifiez le nom proposé par défaut pour l'objet **DataSet** par exemple en **AccessClientsDataSet**.



Qu'est ce que vous venez de faire ?
L'objectif de l'Assistant est de préparer un objet de type **DataSet** qui servira à manipuler vos données une fois extraite de la base de données. En somme, vous venez de dessiner l'ossature de l'objet mémoire pour qu'il puisse stocker des enregistrements de type **Contacts** avec les colonnes spécifiées.

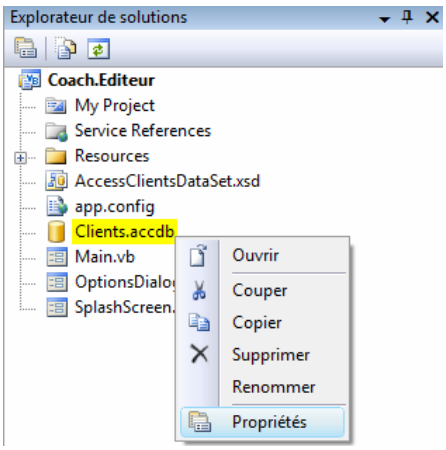
- Cliquez **Terminer** pour terminer la configuration de la source de données.

2.2 Analyser la source de données

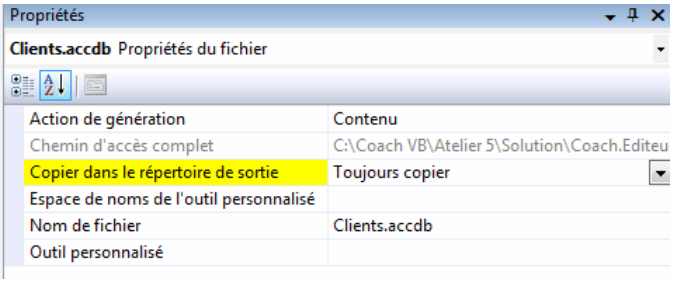
L'objectif de cet exercice est de comprendre ce qui a été construit par l'Assistant Configuration de sources de données dans l'exercice 2.1.

Déroulement de l'exercice :

1. Observez les nouveaux éléments ajoutés au projet dans l'Explorateur de solutions. Commencez par le fichier de données Clients.accdb que nous avions demandé à l'Assistant d'inclure dans le projet :
   - Sélectionnez le fichier de données **Clients.accdb** dans l'**Explorateur de solutions**.
   - Faites un clic droit sur le fichier > **Propriétés** pour afficher la fenêtre de propriétés du fichier.

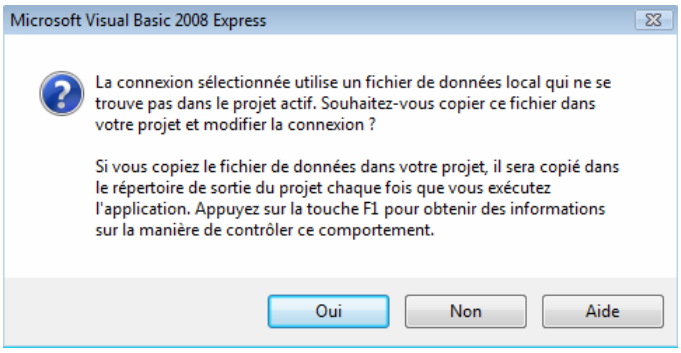


Notez la propriété **Copier dans le répertoire de sortie**.



C'est elle qui contrôle le comportement de Visual Studio vis-à-vis du

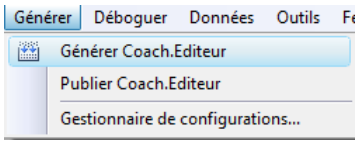
fichier de données tel que nous l'avons évoqué à l'exercice précédent au moment de la question suivante :



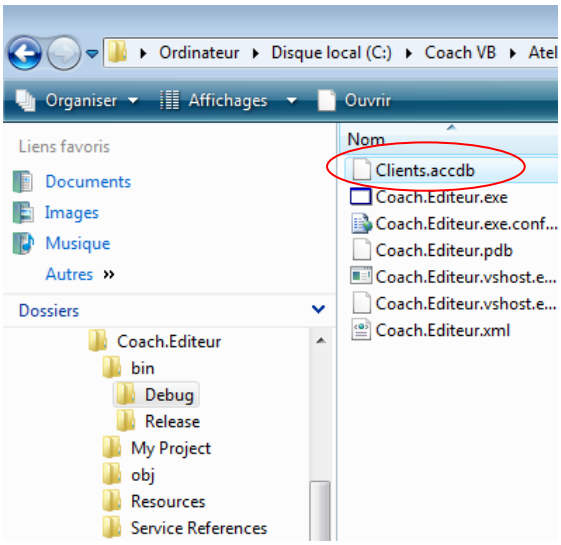
Si le comportement par défaut ne vous convient pas, il vous suffit de modifier cette propriété en conséquence.

msdn Pour retrouver la signification des options de cette propriété : http://msdn.microsoft.com/fr-fr/library/ms246989.aspx

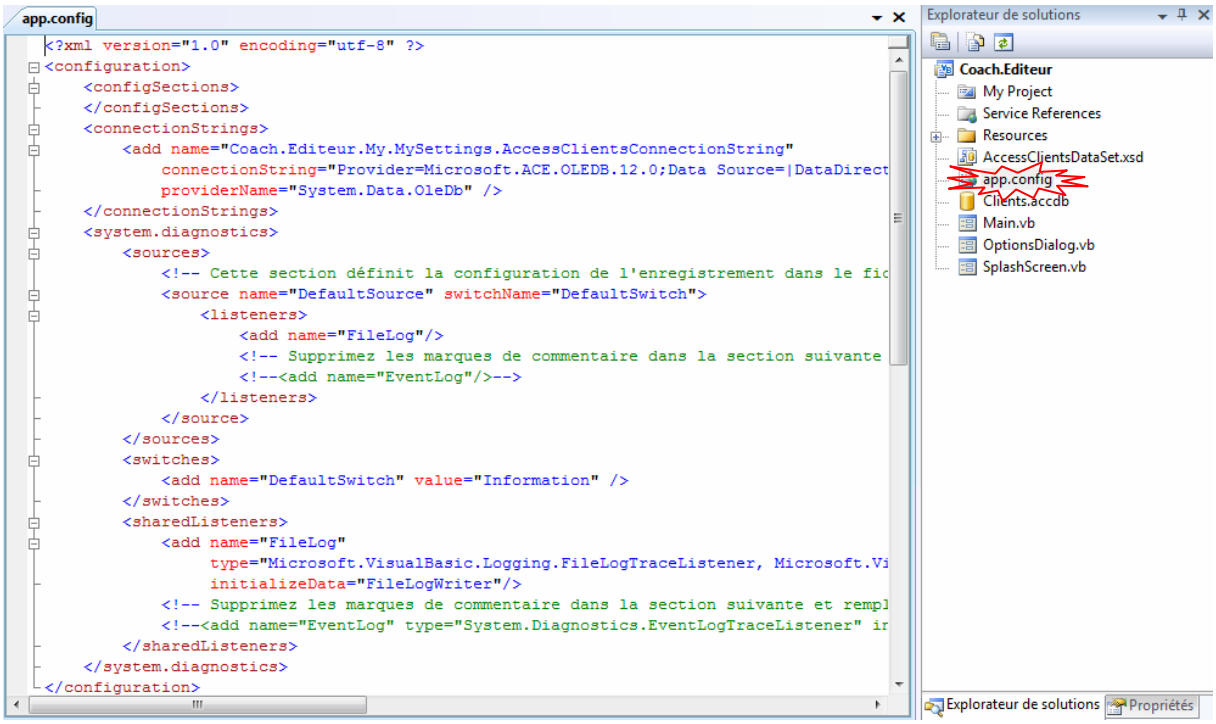
- Cliquez le menu **Générer** > **Générer Coach.Editeur** pour compiler le projet.



- Vérifiez que le fichier de données est effectivement copié dans le répertoire **\bin\Debug** de sortie de l'application dans l'**Explorateur Windows** :

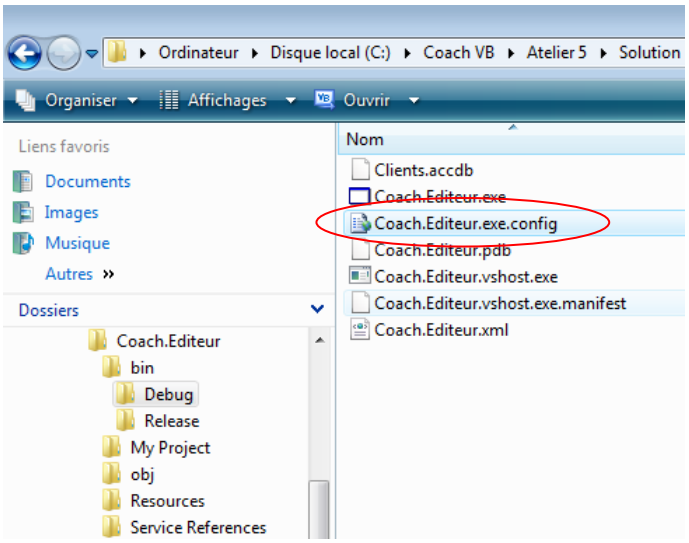


2. Notez ensuite qu'un fichier app.config a été ajouté au projet :
   - Double cliquez sur le fichier **app.config** dans l'**Explorateur de solutions**.

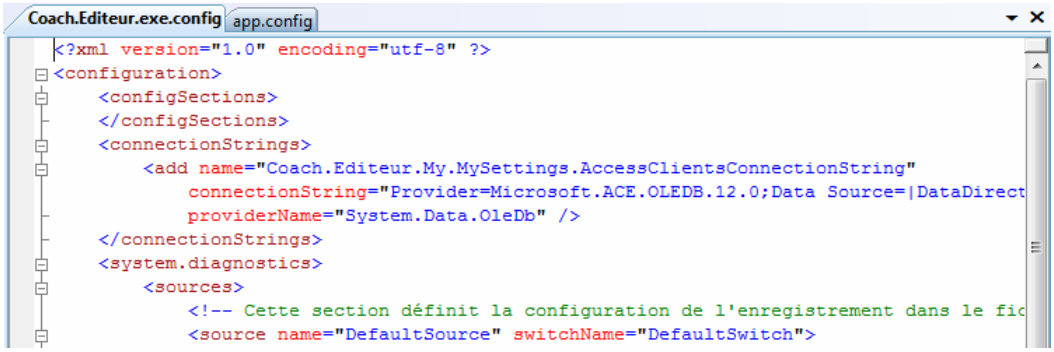


Il s'agit du fichier de configuration de l'application. Au moment de la compilation le fichier prend le nom de l'exécutable suivi de l'extension ***.config**.

- Basculez dans l'**Explorateur Windows** pour vérifier qu'un fichier de configuration a été également ajouté dans le répertoire de sortie de l'application au moment de la compilation précédente :

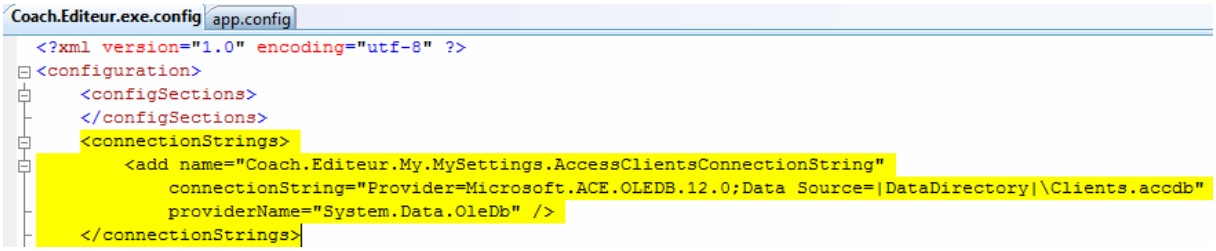


- Double cliquez le fichier pour l'éditer dans Visual Studio. Vous constatez qu'il s'agit rigoureusement du même contenu que dans **app.config** vu précédemment.



C'est donc dans ce fichier que vous devez retrouver la définition de la chaîne de connexion à la source de données que vous aviez nommée **AccessClientsConnectionString**.

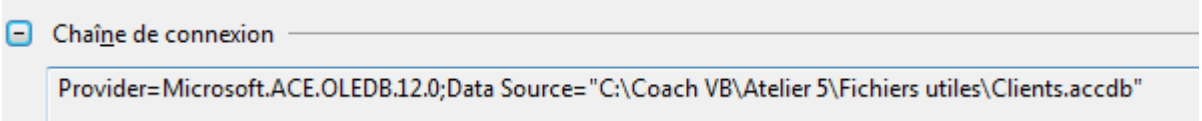
- Repérez la balise **<connectionStrings>** qui est prévue pour recevoir la collection de chaînes de connexion définies dans l'application.
- Vérifiez qu'elle contient la définition de la chaîne de connexion **AccessClientsConnectionString** :



Vous retrouvez le format de la chaîne de connexion vu lors de la configuration de la source de données avec l'Assistant basé sur des couples **Clé = Valeur** :

connectionString="Provider=Microsoft.ACE.OLEDB.12.0;
Data Source=|DataDirectory|\Clients.accdb"

Avez-vous remarqué que cette chaîne n'est pas tout à fait identique à celle que vous aviez obtenue au moment de la configuration avec l'Assistant à savoir :



L'attribut **Data Source** n'est plus tout à fait le même car vous avez décidé dans une étape ultérieure de copier le fichier de données dans le dossier du projet. Du coup, Visual Studio a mis automatiquement à jour la chaîne de connexion pour qu'elle s'appuie non plus sur le chemin initial mais sur le dossier de projet.

Que signifie **|DataDirectory|** ?
Il s'agit d'une chaîne de substitution pour indiquer dynamiquement le chemin du

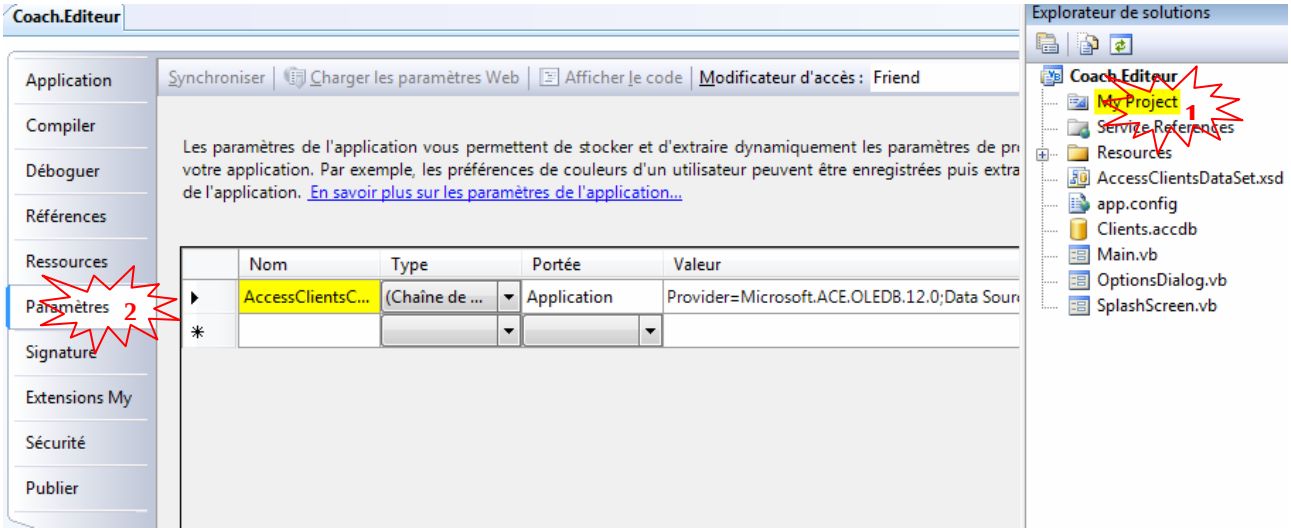
fichier de données. Par défaut il représente le répertoire de l'application.

A noter que si la chaîne de connexion doit comporter des informations sensibles telles que des paramètres de sécurité, il est possible de la chiffrer pour une meilleure protection de l'information.

Pour en savoir plus sur la fonctionnalité de **configuration protégée** appliquée à une chaîne de connexion :
http://msdn.microsoft.com/fr-fr/library/53tyfkaw(VS.80).aspx

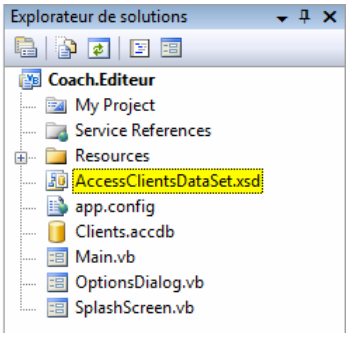
La chaîne de connexion, tout comme n'importe quel paramètre que vous souhaitez configurer dans le fichier de configuration de l'application, est également éditable de façon plus conviviale qu'au format XML via le **Concepteur de projet** de l'application.

- Double cliquez **My Project** dans l'**Explorateur de solutions** pour afficher le **Concepteur de projet**.
- Sélectionnez l'onglet **Paramètres** dans le volet de navigation à gauche :

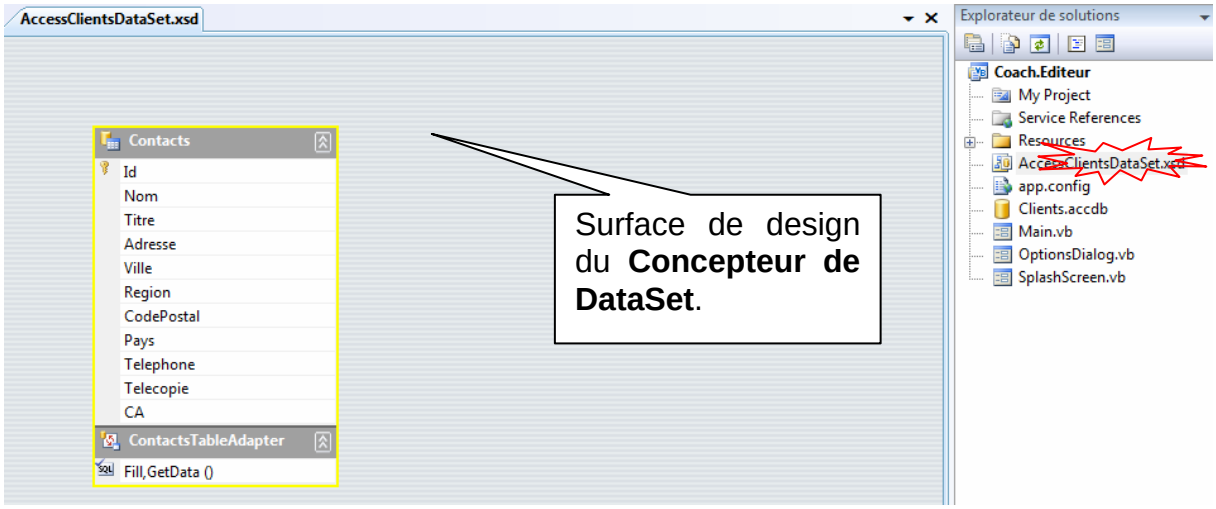


La chaîne de connexion est bien là ! Elle est directement modifiable de façon statique depuis cet écran qui est le reflet exact des paramètres personnels que vous pouvez ajouter au fichier de configuration.
Nous verrons dans la suite de cet atelier comment utiliser cette même information par code.

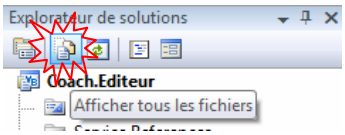
3. Dernier élément généré par l'Assistant et non le moindre : *le groupe de données* (ou DataSet) :
   - Il se matérialise par le fichier **AccessClientsDataSet.xsd** dans l'**Explorateur de solutions** qui décrit la structure (schéma) du **DataSet**:



- Double cliquez sur le fichier **AccessClientsDataSet.xsd** pour afficher le **Concepteur de DataSet** :

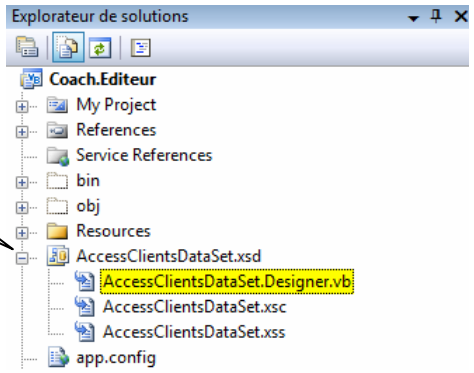


Le **Concepteur de DataSet** vous donne une représentation visuelle et des outils pour manipuler les objets qui constituent le **DataSet** que vous avez généré avec l'Assistant.
En réalité, le groupe de données que nous avons généré correspond évidemment à du code généré et ajouté au projet. Pour le voir, il suffit de cliquer l'icône **Afficher tous les fichiers** dans la barre d'outils de l'**Explorateur de solutions**.



Le fichier qui contient le code généré par le Concepteur (*Designer* en anglais) de DataSet est le fichier d'extension ***.Designer.vb**.

Cliquez sur le sigle ⊞ pour faire apparaître tous les fichiers.



Notez que le contenu de ce fichier est écrasé chaque fois qu'une modification est effectuée depuis la surface de design du Concepteur.

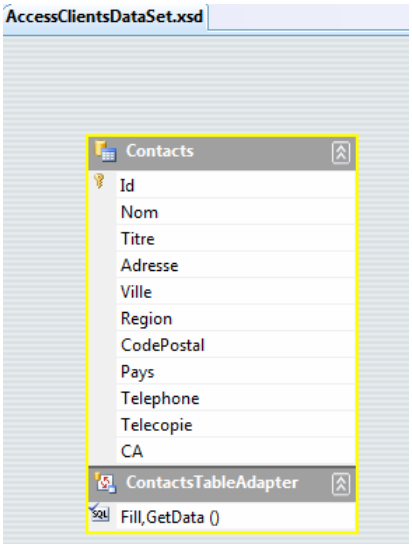
Pour tout savoir sur le **Concepteur de DataSet** :
http://msdn.microsoft.com/fr-fr/library/314t4see.aspx

Les deux autres fichiers ont pour extension :

- ***.xsc** : ce fichier stocke vos préférences utilisateurs au niveau de la configuration de la source de données.
- ***.xss** : ce fichier stocke les informations du Concepteur relatives aux objets du DataSet, telles que leur emplacement ou leur taille.

Si vous voulez définir votre propre code et personnaliser le DataSet, il peut y avoir un troisième fichier d'extension ***.vb** qui contient vos définitions dans une classe partielle. (Pour le générer, il suffit de faire un clic droit sur la surface du **Concepteur de DataSet** > **Afficher le code**).

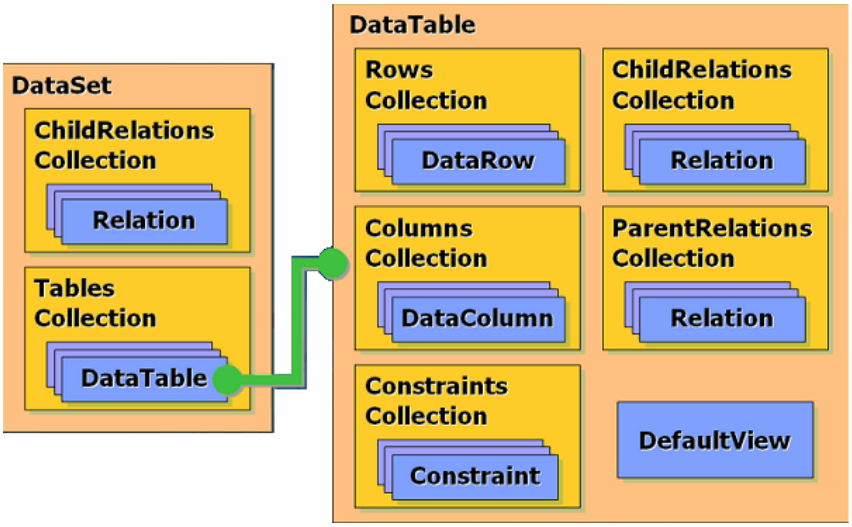
- Basculez à nouveau sur le **Concepteur de DataSet** :



Le concepteur affiche deux objets : un objet **Contacts** et un objet **ContactsTableAdapter**.

A quoi correspondent ces objets ?

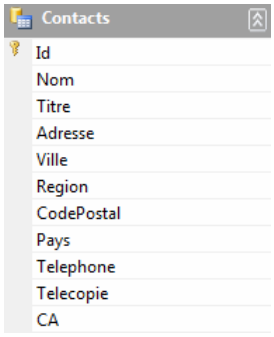
Penchons nous (pas trop quand même pour ne pas tomber) sur la structure d'un groupe de données (DataSet). Elle est très proche de celle d'une base de données relationnelle dans la mesure où un **Dataset** est constitué d'une hiérarchie d'objets représentant des tables (**DataTable**), des lignes (**DataRow**), des colonnes (**DataColumn**), des contraintes (**Constraint**) et même des relations (**Relation**) :



La seule chose importante à comprendre est qu'on travaille ici ***en mémoire***, et qu'on manipule des ***objets***. C'est pourquoi il n'y aucun lien entre un DataSet et ses sources de données physiques.

L'objet **Contacts** représente l'une des tables que peut contenir votre DataSet. Dans notre cas, nous n'avons qu'une seule table car nous n'avons sélectionné qu'un seul objet de la base au moment de la configuration de la source de données.

Avez-vous remarqué que l'objet **DataTable** s'appelle **Contacts** comme la table dans la base de données ?
Non seulement il s'appelle **Contacts**, mais en plus il contient une collection de colonnes (d'objets **DataColumn**) dont les noms sont le reflet des noms de colonnes dans la base de données. Pratique non ?
En fait, notre DataSet a la particularité d'être **typé**.



C'est quoi un *DataSet typé* ?
Comme son nom l'indique, cela veut dire qu'il est typé…ah ouais ! On est bien avancé avec ça…
Plus sérieusement, il est vraiment question de **type de données**. L'idée est de vous aider à manipuler les données le plus simplement possible.

En théorie, un DataSet contient une collection de DataTable qui contiennent elles-mêmes des collections de lignes (DataRow) et colonnes

(DataColumn). Cela veut dire que pour accéder à un élément de données, il vous faut manipuler des écritures du type tabulaire :

Tables(indexe).Rows(indexe de la ligne)(indexe de la colonne)

et en plus vous récupéreriez une donnée dont vous ignorez le type exact car le modèle vous retourne une donnée de type **Object**.

L'intérêt du DataSet typé que nous a généré l'Assistant est qu'il est plus riche qu'un DataSet normal dans la mesure où il contient des objets nommés et dont le type reflète rigoureusement les types des données en base.
Exemple :

- pour manipuler la table de notre DataSet, au lieu d'utiliser un objet retourné par l'écriture **Tables(0**), on va pouvoir directement travailler avec l'objet **DataTable** nommé **Contacts**.
- Pour accéder à une information dans la colonne **Nom**, on va bénéficier d'une propriété d'objet appelée **Nom** dont le type est **String**.

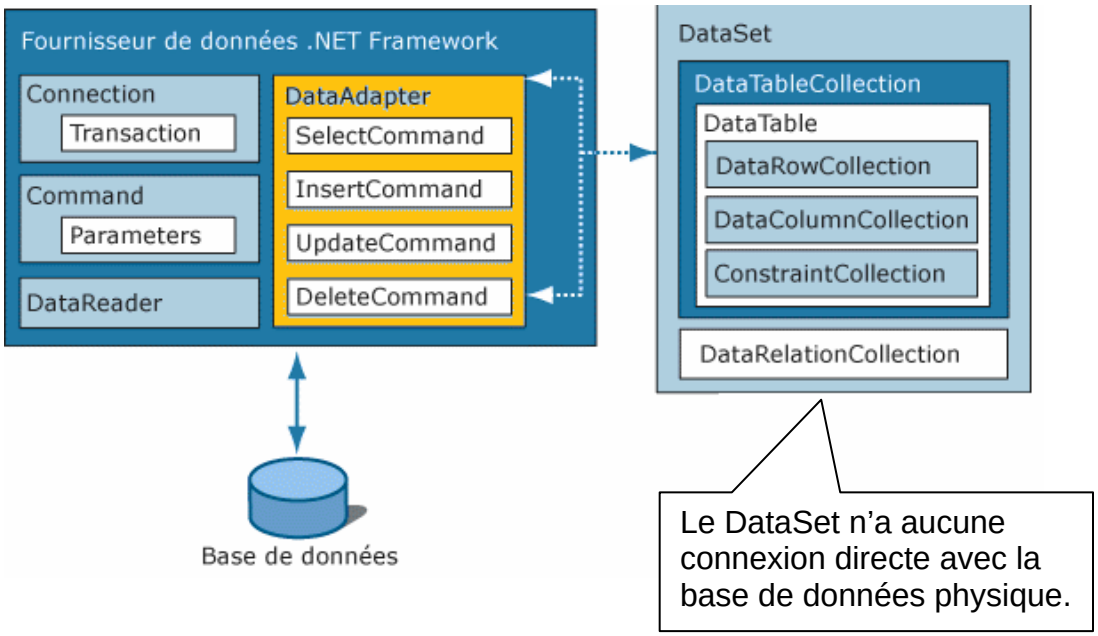
Vous l'avez compris, le code ne va s'en trouver que plus lisible et en plus on va bénéficier de l'aide précieuse de l'IntelliSense et des indications du compilateur en cas d'erreur de manipulation au moment de la compilation !
Moralité, l'Assistant a bien fait les choses ☺…

Pour en savoir plus sur la différence entre un DataSet typé et non typé, rendez-vous sur la rubrique **Groupes de données typés et non typés** de cette page :
http://msdn.microsoft.com/fr-fr/library/8bw9ksd6.aspx

En résumé, le premier objet **Contacts** va donc nous servir à recueillir en mémoire des données de la table **Contacts** en provenance de la source de données.

Bon, et le deuxième objet **ContactsTableAdapter** ?
L'objet **TableAdapter** est celui qui permet de relier la source de données physique à l'objet mémoire **DataSet**. Et oui, car le **DataSet** est indépendant de la source de données. Il ne la connait même pas ! C'est cet autre objet qui fait tout le travail de communication avec la base, à la fois de connexion et à la fois de requêtage pour extraire ou mettre à jour les données.

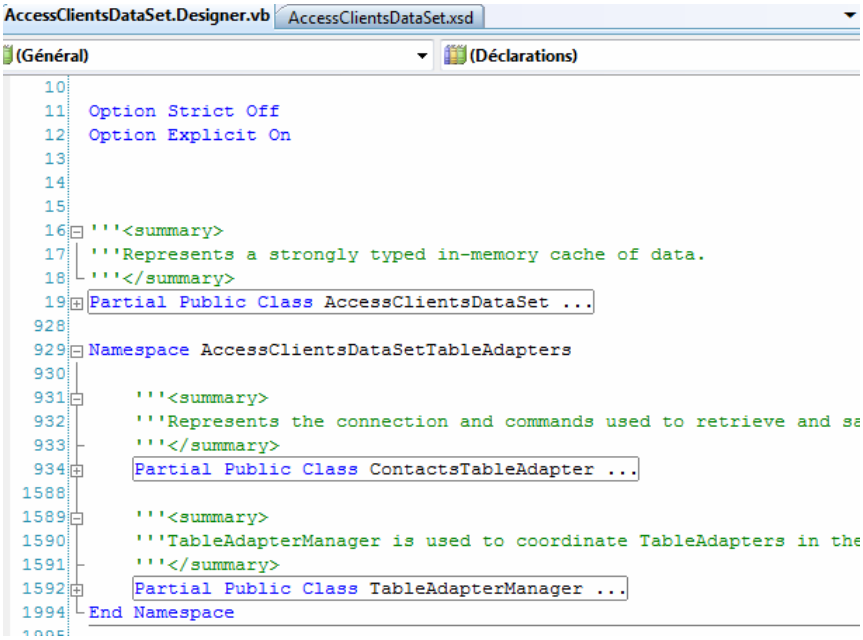
C'est un objet qui comprend 4 sous-objets de commande, chargés de « commander » la base, chacun dédié à un type de requête vers la base. L'objet **SelectCommand** gère l'extraction de données tandis que les trois autres objets s'occupent des autres types de requêtes (**Insert**, **Update** et **Delete**).



Dans le schéma ci-dessus, on parle de ***Data*Adapter** et non de ***Table*Adapter**. Qu'est ce que c'est que cette embrouille encore ?
Dit simplement, retenez qu'un **TableAdapter** est un objet **DataAdapter** mais plus riche que l'objet de base. Il est généré pour nous par l'Assistant Configuration de sources de données pour nous permettre d'aller plus loin en matière d'interaction avec la base de données.

msdn Pour tout savoir sur l'objet **TableAdapter** :
http://msdn.microsoft.com/fr-fr/library/bz9tthwx.aspx

- Si on faisait le parallèle de ce qu'on voit dans le Concepteur avec les objets tels qu'ils sont définis dans le code ? Pour cela, basculez dans le fichier **AccessClientsDataSet.Designer.vb**.
- Utilisez les sigles ⊞ du mode Plan pour réduire les grandes parties du fichier comme suit :



- La classe **AccessClientsDataSet** représente notre **DataSet typé** pour stocker les données dans un objet de type **DataTable**. Les données sont accessibles au travers de la propriété **Contacts** :

La classe **AccessClientsDataSet** hérite de la classe **DataSet** de l'espace de noms **System.Data** du Framework.NET

La classe **AccessClientsDataSet** contient une propriété **Contacts** correspondant à un objet typé **DataTable**.

```
<Global.System.CodeDom.Compiler.GeneratedCodeAttribute("System.Data
  Global.System.Serializable(),  _
  Global.System.ComponentModel.DesignerCategoryAttribute("code"),  _
  Global.System.ComponentModel.ToolboxItem(true),  _
  Global.System.Xml.Serialization.XmlSchemaProviderAttribute("GetTyp
  Global.System.Xml.Serialization.XmlRootAttribute("AccessClientsDat
  Global.System.ComponentModel.Design.HelpKeywordAttribute("vs.data.
Partial Public Class AccessClientsDataSet
    Inherits Global.System.Data.DataSet

    Private tableContacts As ContactsDataTable

    Private _schemaSerializationMode As Global.System.Data.SchemaSe

    Public Sub New ...

    Protected Sub New ...

    <Global.System.Diagnostics.DebuggerNonUserCodeAttribute(),  _
     Global.System.ComponentModel.Browsable(false),  _
     Global.System.ComponentModel.DesignerSerializationVisibility(G
    Public ReadOnly Property Contacts() As ContactsDataTable
        Get
            Return Me.tableContacts
        End Get
    End Property
```

- **ContactsTableAdapter** représente l'objet **TableAdapter** pour discuter avec la table **Contacts**. Cet objet intègre notamment un objet de type **Connection** sous la forme d'une propriété permettant d'établir une connexion sur la base de données :

```
<Global.System.CodeDom.Compiler.GeneratedCodeAttribute("System.Data.Design.TypedI
 Global.System.ComponentModel.DesignerCategoryAttribute("code"),  _
 Global.System.ComponentModel.ToolboxItem(true),  _
 Global.System.ComponentModel.DataObjectAttribute(true),  _
 Global.System.ComponentModel.DesignerAttribute("Microsoft.VSDesigner.DataSource.
    ", Version=8.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"),  _
 Global.System.ComponentModel.Design.HelpKeywordAttribute("vs.data.TableAdapter")
Partial Public Class ContactsTableAdapter
    Inherits Global.System.ComponentModel.Component

    Private WithEvents _adapter As Global.System.Data.OleDb.OleDbDataAdapter

    Private _connection As Global.System.Data.OleDb.OleDbConnection

    Private _transaction As Global.System.Data.OleDb.OleDbTransaction

    Private _commandCollection() As Global.System.Data.OleDb.OleDbCommand

    Private _clearBeforeFill As Boolean

    Public Sub New ...

    Protected Friend ReadOnly Property Adapter ...

    <Global.System.Diagnostics.DebuggerNonUserCodeAttribute()>  _
    Friend Property Connection() As Global.System.Data.OleDb.OleDbConnection
        Get
            If (Me._connection Is Nothing) Then
                Me.InitConnection
```

Si vous faites un clic droit sur le mot **InitConnection** > **Atteindre la définition**, Visual Studio vous emmène sur la définition de la procédure qui récupère par code la chaîne de connexion du fichier de configuration de l'application :

```
<Global.System.Diagnostics.DebuggerNonUserCodeAttribute()>  _
Private Sub InitConnection()
    Me._connection = New Global.System.Data.OleDb.OleDbConnection
    Me._connection.ConnectionString = Global.Coach.Editeur.My.MySettings.Default.AccessClientsConnectionStrir
End Sub
```

Utilisation de l'objet **My** qui permet de retrouver la chaîne de connexion configurée dans le Concepteur de projet.

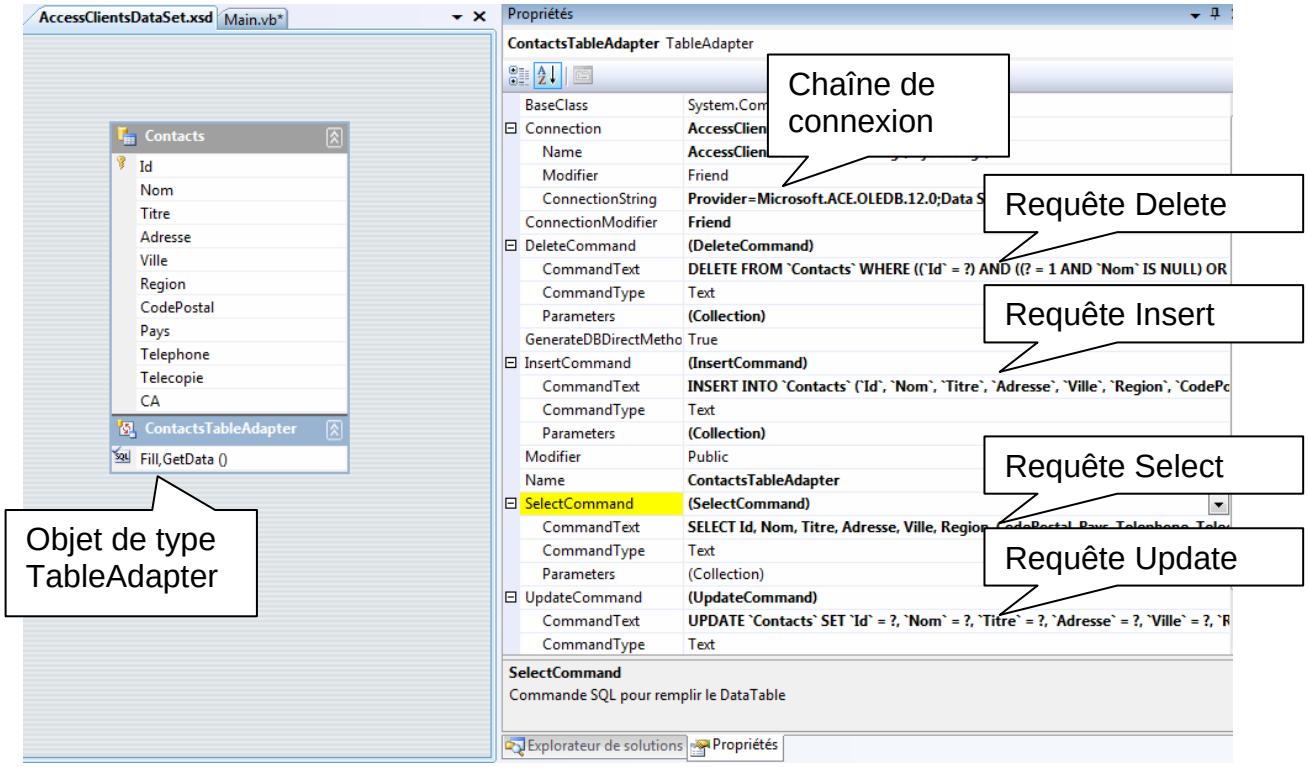
Ce qui va nous intéresser plus encore, ce sont les méthodes de l'objet qui vont nous permettre de travailler sur les données dans la suite de l'atelier, telles que **Fill**, **Update**, **Insert** et **Delete** :

```
1035
1036    Public Property ClearBeforeFill ...
1045
1046    Private Sub InitAdapter ...
1158
1159    Private Sub InitConnection ...
1164
1165    Private Sub InitCommandCollection ...
1174
1175    Public Overloads Overridable Function Fill ...
1186
1187    Public Overloads Overridable Function GetData ...
1196
1197    Public Overloads Overridable Function Update ...
```

La méthode **Fill** par exemple, est celle qu'il va nous falloir utiliser pour extraire les données de la base de données et les charger dans l'objet

DataSet.

Comment la méthode **Fill** fait-elle pour interroger la base de données ?
Elle s'appuie sur l'objet **SelectCommand** contenu dans le **TableAdapter** qui définit la requête **SELECT** à exécuter sur le moteur de base de données.
Pour voir les quatre objets de commande et le détail des requêtes générées de l'objet **ContactsTableAdapter**, sélectionnez l'objet dans le **Concepteur de DataSet** puis tapez la touche **F4** pour faire apparaître la fenêtre de propriétés :



Voilà ! Maintenant nous avons tout ce qu'il nous faut pour accéder à notre base de données Access. Comment ça, vous trouvez ça bien compliqué ?
En réalité il va nous falloir en tout et pour tout 3 malheureuses lignes de code pour extraire les données et les afficher dans la grille…si si, véridique !
Il n'en faudra pas plus pas moins pour mettre à jour la base de données avec les modifications effectuées par l'utilisateur à partir de l'éditeur…

2.3 Afficher les données dans une grille

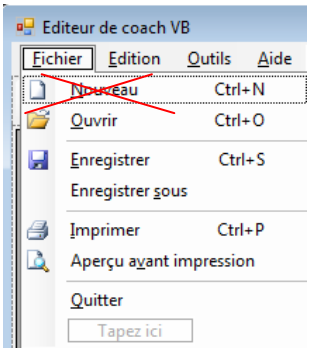
L'objectif de cet exercice est d'extraire les données de la base de données puis de les afficher dans la grille de données de l'éditeur.

Dans cet exercice, vous allez apprendre à :

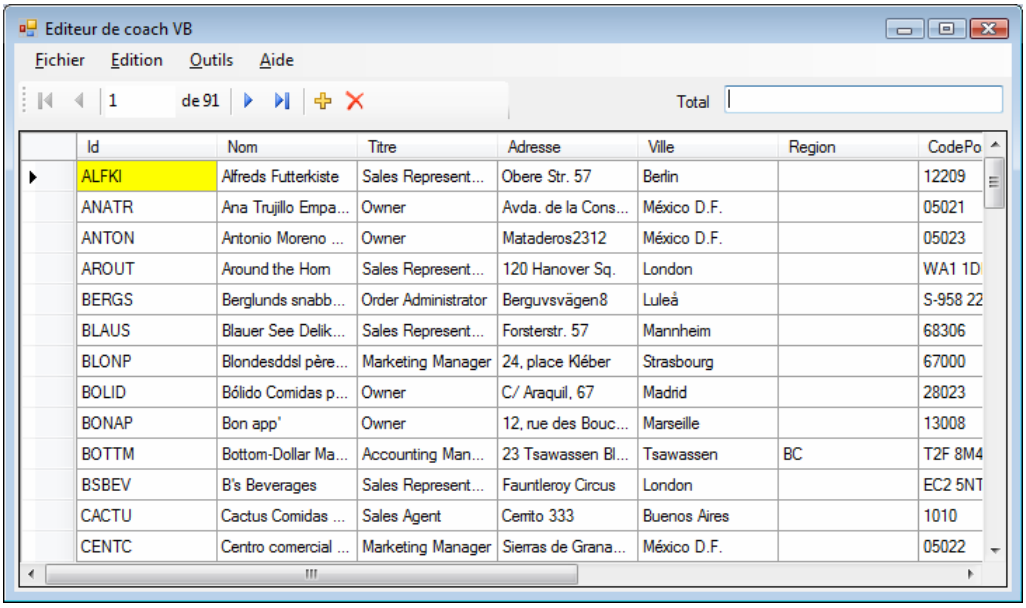
- Utiliser des objets DataSet et TableAdapter,
- Utiliser la méthode Fill d'un TableAdapter pour extraire des données dans une source de données.

Contexte fonctionnel

L'objectif étant d'afficher des données en provenance d'une base de données, nous allons donc supprimer le menu **Fichier** > **Nouveau** de l'application qui n'a plus vraiment de sens.

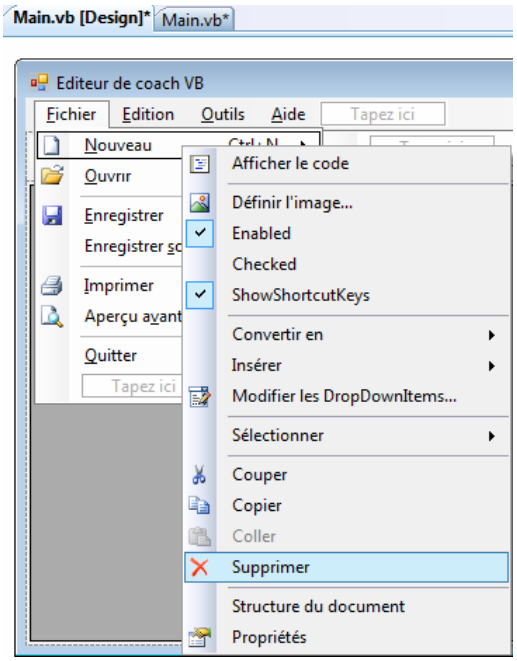


En cliquant le menu **Fichier** > **Ouvrir**, l'utilisateur doit voir s'afficher tous les enregistrements de la base dans la grille de données comme suit :



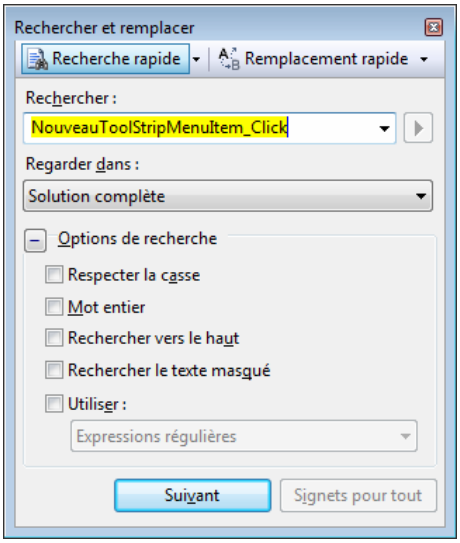
Déroulement de l'exercice :

1. Supprimez l'option de menu Fichier > Nouveau dans le formulaire Main :
   - Ouvrez le formulaire **Main** en mode **Design**.
   - Faites un clic droit sur l'option **Nouveau** du menu **Fichier** puis sélectionnez **Supprimer** pour détruire l'option :



- Basculez dans le code du fichier en faisant un clic droit sur la surface du **Concepteur de formulaire** > **Afficher le code**.
- Repérez le gestionnaire d'évènement **NouveauToolStripMenuItem_Click** dans la classe **Main**.

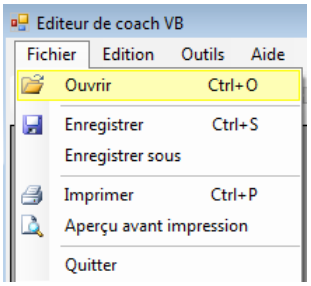
Pensez à utiliser la fonction de recherche de Visual Studio pour retrouver un mot dans le code du projet. Tapez les touches **CTRL + F** pour faire apparaître la fenêtre de recherche :



- Sélectionnez l'intégralité de la procédure **NouveauToolStripMenuItem_Click** et supprimez-la :

```
Main.vb [Design]  Main.vb
Main                                        (Déclarations)
10          mainNotifyIcon.Icon = My.Resources.lan_connected
11      End Sub
12  Déclaration des variables privées de la classe
30  Gestion de la table de données en mémoire
89  Traitement de la fenêtre d'options
149 #Region "Traitement des options de menu"
150     Private Sub Main_SizeChanged ...
155     Private Sub OptionsToolStripMenuItem_Click ...
166     Private Sub FormClose ...
172     Private Sub MaximiserLaFenêtreToolStripMenuItem_Click ...
175     Private Sub RestaurerLaFenêtreToolStripMenuItem_Click ...
178     Private Sub MinimiserLaFenêtreToolStripMenuItem_Click ...
181     Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
182                                                ByVal e As System.EventArgs)
183
184         'Création d'une nouvelle table mémoire vide
185         Dim newDataTable As DataTable
186         newDataTable = CreateDataTable()
187         'Configuration du gestionnaire de liaison sur la source de données
188         MainBindingSource.DataSource = newDataTable
189         'Liaison du gestionnaire avec les contrôles d'affichage de données
190         MainDataGridView.DataSource = MainBindingSource
191         MainBindingNavigator.BindingSource = MainBindingSource
192         'Mettre à jour le chemin de fichier à vide
193         DataFilePath = String.Empty
194         'Remettre à jour le titre du formulaire
195         UpdateFormTitle(DataFilePath)
196     End Sub
```

2. Testez la suppression de l'option de menu :
   - Compilez le projet pour vérifier que la génération se déroule sans souci.
   - Exécutez l'application (CTRL F5) pour vérifier que l'option **Fichier** > **Nouveau** n'existe plus.



   - Fermez l'application.

3. Modifiez maintenant le code de l'option Fichier > Ouvrir pour qu'il extrait les données non plus d'un fichier texte mais de la base de données Microsoft Office Access, pour les afficher dans la grille MainDataGridView du formulaire :
   - Repérez le gestionnaire d'évènement **OuvrirToolStripMenuItem_Click** dans la classe **Main**.

```
182     Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
183                                               ByVal e As System.EventArgs) _
184                                               Handles OuvrirToolStripMenuItem.Click
185         'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture standard
186         Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
187         openDataFileDialog.Filter = "Fichiers coach|*.coach"
188         openDataFileDialog.InitialDirectory = SaveDirectoryPath
189         If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
190             'Code d'ouverture du fichier sélectionné
191             Dim newDataTable As DataTable
192             newDataTable = ReadFile(openDataFileDialog.FileName)
193             'Configuration du gestionnaire de liaison sur la source de données
194             MainBindingSource.DataSource = newDataTable
195             'Liaison du gestionnaire avec les contrôles d'affichage de données
196             MainDataGridView.DataSource = MainBindingSource
197             MainBindingNavigator.BindingSource = MainBindingSource
198             'Mémorisation du chemin du fichier pour une éventuelle sauvegarde
199             DataFilePath = openDataFileDialog.FileName
200             'Affichage du nom du fichier dans la barre de titre du formulaire
201             UpdateFormTitle(DataFilePath)
202         End If
203     End Sub
```

- Supprimez les trois premières lignes du code qui configurent la boîte de dialogue d'ouverture de fichier de Windows.
- Supprimez également la structure de boucle **If…Then…End If** qui entoure le code de lecture puis le chargement des données.
- Supprimez enfin les deux lignes de création puis de chargement de la table mémoire **newDataTable** de type **DataTable** que nous allons remplacer par le code d'extraction des données de la base Access.
- Les deux dernières lignes de code de la procédure visant à mémoriser le chemin du fichier et à mettre à jour le titre de la fenêtre en correspondance ne sont plus non plus d'actualité.
- Cela donne :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Sélection du fichier à l'aide de la boîte de dialogue d'ouverture standard
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers coach|*.coach"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
        'Code d'ouverture du fichier sélectionné
        Dim newDataTable As DataTable
        newDataTable = ReadFile(openDataFileDialog.FileName)
        'Configuration du gestionnaire de liaison sur la source de données
        MainBindingSource.DataSource = newDataTable
        'Liaison du gestionnaire avec les contrôles d'affichage de données
        MainDataGridView.DataSource = MainBindingSource
        MainBindingNavigator.BindingSource = MainBindingSource
        'Mémorisation du chemin du fichier pour une éventuelle sauvegarde
        DataFilePath = openDataFileDialog.FileName
        'Affichage du nom du fichier dans la barre de titre du formulaire
        UpdateFormTitle(DataFilePath)
    End If
```

```
    End Sub
```

- Il ne doit rester que le code de liaison de la structure de données mémoire avec la grille et le gestionnaire de liaison du formulaire. Vous devez donc obtenir :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click

    'Lecture des données dans la base de don

    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.Binding Source = MainBindingSource
End Sub
```

Futur emplacement du code d'extraction des données de la base

Code de liaison des données mémoire avec les contrôles d'affichage qu'il faut conserver.

De quoi avons-nous besoin ? Sur la base de ce que nous avons vu à l'exercice précédent, il nous faut :

- L'objet **ContactsTableAdapter** pour envoyer l'ordre d'extraction des données à la base de données.
- L'objet **Contacts** pour stocker les données en mémoire en retour de l'extraction.

Ensuite, il ne nous restera qu'à charger les contrôles d'affichage en s'appuyant sur le mécanisme de **liaison de données** (databinding) que nous avons vu à l'atelier précédent. Et oui, on retombe exactement sur le même fonctionnement puisque l'objet **Contacts** dans lequel nous allons charger les données est aussi un objet de type **DataTable** au même titre que l'objet **newDataTable** que nous avions utilisé pour lire les données du fichier texte.

- Déclarez une variable nommée **ClientsDataSet** de type **AccessClientsDataSet** juste après la ligne de définition de la classe **Main** comme suit :

Code VB

```
Public Class Main
    Dim ClientsDataSet As AccessClientsDataSet
…
```

D'où sort le type de données **AccessClientsDataSet** ?
C'est le nom de la classe générée par l'Assistant Configuration de source de données que nous avons observée dans le fichier **AccessClientsDataSet.Designer.vb** à l'exercice précédent.

Pourquoi définir l'objet DataSet au niveau de la classe ?
Tout simplement parce qu'il doit perdurer sur toute la durée de vie du programme. N'oubliez pas qu'il va héberger les données extraites de la base et que ces données ont pour vocation d'être manipulées au travers de l'interface.
Attention ! Cela ne veut pas dire que la connexion à la base de données elle aussi perdure sur toute la durée de vie de l'application. Rappelez-vous que c'est l'objet **TableAdapter** qui a en charge de gérer la connexion pour discuter avec la base. Une fois les données rapatriées ou les ordres de mise à jour exécutés, la connexion est immédiatement relâchée ! C'est pour cela qu'on parle de mode *déconnecté*.
Pendant ce temps le **DataSet** peut mener sa petite vie tranquille sans se préoccuper de rien…

- Repérez le constructeur de la classe **Main** nommé **New** :

```
4       Public Sub New()
5
6           ' Cet appel est requis par le Concepteur Windows Form.
7           InitializeComponent()
8
9           ' Ajoutez une initialisation quelconque après l'appel InitializeC
10          mainNotifyIcon.Icon = My.Resources.lan_connected
11      End Sub
```

- Ajoutez à la fin du constructeur l'ordre de création de l'objet **ClientsDataSet** en invoquant le constructeur de la classe **AccessClientsDataSet** :

Code VB

```
Public Class Main
…
  Public Sub New()

      ' Cet appel est requis par le Concepteur Windows Form.
      InitializeComponent()

      ' Ajoutez une initialisation quelconque après l'appel InitializeComponent().
      mainNotifyIcon.Icon = My.Resources.lan_connected

      'Instanciation du DataSet qui va servir à travailler sur les données
      ClientsDataSet = New AccessClientsDataSet()
  End Sub
…
```

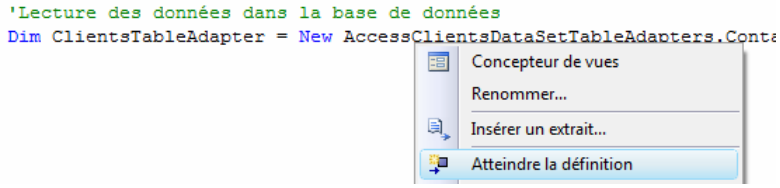
- Revenons sur le code de la procédure **OuvrirToolStripMenuItem_Click**.

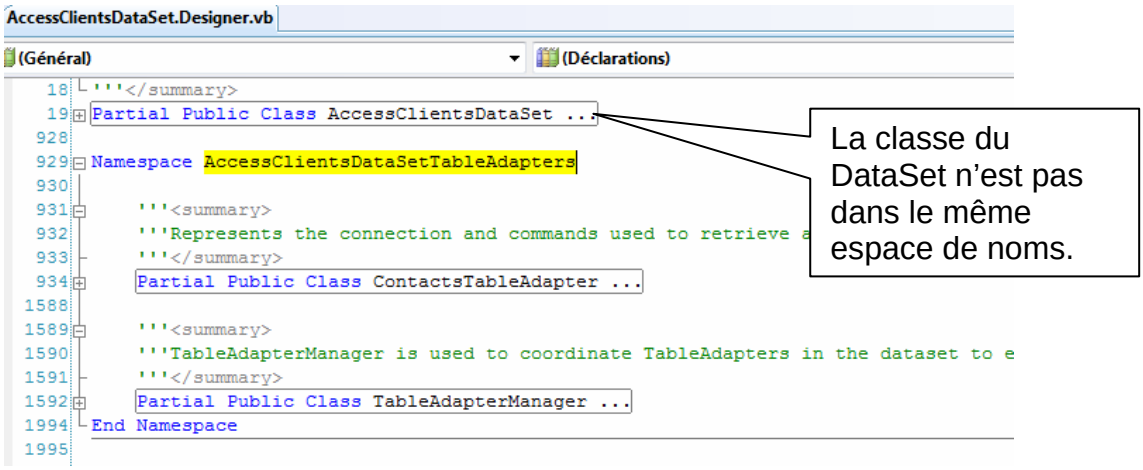
- Déclarez un objet nommé **ClientsTableAdapter** de type **ContactsTableAdapter** :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Lecture des données dans la base de données
    Dim ClientsTableAdapter = new _
            AccessClientsDataSetTableAdapters.ContactsTableAdapter()

    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.BindingSource = MainBindingSource
End Sub
```

Pourquoi précédez le nom de la classe **ContactsTableAdapter** du nom **AccessClientsDataSetTableAdapters** ?
AccessClientsDataSetTableAdapters est l'espace de noms dans lequel a été généré la classe **ContactsTableAdapter**. Vous pouvez retrouver sa définition en faisant un clic droit sur le nom > **Atteindre la définition**.





Cet espace de noms n'a d'autre objectif que de séparer l'objet **TableAdapter** des autres objets du groupe de données. La classe **DataSet** est elle-même placée dans l'espace de noms racine du projet.

- Utilisez la méthode **Fill** de l'objet **ClientsTableAdapter** pour remplir la table **Contacts** de l'objet **ClientsDataSet** avec les données en provenance de la base :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Lecture des données dans la base de données
    Dim ClientsTableAdapter = new _
            AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    'Extraction des données et chargement du DataSet
    ClientsTableAdapter.Fill(ClientsDataSet.Contacts)
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.BindingSource = MainBindingSource
  End Sub
```

N'oubliez pas qu'un **DataSet** en lui-même ne contient pas de données directement. Les données sont en fait chargées dans la collection de **DataTable** qu'il contient.
Dans notre cas, le **DataSet** ne contient qu'une seule table (Contacts) et comme il est typé nous pouvons accéder directement à l'objet **DataTable** via l'écriture **ClientsDataSet.Contacts**. Pratique non ?

Peut-être vous attendiez-vous à une écriture du type :
DataTable en retour = TableAdapter.Fill() ?
Pourquoi pas mais il se trouve que la méthode **Fill** attend le conteneur de données en paramètre de la procédure. Le paramètre est passé **par référence** c'est-à-dire que les modifications effectuées dans le traitement de la procédure sont directement appliquées à l'objet initial passé en paramètre.

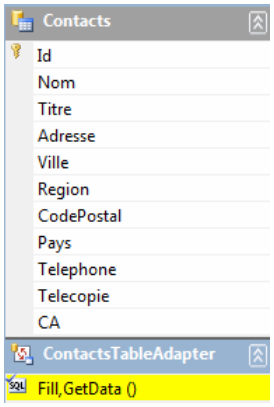
Que fait la méthode **Fill** concrètement ?
La méthode **Fill** récupère des lignes de la source de données à l'aide de l'instruction **SELECT** définie dans l'objet **SelectCommand**.

Pourquoi est-ce qu'il ne faut pas ouvrir une connexion sur la base de données avant d'appeler la méthode **Fill** ?
Tout simplement, vous vous en doutez, parce que c'est la méthode qui s'occupe de tout ☺. Si une connexion est déjà ouverte avant l'appel à la méthode **Fill**, elle est utilisée. Sinon, elle est automatiquement ouverte le temps de récupérer les données. La connexion est aussitôt fermée une fois les données rapatriées dans le DataSet. Si la connexion était ouverte au préalable, elle reste ouverte.

Par défaut un **TableAdapter** comprend deux méthodes pour extraire les données d'une base de données afin de les insérer dans un **DataTable** :

- La méthode **Fill**, qui utilise un **DataTable** existant comme paramètre et le remplit.
- La méthode **GetData** qui renvoie un nouveau **DataTable** déjà rempli.

Dans notre cas, c'est bien la méthode **Fill** la plus appropriée puisque notre **DataSet** typé contient un objet **DataTable** tout prêt pour réceptionner les données.



Pour tout savoir sur la méthode **Fill** :
http://msdn.microsoft.com/fr-fr/library/377a8x4t(VS.80).aspx
Pour en savoir plus sur comment remplir de données un groupe de données
http://msdn.microsoft.com/fr-fr/library/ms171920(VS.80).aspx

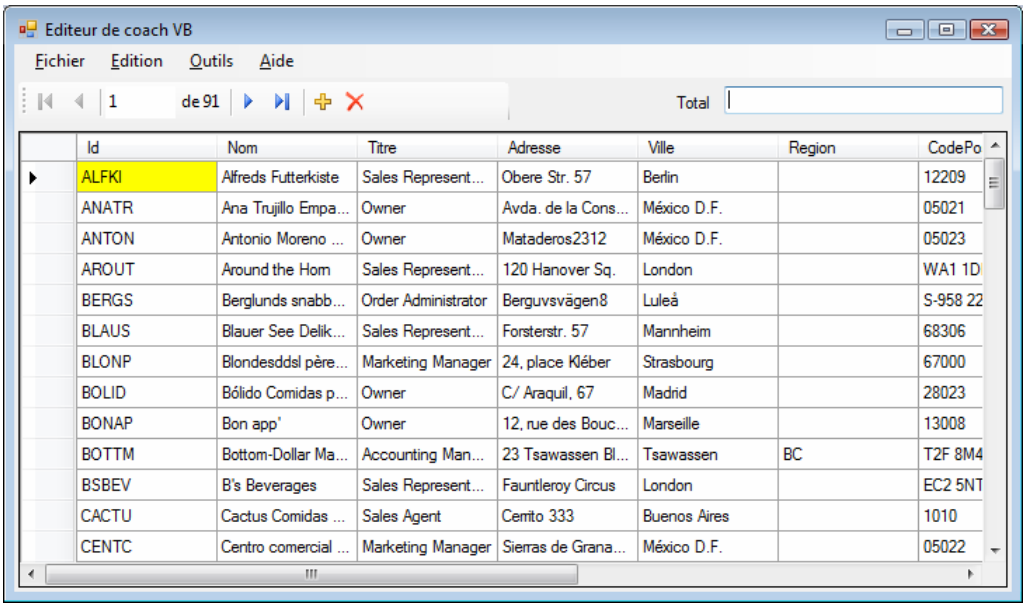
Que nous reste-il à faire ?
Il ne reste qu'à afficher les données dans les contrôles d'affichage en utilisant le mécanisme de liaison de données (databinding). Les contrôles d'affichage doivent être liés à une source de données telle qu'un objet **DataTable**. Et bien on l'a cet objet ! C'est **ClientsDataSet.Contacts** que vous venez de remplir à l'aide du **TableAdapter**.

- Corrigez le databinding de la grille de données pour qu'elle soit liée à la table **ClientsDataSet.Contacts** au lieu de l'objet **newDataTable** initial :

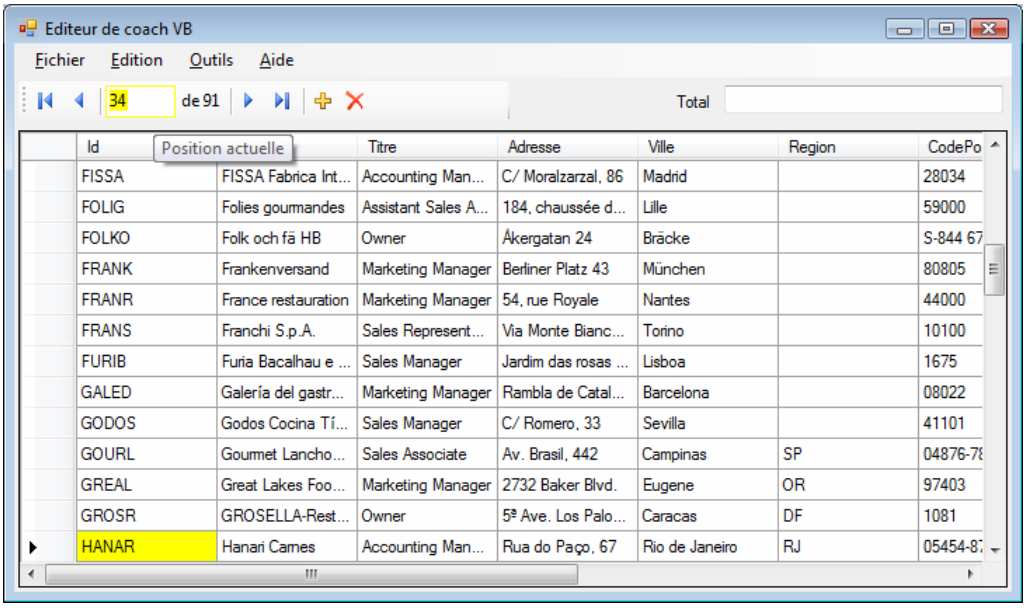
Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Lecture des données dans la base de données
    Dim ClientsTableAdapter = new _
            AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    'Extraction des données et chargement du DataSet
    ClientsTableAdapter.Fill(ClientsDataSet.Contacts)
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable ClientsDataSet.Contacts
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.BindingSource = MainBindingSource
End Sub
```

4. Testez le chargement des données dans l'Editeur :
   - Exécutez l'application en cliquant sur ▶ (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir**.

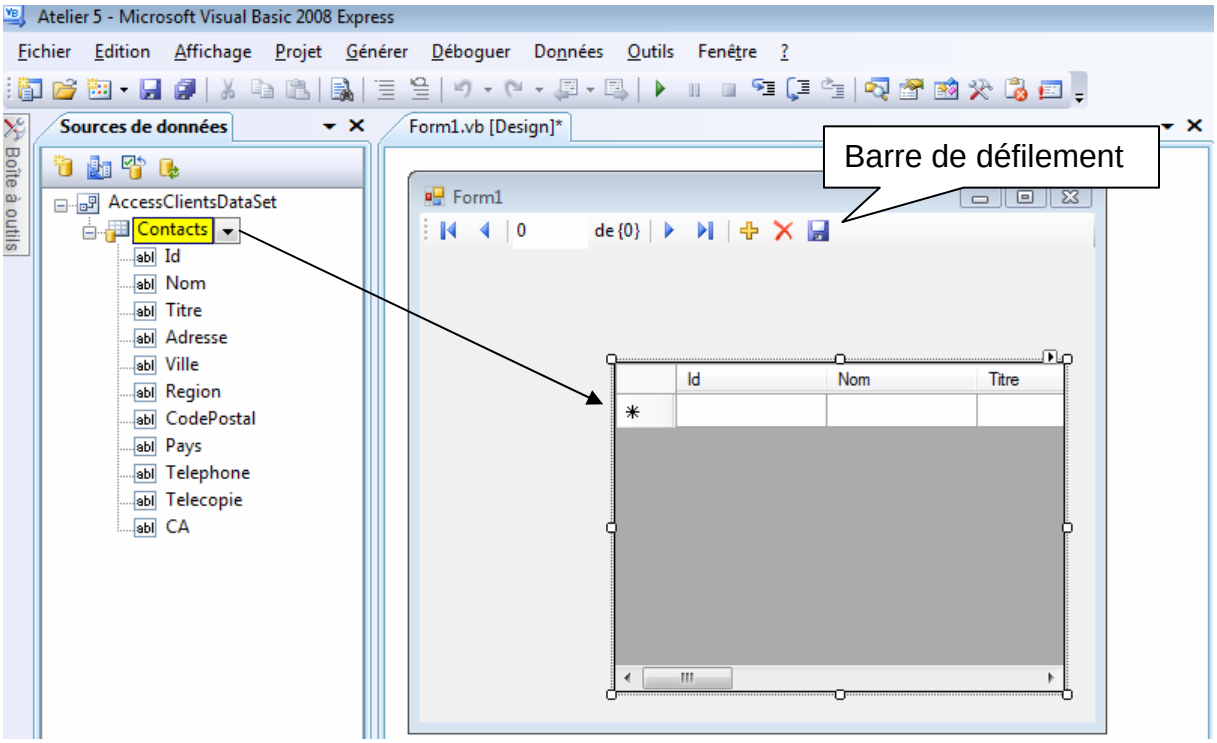


- Vérifiez que le gestionnaire de liaison fonctionne correctement en utilisant les flèches et /ou la zone de texte de positionnement.



Alors ? Je ne vous avais pas dit que le code tenait en trois lignes ? Un jeu d'enfant…

En fait, il y a encore plus court en utilisant le menu **Données** > **Afficher la source de données** de Visual Studio. Il suffit de faire glisser les éléments (par exemple la table Contacts) de la fenêtre sur la surface de design d'un formulaire et automatiquement la grille est générée avec tous les objets associés.



msdn Pour en savoir plus sur la **fenêtre Source de données** :
http://msdn.microsoft.com/fr-fr/library/6ckyxa83(VS.80).aspx

2.4 Mettre à jour les données

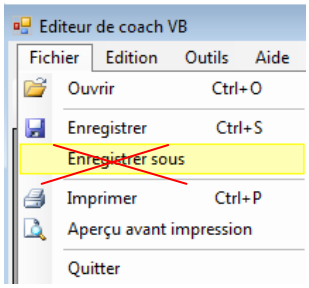
L'objectif de cet exercice est de mettre à jour la base de données avec les modifications apportées aux données par l'utilisateur dans l'interface.

Dans cet exercice, vous allez apprendre à :

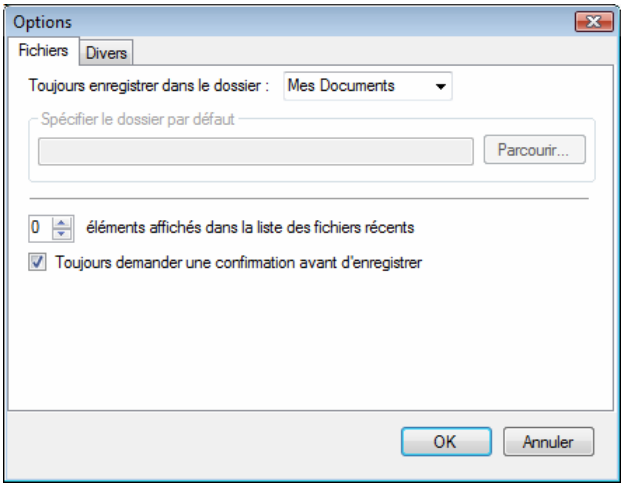
- Utiliser la méthode Update d'un DataAdapter pour reporter des modifications de données dans une source de données,
- Utiliser la méthode RejectChanges d'un DataSet pour annuler des modifications faites avec les contrôles d'affichage,
- Utiliser une boucle de décision Try…Catch…End Try.

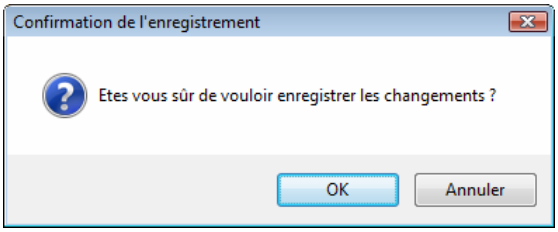
Contexte fonctionnel

L'option de menu **Fichier** > **Enregistrer sous** n'étant plus applicable, nous allons donc la supprimer à son tour.



La sauvegarde des modifications est effectuée lorsque l'utilisateur sélectionne le menu **Fichier** > **Enregistrer**.
Nous allons utiliser l'option **Toujours demander une confirmation avant d'enregistrer** de la boîte d'**Options** de l'application pour, si elle est cochée, proposer un message de confirmation à l'utilisateur lui permettant d'annuler ses modifications en cas de besoin.

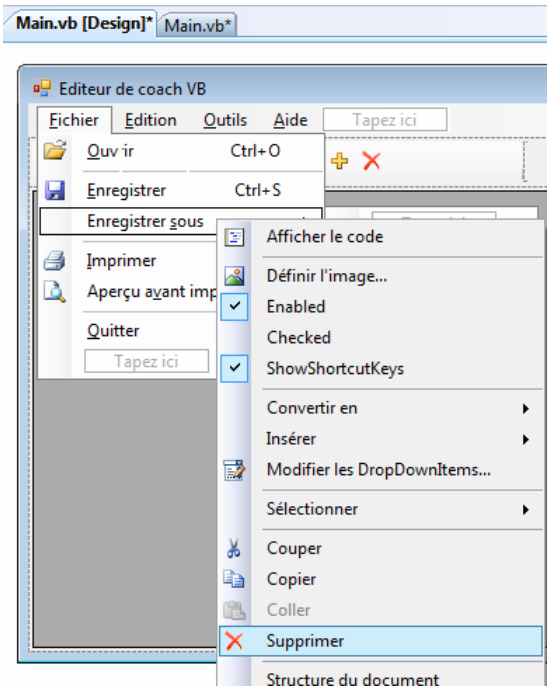




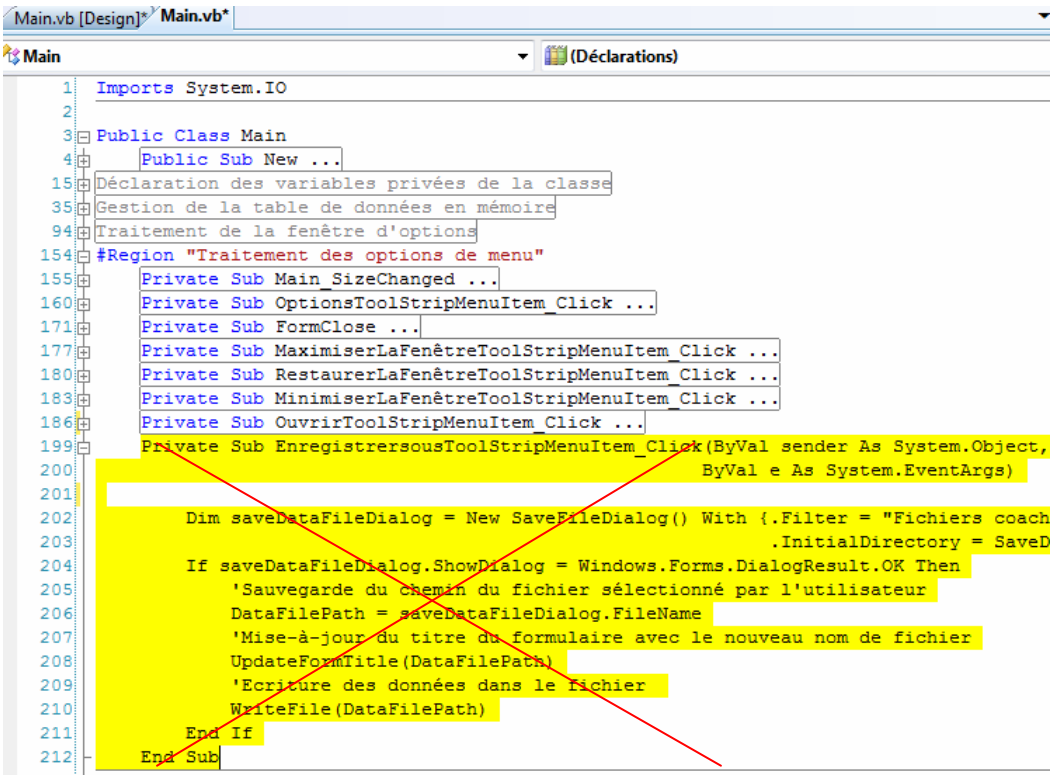
Si l'utilisateur clique **Annuler**, les changements ne sont pas enregistrés dans la base de données et la grille est réinitialisée avec les valeurs initiales.

Déroulement de l'exercice :

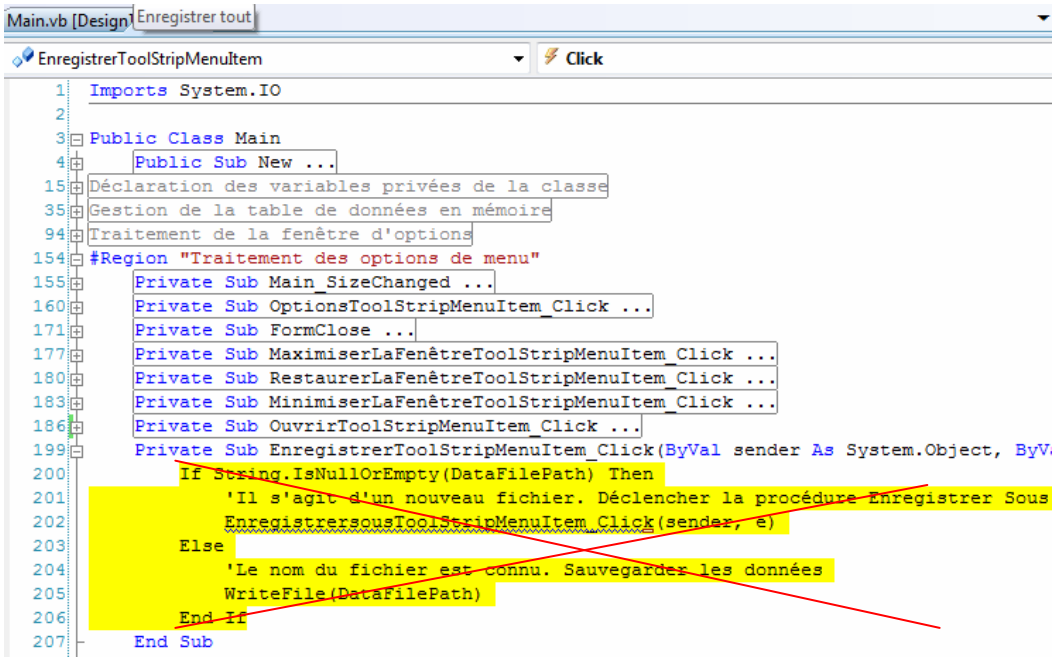
1. Supprimez l'option de menu Fichier > Enregistrer sous dans le formulaire Main:
   - Ouvrez le formulaire **Main** en mode **Design**.
   - Faites un clic droit sur l'option **Enregistrer sous** du menu **Fichier** puis sélectionnez **Supprimer** pour détruire l'option :



- Basculez dans le code du fichier en faisant un clic droit sur la surface du **Concepteur de formulaire** > **Afficher le code**.
- Repérez le gestionnaire d'évènement **EnregistrersousToolStripMenuItem_Click** dans la classe **Main**.
- Sélectionnez l'intégralité de la procédure **EnregistrersousToolStripMenuItem_Click** et supprimez-la :



- Repérez le gestionnaire d'évènement **EnregistrerToolStripMenuItem_Click** dans la classe **Main** qui correspond au clic sur le menu **Fichier** > **Enregistrer**.
- Supprimez le contenu de la procédure que nous allons remplacer par le code de mise-à-jour des données dans la base de données Access :

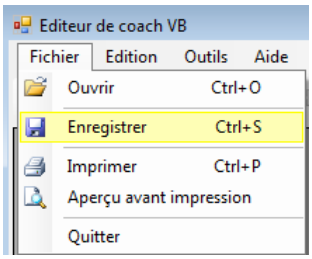


- Vous obtenez :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click(…) _
                Handles EnregistrerToolStripMenuItem.Click
    'Mise-à-jour des données dans la base de données

End Sub
```

2. Testez la suppression de l'option de menu :
   - Compilez le projet pour vérifier que la génération se déroule sans souci.
   - Exécutez l'application (CTRL F5) pour vérifier que l'option **Fichier** > **Enregistrer sous** n'existe plus.



- Fermez l'application.

Que va-t-il falloir faire pour enregistrer les données ?

Est-ce que vous vous souvenez comment fonctionne la grille de données ?
Nous avons vu à l'atelier 4 précédent que la grille de données est capable d'enregistrer les modifications effectuées par l'utilisateur automatiquement dans sa source de données (DataTable). C'est le principe de la liaison de données (ou databinding) que l'on dit bidirectionnel puisque cela marche aussi bien dans un sens que dans l'autre.

Donc, c'est dans l'objet **DataTable** qui constitue la source de données de la grille qu'il faut récupérer les changements pour les appliquer à la base de données.

Et pour discuter avec la base il nous faut …?
L'objet de type **TableAdapter** qui propose une méthode nommée **Update** pour répercuter les changements dans la base de données.

Alors allons-y !

3. Coder la mise-à-jour des changements opérés par l'utilisateur dans la grille de données dans la base Access :
   - Au début de la procédure **EnregistrerToolStripMenuItem_Click**, déclarez un objet de type **ContactsTableAdapter** comme suit :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
    'Mise-à-jour des données dans la base de données
    Dim ClientsTableAdapter = _
            New  AccessClientsDataSetTableAdapters.ContactsTableAdapter()
End Sub
```

   - Demandez au gestionnaire de source de données **MainBindingSource** d'appliquer toutes les modifications en attente dans la grille d'affichage dans la source de données (DataTable). Pour cela utilisez la méthode **EndEdit** du gestionnaire :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
    'Mise-à-jour des données dans la base de données
    Dim ClientsTableAdapter = _
            New AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    'Appliquer les changements des contrôles d'affichage
    'dans la source de données sous-jacente
    MainBindingSource.EndEdit()
End Sub
```

Pour tout savoir sur la méthode **EndEdit** de l'objet **BindingSource** :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.bindingsource.endedit(VS.85).aspx

- Vérifiez que des changements se sont produits en utilisant la propriété **HasChanges** de la source de données **ClientsDataSet** :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
    'Mise-à-jour des données dans la base de données
    Dim ClientsTableAdapter = _
            New AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    'Appliquer les changements des contrôles d'affichage
    'dans la source de données sous-jacente
    MainBindingSource.EndEdit()
    'Vérifiez que des modifications ont eu lieu
    If ClientsDataSet.HasChanges Then

    End If
  End Sub
```

- Utilisez la méthode **Update** de l'objet **ContactsTableAdapter** pour mettre à jour les changements dans la base de données :

Code VB

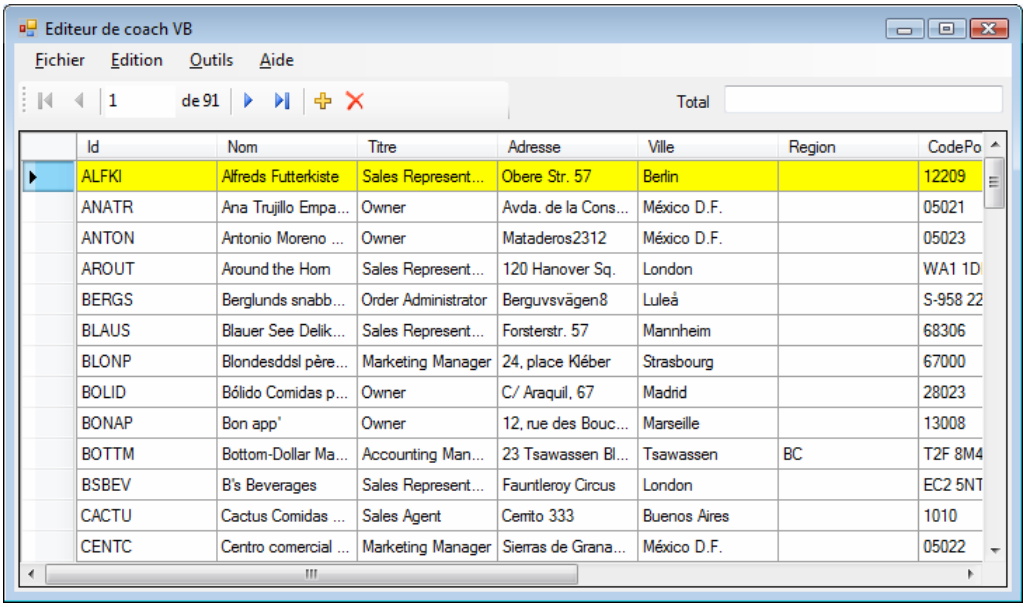
```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
    'Mise-à-jour des données dans la base de données
    Dim ClientsTableAdapter = _
            New AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    'Appliquer les changements des contrôles d'affichage
    'dans la source de données sous-jacente
    MainBindingSource.EndEdit()
    'Vérifiez que des modifications ont eu lieu
    If ClientsDataSet.HasChanges Then
       'Appliquer les changements dans la base de données
       ClientsTableAdapter.Update(ClientsDataSet.Contacts)
    End If
  End Sub
```

Que fait la méthode **Update** concrètement ?
Elle analyse chaque modification apportée dans la source de données passée en paramètre et exécute la commande appropriée pour propager la ligne modifiée à la base de données :

- s'il s'agit d'une nouvelle ligne, elle exécute la commande **INSERT**,
- s'il s'agit d'une suppression de ligne, elle exécute la commande **DELETE**,
- et s'il s'agit d'une ligne ayant subi une modification, elle exécute la commande **UDPATE**.

Pour tout savoir sur la méthode **Update**:
http://msdn.microsoft.com/fr-fr/library/33y2221y(VS.80).aspx
ou encore ce lien :
http://msdn.microsoft.com/fr-fr/library/system.data.common.dataadapter.update.aspx

4. Testez le fonctionnement de la mise à jour :
   - Exécutez l'application en cliquant sur ▶ (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir** pour charger les données de la base Access dans la grille d'affichage.

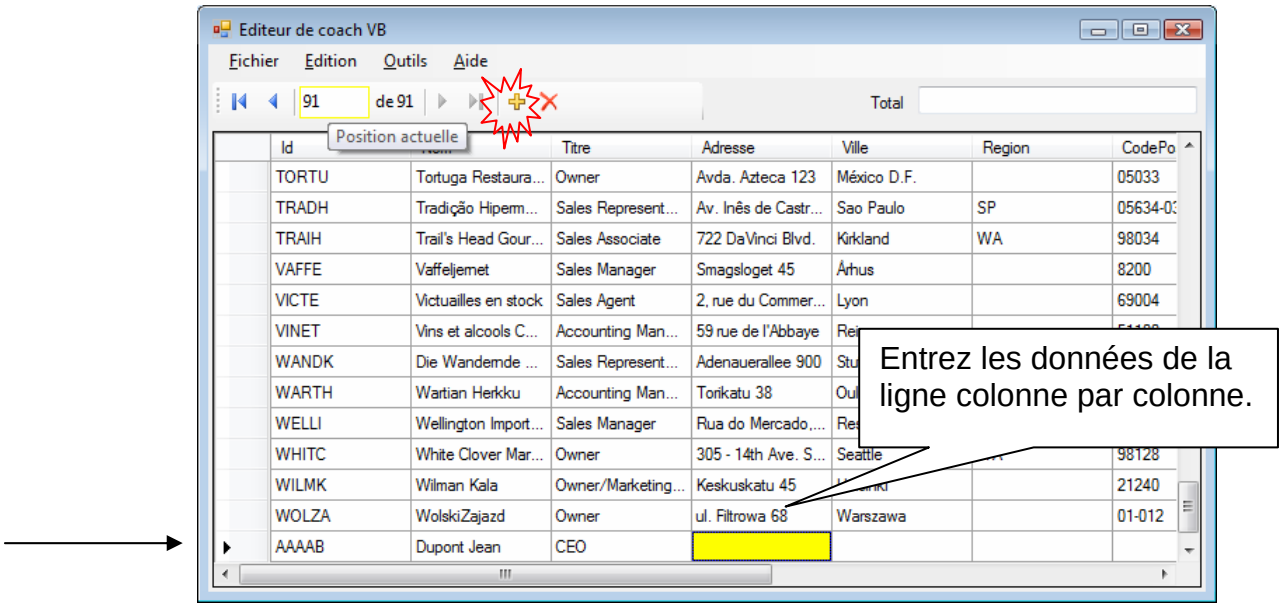


Id	Nom	Titre	Adresse	Ville	Region	CodePo
ALFKI	Alfreds Futterkiste	Sales Represent...	Obere Str. 57	Berlin		12209
ANATR	Ana Trujillo Empa...	Owner	Avda. de la Cons...	México D.F.		05021
ANTON	Antonio Moreno ...	Owner	Mataderos2312	México D.F.		05023
AROUT	Around the Horn	Sales Represent...	120 Hanover Sq.	London		WA1 1D
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	Bólido 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

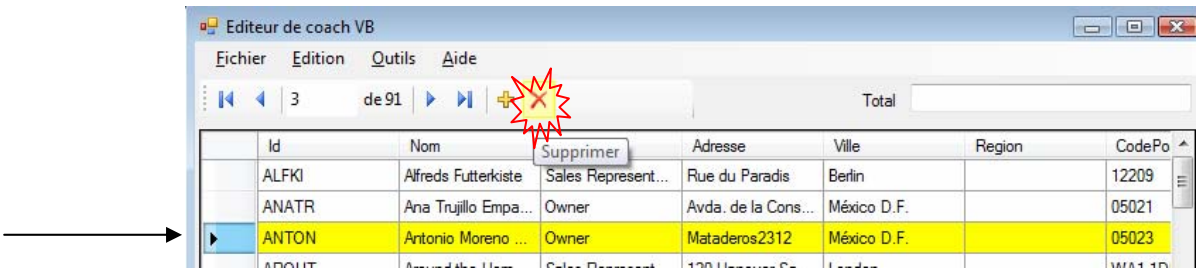
- Modifiez le contenu d'une ligne existante.

Id	Nom	Titre	Adresse			
ALFKI	Alfreds Futterkiste	Sales Represent...	Rue du Paradis	Berlin		12209
ANATR	Ana Trujillo Empa...	Owner	Avda. de la Cons...	México D.F.		05021

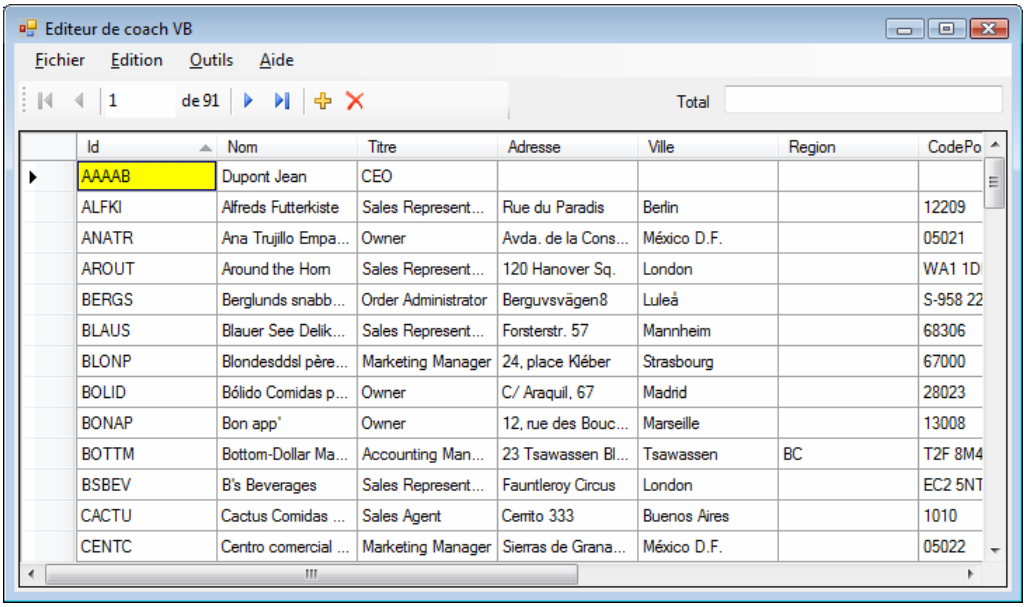
- Insérez une nouvelle ligne en cliquant ✚ dans la barre de navigation de l'éditeur.



- Supprimez une ligne en cliquant ✕.



- Cliquez le menu **Fichier** > **Enregistrer**.
- Cliquez le menu **Fichier** > **Ouvrir** pour recharger les données en provenance de la base de données. Vérifiez que vos changements ont été retenus.



Pour être sûr, vous pouvez fermer et rouvrir l'application mais faites attention de ne pas la recompiler sinon pour rappel, Visual Studio réinitialise le fichier de données dans le répertoire de sortie à partir de la copie dans le répertoire racine du projet !

La mise à jour de données est plus délicate que l'extraction dans le sens où des erreurs peuvent se produire, voire des conflits.
Pour éviter que des messages intempestifs apparaissent à l'utilisateur, il faut gérer les erreurs potentielles dans le code. Pour cela, on utiliser une structure de décision **Try…Catch…End Try** :

- Le principe consiste à précéder le bloc d'instructions dans lequel une erreur peut se produire de l'instruction **Try**.
- Ensuite on peut ajouter autant de bloc **Catch** que de type d'exception générée par le système, à condition que vous sachiez les anticiper. Si vous ne savez pas à l'avance le type d'erreur possible, alors un seul bloc **Catch** fera l'affaire.

C'est quoi une **Exception** ?
Lorsqu'une erreur se produit, le Framework .NET génère une sorte d'alerte pour prévenir l'application qu'un problème est survenu pendant l'exécution. Cette erreur peut être analysée dans le code à l'aide d'un objet qui détaille précisément le problème (source de l'erreur, message de l'erreur etc…) qu'on appelle Exception. D'autres classes dérivées de la classe de base caractérisent des types d'exception particuliers (System.OverflowException, System.IO.FileNotFoundException etc…)

Pour en savoir plus sur la classe **Exception** :
http://msdn.microsoft.com/fr-fr/library/system.exception(VS.80).aspx
Pour savoir **comment intercepter une exception dans Visual Basic** :
http://msdn.microsoft.com/fr-fr/library/ys1b32h3(VS.80).aspx

5. Encadrez le code de mise à jour dans un bloc Try…Catch :
   - Ajoutez l'instruction **Try…End Try** au début et à la fin de la procédure comme suit :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
    Dim ClientsTableAdapter = _
            New AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    Try
        'Mise-à-jour des données dans la base de données
        'Appliquer les changements des contrôles …
        MainBindingSource.EndEdit()
        'Vérifiez que des modifications ont eu lieu
        If ClientsDataSet.HasChanges Then
            'Appliquer les changements dans la base de données
            ClientsTableAdapter.Update(ClientsDataSet.Contacts)
        End If
```

```
        End Try
End Sub
```

Un bon réflexe est de laisser les déclarations de variable (par exemple ici celle de l'objet TableAdapter) à l'extérieur du bloc **Try** car tout ce que vous déclarez dans un bloc **Try** n'est accessible qu'à l'intérieur du bloc.

- Ajoutez un bloc **Catch** comme suit :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
    Dim ClientsTableAdapter = _
            New AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    Try
        'Mise-à-jour des données dans la base de données
        'Appliquer les changements des contrôles …
        MainBindingSource.EndEdit()
        'Vérifiez que des modifications ont eu lieu
        If ClientsDataSet.HasChanges Then
            'Appliquer les changements dans la base de données
            ClientsTableAdapter.Update(ClientsDataSet.Contacts)
        End If
     Catch ex As Exception

     End Try
End Sub
```

Logiquement il faudrait maintenant rechercher la (ou les) ligne(s) du **DataTable** qui a (ont) provoqué l'erreur. Nous allons nous contenter ici d'afficher un message à l'utilisateur.

Pour savoir comment trouver les lignes d'un DataSet contenant des erreurs :
http://msdn.microsoft.com/fr-fr/library/bb747e9b(VS.80).aspx

- Ajoutez un message à l'attention de l'utilisateur en utilisant la méthode **Show** de la classe **MessageBox** du Framework .NET et l'objet **ex** de type **Exception** qui détaille l'erreur :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
    Dim ClientsTableAdapter = _
            New AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    Try
        …
     Catch ex As Exception
          MessageBox.Show(  _
```

```
                "Une erreur est survenue au moment de la mise à jour", _
                "Erreur de mise-à-jour")
        End Try
End Sub
```

Pour tout savoir sur la structure de décision **Try…Catch…End Try** :
http://msdn.microsoft.com/fr-fr/library/ms172857.aspx
Pour tout savoir sur la méthode **Show** de la classe **MessageBox** :
http://msdn.microsoft.com/fr-fr/library/system.windows.messagebox.show(VS.85).aspx

6. Reste à implémenter la confirmation des changements en s'appuyant sur l'option configurée par l'utilisateur dans la boîte Options :

Pour rappel, nous avions sauvegardé l'état de la case à cocher **Toujours demander une confirmation avant d'enregistrer** de la boîte **Options** dans la variable privée **ConfirmBeforeSave** de type **Boolean** de la classe **Main**.

Main.vb [Design] | Main.vb

Main | (Déclarations)

```
1   Imports System.IO
2
3   Public Class Main
4       Public Sub New ...
15  #Region "Déclaration des variables privées de la classe"
16      'Déclaration des variables d'accès aux données
17      Dim ClientsDataSet As AccessClientsDataSet
18      'Déclaration des variables de fichier
19      Dim DataFilePath As String = String.Empty
20      'Déclaration des propriétés de la boîte des options
21      Dim RecentFileListNumber As Short = 0
22      Dim ConfirmBeforeSave As Boolean = Boolean.FalseString
23      Dim SaveDirectoryPath As String = String.Empty
24      Dim TraceEnabled As Boolean = Boolean.FalseString
25      Dim SaveDirectoryType As DirectoryType = DirectoryType.MyDocuments
26      Dim AuthorInfo() As String
27      'Déclaration des énumérations
28      Enum DirectoryType
29          MyDocuments = 0
30          Other = 1
31      End Enum
32      'Déclaration des constantes
33      Const SEPARATOR_SEMICOLON As String = ";"
34  #End Region
```

La sauvegarde de la dite variable est codée dans la méthode **SaveOptions** de la classe **Main** :

```
Private Sub SaveOptions(ByVal form As OptionsDialog)
    'Sauvegarde du type de dossier d'enregistrement
    Select Case form.SaveDirectoryTypeComboBox.SelectedIndex
        Case DirectoryType.MyDocuments
            SaveDirectoryPath = My.Computer.FileSystem.SpecialDirectories
        Case DirectoryType.Other
            SaveDirectoryPath = form.SaveDirectoryPathTextBox.Text
    End Select
    'Sauvegarde de la demande de confirmation avant sauvegarde
    ConfirmBeforeSave = form.ConfirmBeforeSaveCheckBox.Checked
    'Sauvegarde de l'activation du traçage dans le journal de Windows
    TraceEnabled = form.TraceCheckBox.Checked
    If form.TraceCheckBox.Checked Then
        LogOptions()
    End If
    'sauvegarde du nombre d'élements dans la liste des fichiers récents
    RecentFileListNumber = CShort(form.RecentFileListNumericUpDown.Value)
    'sauvegarde des informations de l'auteur
    AuthorInfo = form.AuthorInfoTextBox.Text.Split(SEPARATOR_SEMICOLON)
End Sub
```

L'algorithme de confirmation est le suivant :
Si l'option de confirmation de la boîte **Options** est cochée, il faut donc demander une confirmation à l'utilisateur en utilisant par exemple la méthode **Show** de la classe **MessageBox**.

Si l'utilisateur valide l'enregistrement, le code que nous venons d'écrire pour envoyer les changements dans la base s'applique.
Sinon il faut annuler les changements.

Comment fait-on pour annuler les changements opérés par l'utilisateur dans la grille ?
Cette opération est possible grâce à la richesse de l'objet mémoire **DataSet** dans lequel sont stockées les données. En effet, chaque ligne d'une **DataTable** est un objet **DataRow** qui possède un statut indiquant l'état de la ligne (propriété **RowState**). Chaque ligne mémorise également les valeurs initiales des données avant modification ainsi que les nouvelles valeurs après modification.

Valeurs possibles de la propriété **RowState** d'un objet **DataRow** :

	Nom de membre	Description
	Added	La ligne a été ajoutée à **DataRowCollection** et AcceptChanges n'a pas été appelé.
	Deleted	La ligne a été supprimée à l'aide de la méthode **Delete** de **DataRow**.
	Detached	La ligne a été créée, mais n'appartient à aucun **DataRowCollection**. **DataRow** est dans cet état immédiatement après sa création et avant son ajout à une collection, ou s'il a été supprimé d'une collection.
	Modified	La ligne a été modifiée et **AcceptChanges** n'a pas été appelé.
	Unchanged	La ligne n'a pas été modifiée depuis le dernier appel à **AcceptChanges**.

Valeurs de l'énumération **DataRowState**

Pour en savoir plus sur la propriété **DataRow.RowState** :
http://msdn.microsoft.com/fr-fr/library/system.data.datarow.rowstate(VS.80).aspx

Du coup, pour valider ou annuler tous les changements d'un groupe de données, le Framework.NET nous fournit deux méthodes, respectivement **AcceptChanges** et **RejectChanges**. Ces méthodes peuvent s'appliquer sur le **DataSet** entier (s'il contient plusieurs tables) ou sur une table

DataTable particulière, ou même sur une ligne **DataRow**.

Pour en savoir plus sur la méthode **RejectChanges** :
http://msdn.microsoft.com/fr-fr/library/system.data.datatable.rejectchanges(VS.80).aspx
Idem sur la méthode **AcceptChanges** :
http://msdn.microsoft.com/fr-fr/library/system.data.dataset.acceptchanges.aspx

Notez que la méthode **AcceptChanges** est appelée automatiquement à la fin du processus de traitement de la méthode **Update** du **TableAdapter** pour marquer toutes les lignes à l'état **Unchanged**.

- Editez la procédure **EnregistrerToolStripMenuItem_Click**.
- Ajoutez une structure de décision pour valider l'état de la variable **ConfirmBeforeSave** attestant de la volonté de l'utilisateur de bénéficier d'une confirmation avant enregistrement :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
    Dim ClientsTableAdapter = _
            New AccessClientsDataSetTableAdapters.ContactsTableAdapter()
    Try
       'Mise-à-jour des données dans la base de données
       'Appliquer les changements des contrôles …
       MainBindingSource.EndEdit()
       'Vérifiez que des modifications ont eu lieu
       If ClientsDataSet.HasChanges Then
          'Vérifiez si l'option de confirmation est cochée dans la boîte Options
          If ConfirmBeforeSave Then
              'Appliquer les changements dans la base de données
              ClientsTableAdapter.Update(ClientsDataSet.Contacts)
          End If
       End If
     Catch ex As Exception
          MessageBox.Show(  _
              "Une erreur est survenue au moment de la mise à jour",  _
              "Erreur de mise-à-jour")
    End Try
End Sub
```

- Si le résultat du test est vrai, demandez confirmation à l'utilisateur avec la méthode **Show** de la classe **MessageBox** avant d'enregistrer les changements :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
```

```
…
            'Vérifiez si l'option de confirmation est cochée dans la boîte Options
            If ConfirmBeforeSave Then
                If MessageBox.Show("Etes vous sûr de vouloir enregistrer
                                            les changements ?", _
                                        "Confirmation de l'enregistrement", _
                                        MessageBoxButtons.OKCancel,  _
                                        MessageBoxIcon.Question) _
                        = Windows.Forms.DialogResult.OK Then
                    'Appliquer les changements dans la base de données
                    ClientsTableAdapter.Update(ClientsDataSet.Contacts)
                End If
            End If
…
End Sub
```

- Sinon, annulez les changements avec la méthode **RejectChanges** de l'objet **ClientsDataSet** :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
                    Handles EnregistrerToolStripMenuItem.Click
…
            'Vérifiez si l'option de confirmation est cochée dans la boîte Options
            If ConfirmBeforeSave Then
                If MessageBox.Show("Etes vous sûr de vouloir enregistrer
                                            les changements ?", _
                                        "Confirmation de l'enregistrement", _
                                        MessageBoxButtons.OKCancel,  _
                                        MessageBoxIcon.Question) _
                        = Windows.Forms.DialogResult.OK Then
                    'Appliquer les changements dans la base de données
                    ClientsTableAdapter.Update(ClientsDataSet.Contacts)
                Else
                    ClientsDataSet.RejectChanges()
                End If
            End If
…
End Sub
```

Attention ! Il ne faut pas oublier de réaliser l'enregistrement même si l'option de confirmation n'a pas été cochée par l'utilisateur.

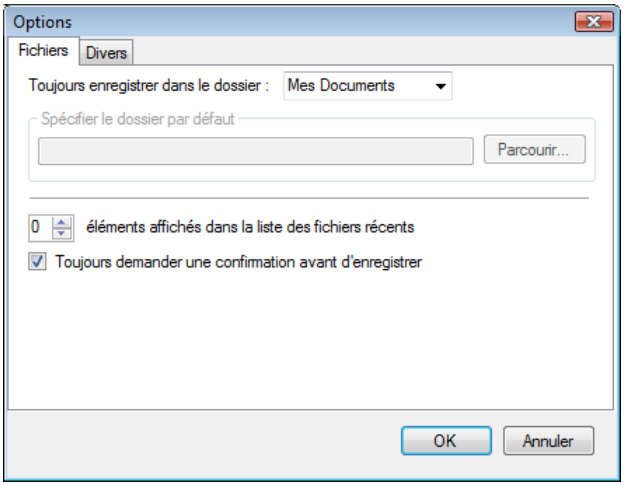
- Rajoutez une instruction **Else** sur le test de la valeur de **ConfirmBeforeSave** pour enregistrer les modifications dans la source de données sans aucune confirmation pour l'utilisateur :

Code VB

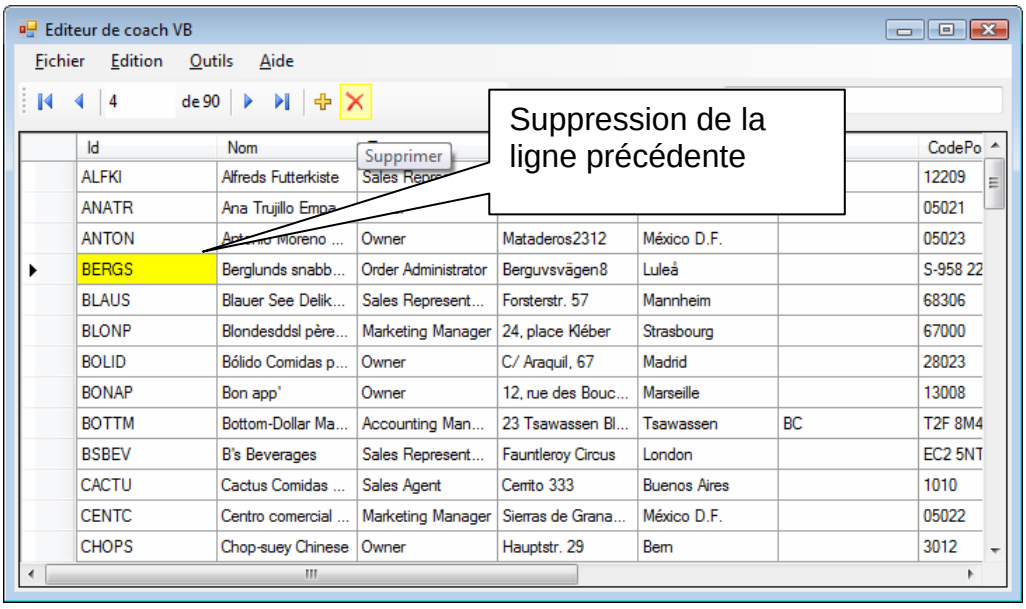
```
Private Sub EnregistrerToolStripMenuItem_Click( …)  _
```

```
                    Handles EnregistrerToolStripMenuItem.Click
…
        'Vérifiez si l'option de confirmation est cochée dans la boîte Options
        If ConfirmBeforeSave Then
            If MessageBox.Show("Etes vous sûr de vouloir enregistrer
                                    les changements ?", _
                                "Confirmation de l'enregistrement", _
                                MessageBoxButtons.OKCancel,  _
                                MessageBoxIcon.Question) _
                = Windows.Forms.DialogResult.OK Then
                'Appliquer les changements dans la base de données
                ClientsTableAdapter.Update(ClientsDataSet.Contacts)
            Else
                ClientsDataSet.RejectChanges()
            End If
        Else
            'Appliquer les changements sans confirmation
            ClientsTableAdapter.Update(ClientsDataSet.Contacts)
        End If
…
End Sub
```

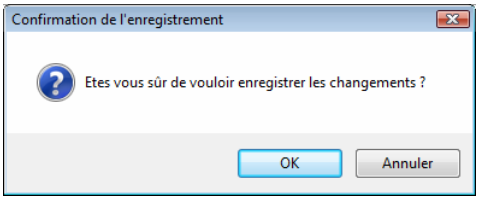
7. Testez la mise en place de la confirmation :
   - Exécutez l'application en cliquant sur ▶ (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir** pour charger les données de la base Access dans la grille d'affichage.
   - Cliquez le menu **Outils** > **Options** et cochez la case **Toujours demander une confirmation avant d'enregistrer** :



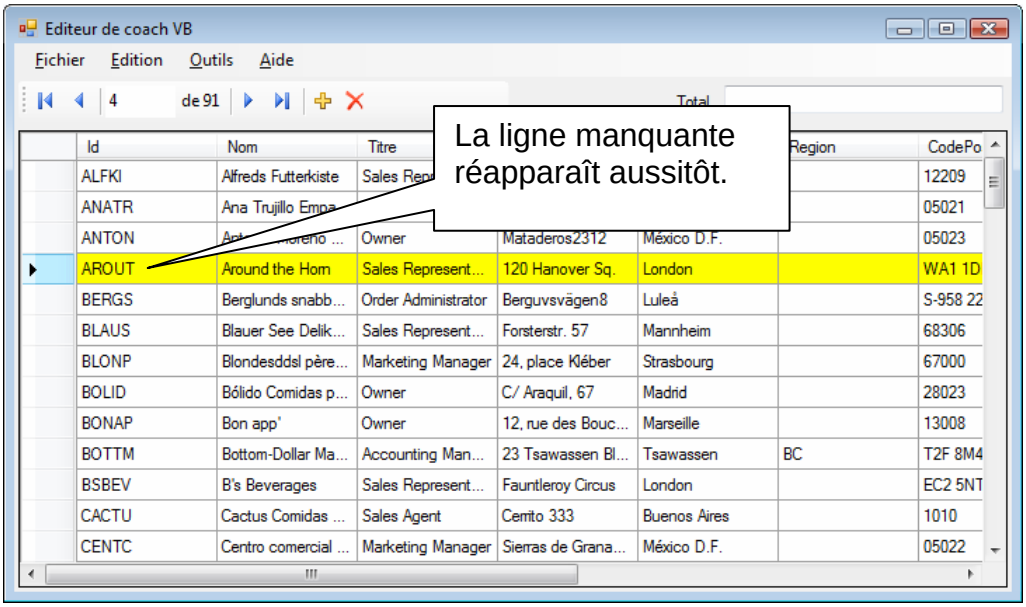
   - Validez par **OK**.
   - Supprimez par exemple une ligne de données au hasard dans la grille en utilisant l'icône ✕.



- Cliquez le menu **Fichier** > **Enregistrer**.
- Répondez par **Annuler** à la question qui s'affiche :



- La grille revient aussitôt à son état initial reflétant la réinitialisation du DataSet. Merci le databinding ☺…



Pour tout savoir sur **Comment mettre à jour des modifications apportées à un groupe de données dans une base de données** : http://msdn.microsoft.com/fr-fr/library/xzb1zw3x(VS.80).aspx

3 Travailler avec des données de SQL Server 2005 Express Edition

Et si vous envisagiez de mettre les mêmes données que dans l'exercice précédent mais cette fois dans le moteur de SQL Server ? L'objectif de cet exercice est de vous montrer que ce n'est pas plus ou moins compliqué qu'avec Access.
Même si SQL Server 2005 Express Edition est gratuit, vous bénéficiez bien du même moteur que les autres éditions de SQL Server. Vous pourriez par exemple créer et exploiter des procédures stockées.

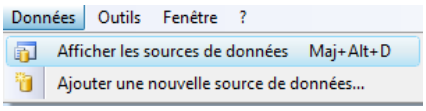
3.1 Définir la source de données

La procédure que nous avons suivie à l'exercice précédent s'applique aussi bien à une base de données Access qu'à une base de données SQL Server. La première étape consiste donc à définir une **source de données** pour générer un groupe de données typé reprenant la structure des objets qui nous intéressent dans notre base SQL Server Express.

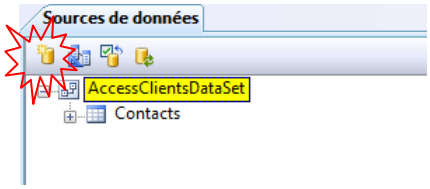
Contexte fonctionnel
La base de données qui vous est fournie à titre d'exemple a une structure similaire à la base Access de l'exercice précédent. Elle est constituée d'une seule table nommée **Contacts**.
Vous trouverez le fichier de données **Clients.mdf** dans le répertoire **…Atelier 5\Fichiers utiles**.

Déroulement de l'exercice :

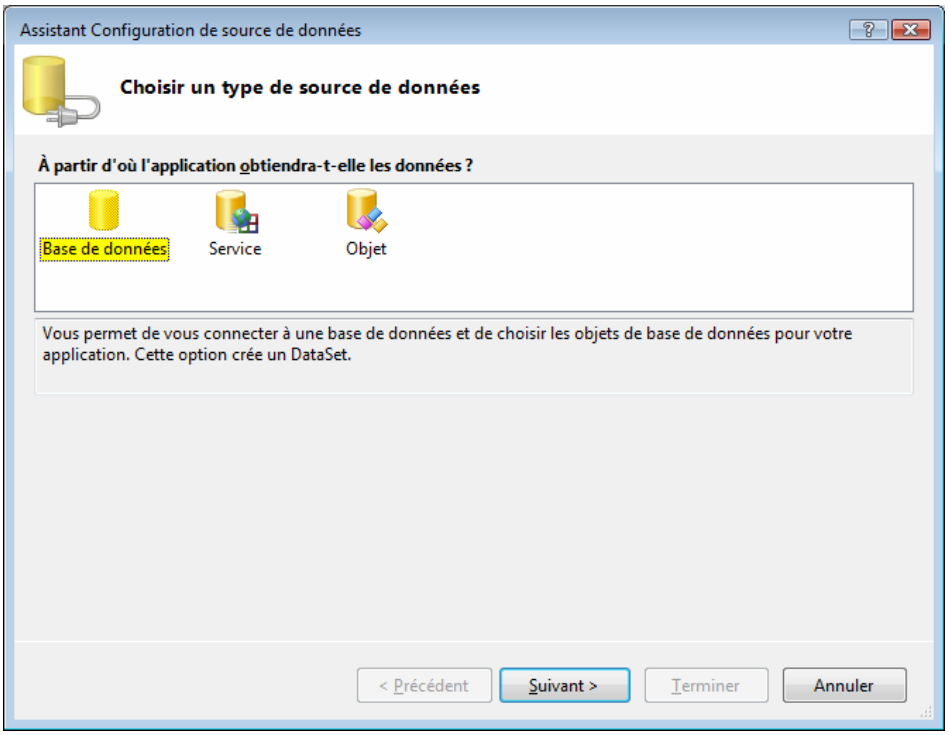
1. Créer une nouvelle source de données pointant sur le fichier Clients.mdf :
   - Toujours sur la base du projet précédent ouvert dans Visual Studio, sélectionnez le menu **Données** > **Afficher les sources de données** pour afficher la fenêtre **Sources de données** :



   - Dans la fenêtre **Sources de données**, cliquez l'icône pour ajouter une nouvelle source de données au projet (en plus de celle que nous avons défini pour Access) :



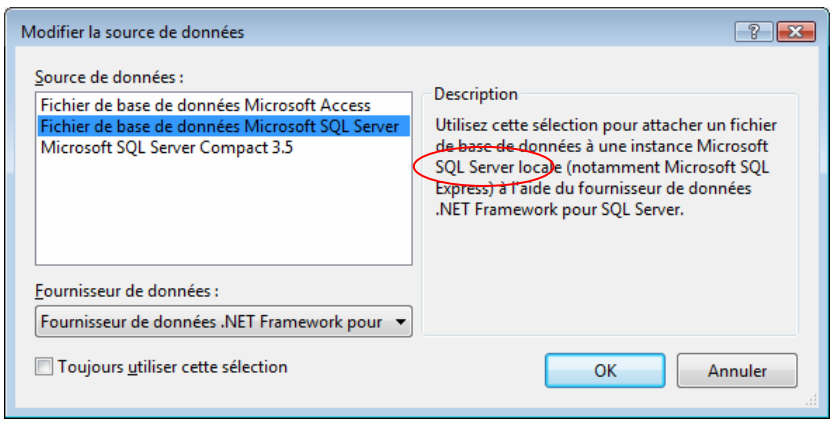
   - Dans l'écran **Choisir un type de source de données**, sélectionnez **Base de données**.



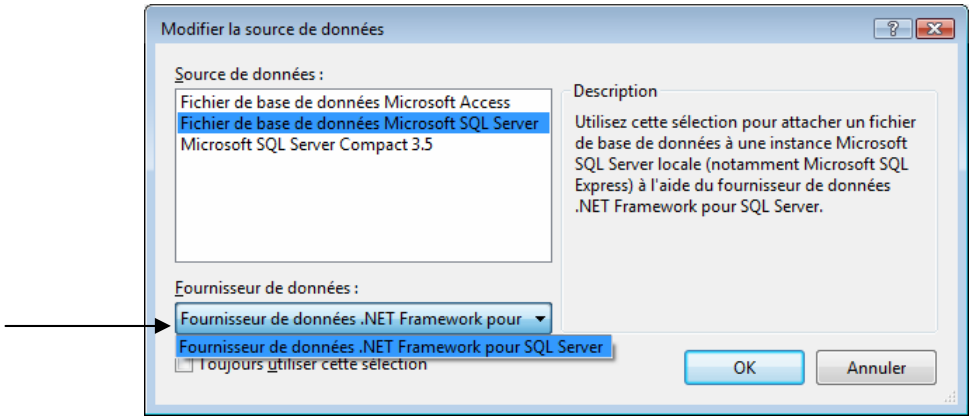
- Cliquez **Suivant**.
- Dans l'écran **Choisir votre connexion de données**, cliquez **Nouvelle connexion**.

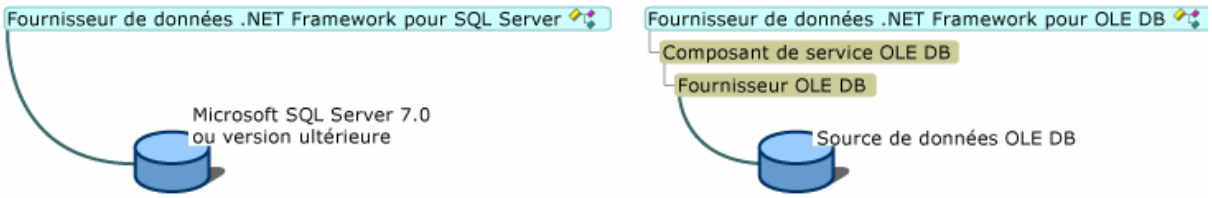
Vous devez voir dans la liste la connexion que vous avez construite à l'exercice précédent pointant sur la base Access.

- Dans la boîte **Ajouter une connexion**, cliquez sur **Modifiez** pour sélectionner le fournisseur de données pour accéder cette fois-ci à une base SQL Server.
- Sélectionnez **Fichier de base de données Microsoft SQL Server** dans la liste des sources de données proposée :

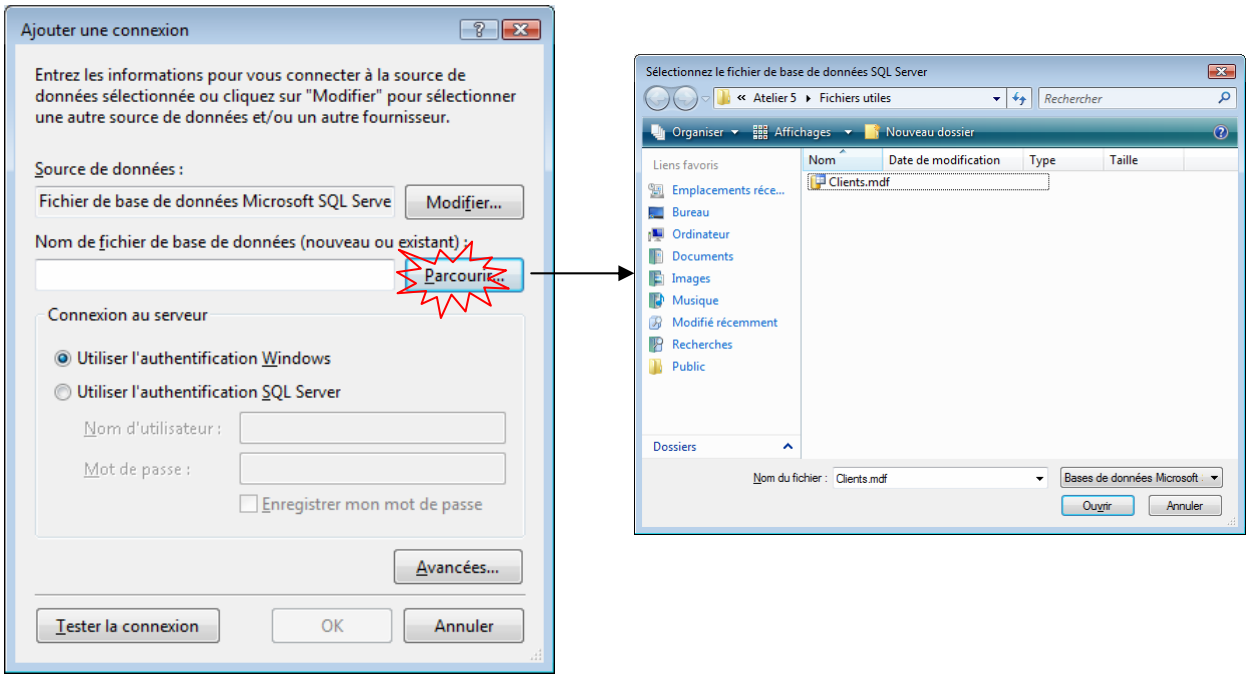


Il faut savoir que le fournisseur de données pour SQL Server fourni dans le Framework.NET est très léger et donc très performant. Il est en effet optimisé de façon à accéder directement à SQL Server, sans ajout d'une couche OLEDB ou ODBC comme c'est le cas pour le fournisseur que nous avons utilisé à l'exercice précédent pour accéder à la base Access :



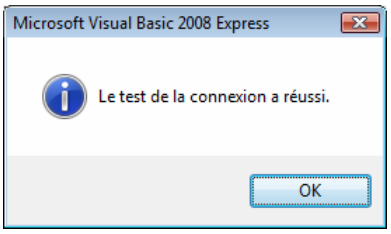


- Validez la sélection du fournisseur en cliquant **OK**.
- De retour dans l'écran **Ajouter une connexion**, cliquez **Parcourir** à droite de la zone de nom de fichier de base de données.
- Sélectionnez le fichier de base de données Microsoft SQL Server 2005 **Clients.mdf** fourni en exemple dans le dossier **..\Atelier 5\Fichiers utiles** :



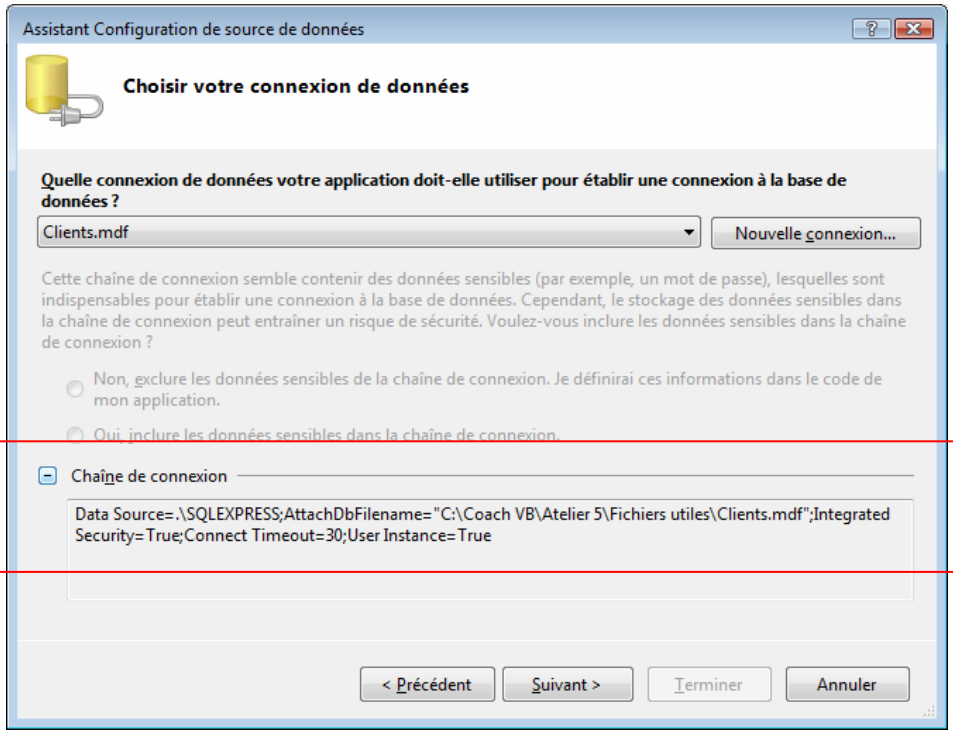
- Cliquez **Ouvrir**.
- Gardez l'option de sécurité par défaut qui utilise l'authentification Windows.

Avant de valider la connexion établie, utilisez le bouton **Tester la connexion** de la boîte **Ajouter une connexion** pour vérifier que Visual Studio est à même d'établir la connexion sur la base de données avec les éléments que vous lui avez fournis :



Un message vous informe si la connexion a pu être établie ou non.

- Une fois la connexion vérifiée, cliquez sur **OK** pour quitter la boîte **Ajouter une connexion**.
- De retour dans l'écran **Choisir votre connexion de données**, cliquez le sigle ⊞ devant **Chaîne de connexion** pour afficher le détail de la connexion que vous venez de configurer :



Que contient cette chaîne de connexion ?
Elle est composée de plus de couples **Clé/Valeur** que celle pour Access. Chaque fournisseur a en effet ses spécificités.

Ici, on retrouve :

- **Data Source** : cette clé détermine le nom de l'instance SQL Server dans laquelle est prise en charge la base de données. Une instance de SQL Server Express Edition est une instance nommée donc elle suit le format :
  <Nom de la machine>\<Nom de l'instance>
  Le nom de la machine peut être substitué par un point et le nom de l'instance est **SQLEXPRESS**.
- **AttachDbFilename** : cette clé indique le nom du fichier de base de données d'extension ***.mdf** précédé du chemin d'accès complet. Grâce à cette clé, vous n'avez pas à vous soucier d'attacher le fichier de données au moteur SQL Server via la console de gestion de SQL Server. Le Framework .NET va attacher lui-même la base au serveur de manière automatique à chaque exécution de l'application. Pratique non ?
- **Integrated Security** : cette clé indique que vous vous appuyez sur l'authentification Windows. Mais alors comment est-ce qu'un compte utilisateur local va être à même d'attacher dynamiquement la base de données au moteur SQL Server Express ?
  En fait, c'est grâce à une nouvelle fonctionnalité de SQL Server Express qu'on appelle *instances utilisateur*. Il s'agit d'une instance séparée du moteur de base de données qui est générée automatiquement et qui permet donc à l'utilisateur de se connecter sans avoir de privilèges administrateurs système.
- **User Instance** : cette clé indique que vous allez justement utiliser

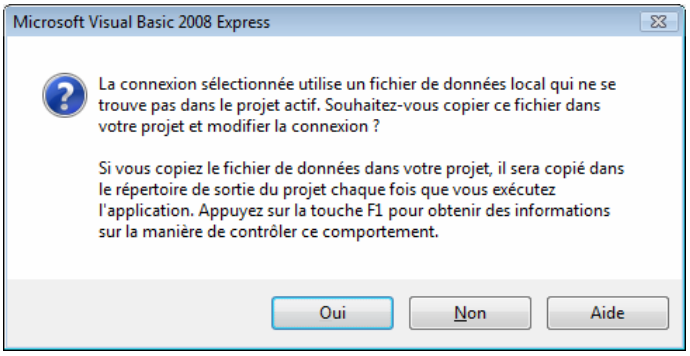
une instance utilisateur de SQL Server Express. Elle va être générée par l'instance parent que vous avez indiqué dans la clé **Data Source**.

- **Connect Timeout** : indique le temps maximum (en secondes) pour établir la connexion. Au-delà, la connexion échoue.

Pour tout savoir sur la syntaxe à utiliser dans les chaînes de connexion ADO.NET : http://msdn.microsoft.com/fr-fr/library/ms254500.aspx
Pour en savoir plus sur les instances utilisateur de SQL Server Express : http://msdn.microsoft.com/fr-fr/library/ms254504.aspx

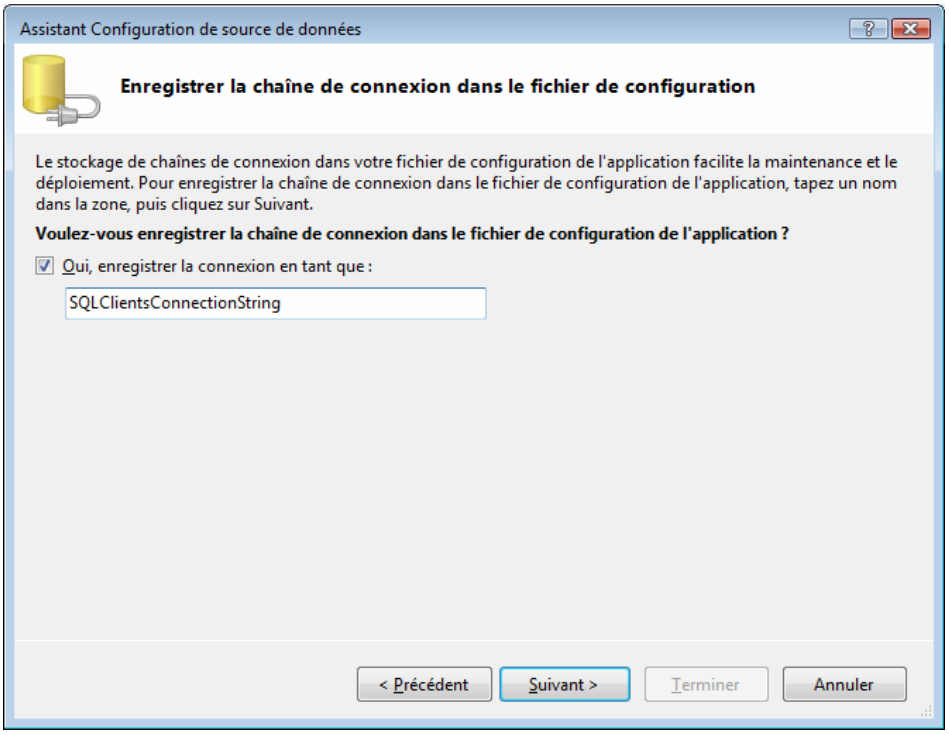
- Validez en cliquant **Suivant**.

Visual Studio détecte que vous utilisez un fichier de données et vous suggère de l'inclure *en tant que fichier* dans le projet.



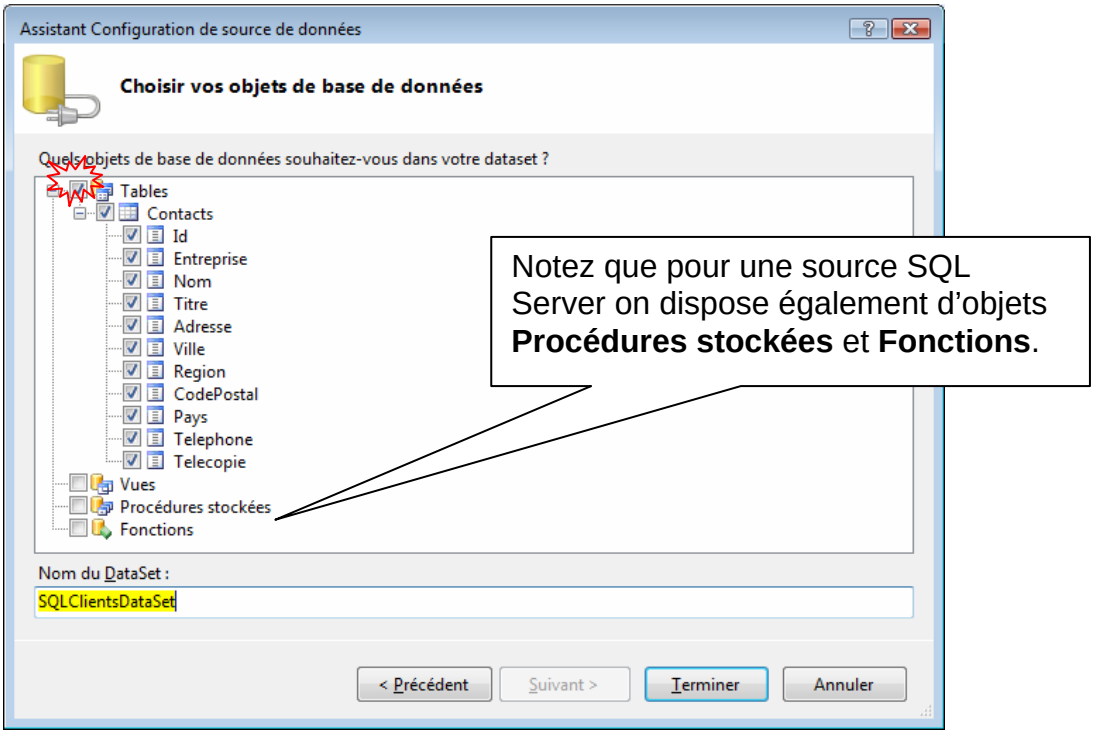
N'oubliez pas qu'en répondant **Oui** à la question, la chaîne de connexion va donc être mise à jour pour pointer non plus sur le chemin initial du fichier de données mais sur le répertoire de l'application défini via la chaîne de substitution **|DataDirectory|**.

- Répondez **Oui** à la question.
- Dans l'écran **Enregistrer la chaîne de connexion dans le fichier de configuration**, modifiez le nom de la chaîne en : **SQLClientsConnectionString**.

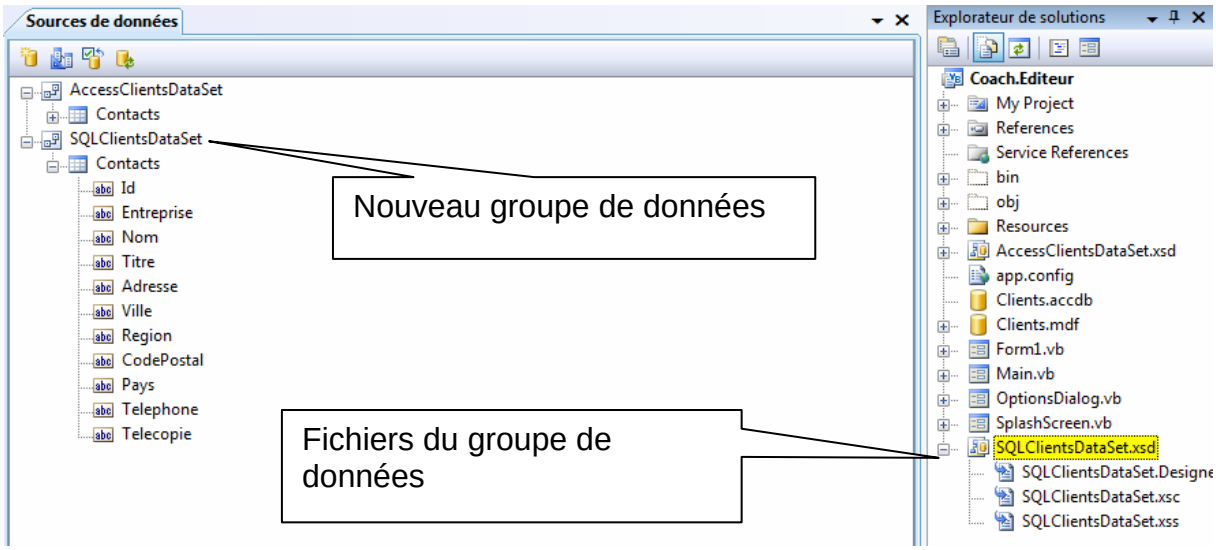


Une nouvelle balise **<add>** dans la section **<connectionStrings>** va donc être ajoutée au fichier de configuration du projet avec la définition de la chaîne de connexion vers le fichier de données SQL Server.

- Cliquez sur **Suivant** pour accepter l'enregistrement de la connexion.
- Dans l'étape **Choisir vos objets de base de données**, étendez les différents objets en cliquant sur ⊞ pour voir les objets disponibles dans la source SQL Server.
- Cochez la table **Contacts**. Toutes les colonnes sont alors automatiquement sélectionnées.
- Modifiez le nom proposé par défaut pour l'objet DataSet par exemple en **SQLClientsDataSet**.



- Cliquez **Terminer** pour terminer la configuration de la source de données.

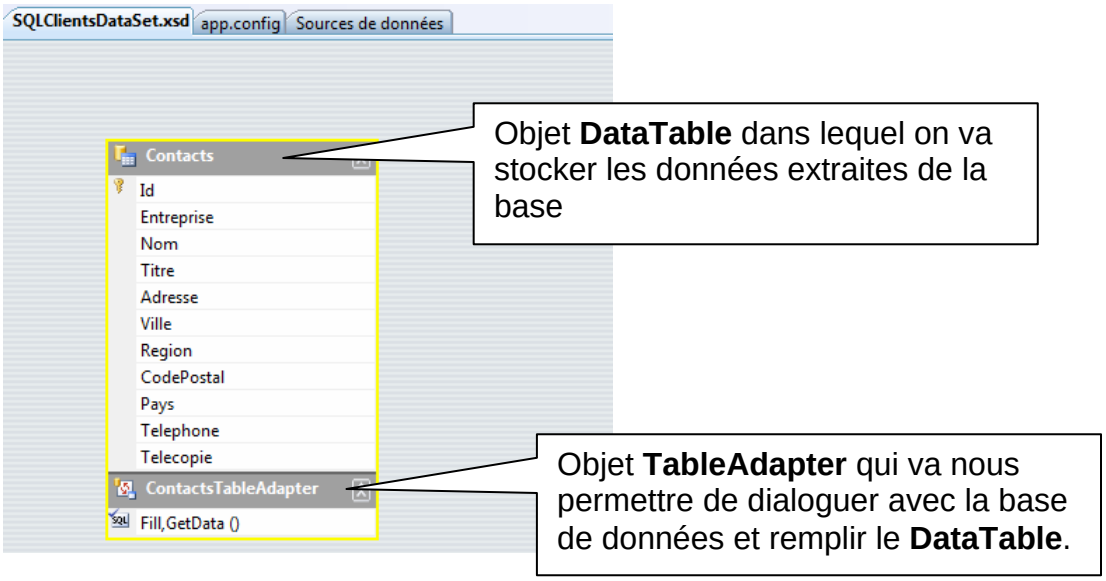


Vous pouvez ouvrir le fichier **app.config** pour vérifier la chaîne de connexion qui a été ajoutée à la section **connectionStrings** :

app.config | Sources de données

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
    </configSections>
    <connectionStrings>
        <add name="Coach.Editeur.My.MySettings.AccessClientsConnectionString"
            connectionString="Provider=Microsoft.ACE.OLEDB.12.0;Data Source=|DataDirectory|\Clients.a
            providerName="System.Data.OleDb" />
        <add name="Coach.Editeur.My.MySettings.SQLClientsConnectionString"
            connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Clients.mdf;I
            providerName="System.Data.SqlClient" />
    </connectionStrings>
    <system.diagnostics>
```

Double cliquez sur le fichier **SQLClientsDataSet.xsd** pour afficher le **Concepteur de DataSet** et observer les objets qui sont à votre disposition pour accéder aux données :



3.2 Manipuler les données

Reste à coder l'extraction et la mise à jour des données comme nous l'avons fait pour la base Access.

1. Modifiez la définition du DataSet dans la déclaration des variables privées de la classe Main pour qu'elle pointe désormais sur le groupe de données SQLClientsDataSet :
    - Modifiez la déclaration comme suit :

Code VB

```
Public Class Main
…
Dim ClientsDataSet As ~~AccessClientsDataSet~~ SQLClientsDataSet
…
```

    - Modifiez également le nom du constructeur de l'objet dans la procédure **New** :

Code VB

```
Public Class Main
   Public Sub New()
      …
      'Instanciation du DataSet qui va servir à travailler sur les données
      ClientsDataSet = New ~~AccessClientsDataSet()~~ SQLClientsDataSet()
   End Sub
…
```

2. Modifiez les gestionnaires d'évènement des options de menu Ouvrir et Enregistrer pour qu'ils utilisent le nouveau groupe de données :
   - Repérez la procédure **OuvrirToolStripMenuItem_Click**.
   - Modifiez le nom du constructeur dans la déclaration de l'objet TableAdapter :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
      'Lecture des données dans la base de données
      Dim ClientsTableAdapter = _
            ~~New AccessClientsDataSetTableAdapters.ContactsTableAdapter()~~
            New SQLClientsDataSetTableAdapters.ContactsTableAdapter()
      'Extraction des données et chargement du DataSet
      ClientsTableAdapter.Fill(ClientsDataSet.Contacts)
      'Configuration du gestionnaire de liaison sur la source de données
      MainBindingSource.DataSource = ClientsDataSet.Contacts
      'Liaison du gestionnaire avec les contrôles d'affichage de données
      MainDataGridView.DataSource = MainBindingSource
      MainBindingNavigator.BindingSource = MainBindingSource
   End Sub
```

   - Idem pour la procédure **EnregistrerToolStripMenuItem_Click** :

Code VB

```
Private Sub EnregistrerToolStripMenuItem_Click(…) _
                  Handles EnregistrerToolStripMenuItem.Click
      Dim ClientsTableAdapter = _
            ~~New AccessClientsDataSetTableAdapters.ContactsTableAdapter()~~
            New SQLClientsDataSetTableAdapters.ContactsTableAdapter()
      Try
         …
      Catch ex As Exception
         …
      End Try
End Sub
```

3. Testez le fonctionnement de l'application sur la base de données SQL Server :

- Exécutez l'application en cliquant sur (ou touche F5).
- Cliquez le menu **Fichier** > **Ouvrir** pour charger les données.
- Effectuez quelques modifications.
- Cliquez le menu **Fichier** > **Enregistrer** pour sauvegarder les changements.
- Cliquez menu le **Fichier** > **Ouvrir** pour vérifier que vos modifications ont été prises en compte.

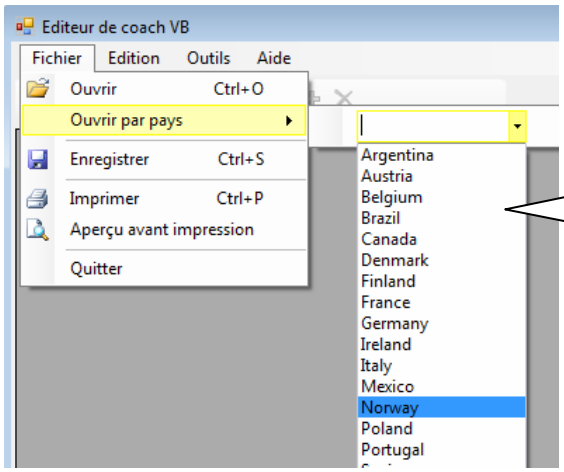
Bravo ! Vous avez écrit une application capable de manipuler des données provenant d'une base SQL Server.

4 Pour aller plus loin…

4.1 Définir de nouvelles requêtes utilisant des procédures stockées

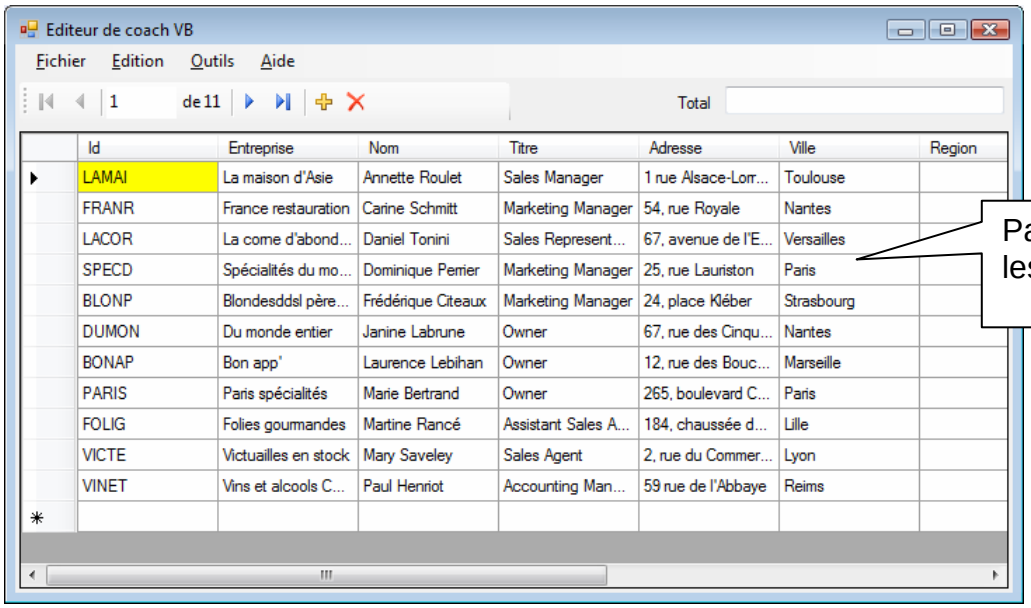
Contexte fonctionnel

Imaginez que vous souhaitiez donner la possibilité de rapatrier les clients en fonction d'un pays donné. Par exemple avec une option de menu du type :



Liste déroulante qui permet de sélectionner un pays déterminé dont on veut extraire la liste de contacts.

L'objectif est de charger dans la liste uniquement les clients du pays voulu :



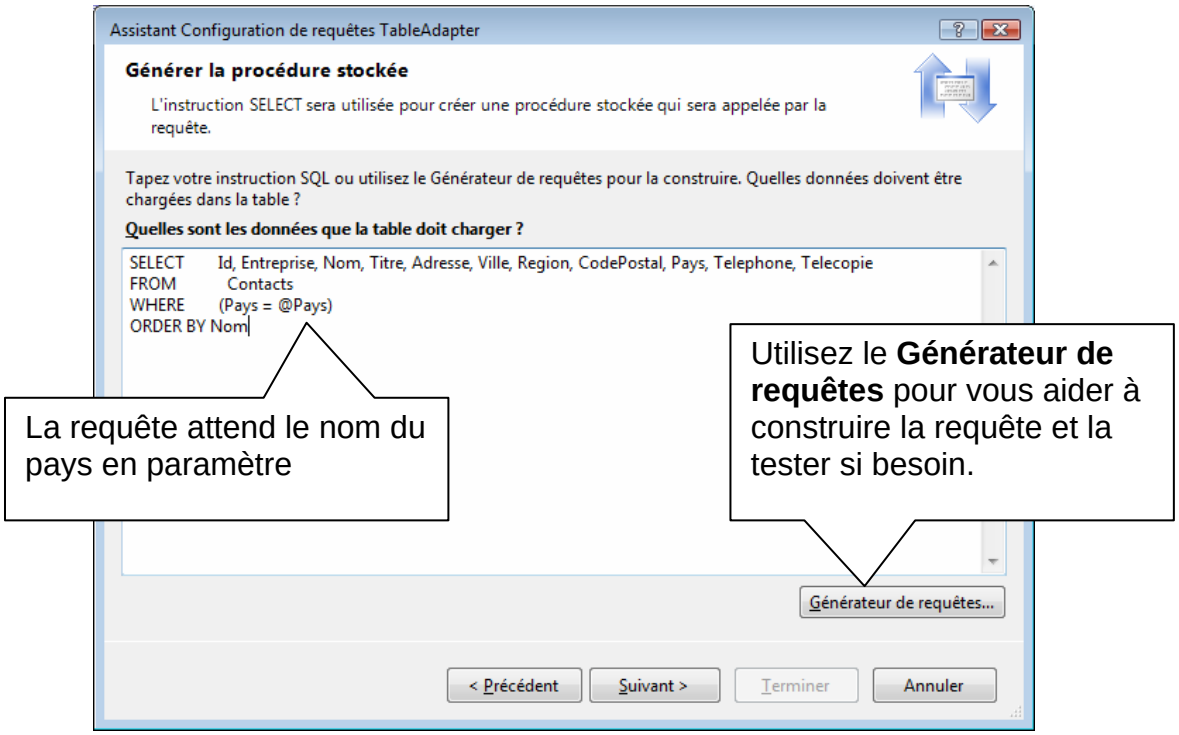
Par exemple tous les clients français.

Déroulement de l'exercice :

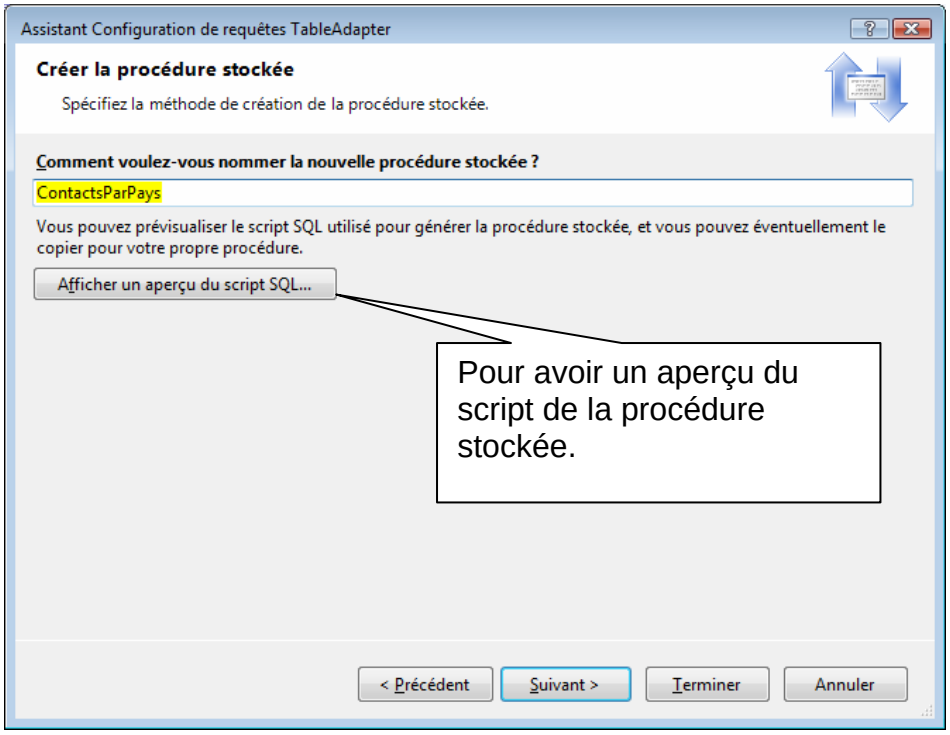
L'objectif est de vous montrer qu'avec les nombreux outils de Visual Studio que sont le Concepteur de DataSet, le Concepteur de DataAdapter, l'Explorateur de bases de données, l'Assistant

Configuration Sources de données, il existe de nombreuses possibilités d'extension de ce que nous avons mis en place dans les deux exercices précédents.
Par exemple, nous allons rajouter une requête à l'objet TableAdapter à partir du Concepteur de DataSet pour construire la requête qui rapatrie une liste de contacts en fonction d'un pays passé en paramètre.
Au passage, l'exercice montre comment créer une procédure stockée sur la base de la requête souhaitée dans la base SQL Server…

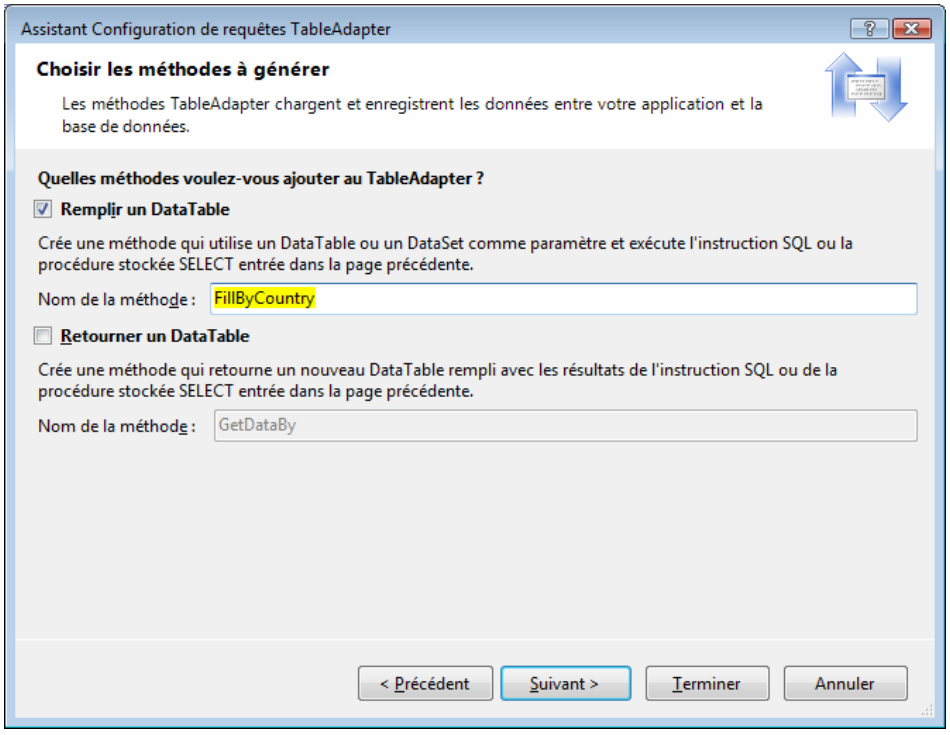
1. Créez une nouvelle requête sur l'objet ContactsTableAdapter :
   - Affichez le **Concepteur de DataSet** en double cliquant sur le fichier **SQLClientsDataSet.xsd** dans l'**Explorateur de solutions**.
   - Sélectionnez l'objet **ContactsTableAdapter**.
   - Faites un clic droit > **Ajouter une requête…**
   - Dans l'écran **Choisissez un type de commande**, sélectionnez **Créer une nouvelle procédure stockée** puis cliquez **Suivant**.
   - Dans l'écran **Choisir un type de requête**, sélectionnez **SELECT qui retourne des lignes** puis cliquez **Suivant**.
   - Dans l'écran **Générer la procédure stockée**, entrez la requête comme suit :



   - Cliquez sur **Suivant**.
   - Dans l'écran **Créer la procédure stockée**, entrez le nom de la requête par exemple : **ContactsParPays**.

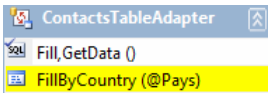


- Cliquez sur **Suivant**.
- Dans l'écran **Choisir les méthodes à générer**, **décochez Retourner un DataTable**, et entrez le nom de la première méthode par exemple : **FillByCountry**.



- Cliquez sur **Suivant**.
- Cliquez **Terminer** pour valider l'écran **Résultats de l'Assistant**.

- L'objet **ContactsTableAdapter** possède une requête complémentaire dans le **Concepteur de DataSet** :



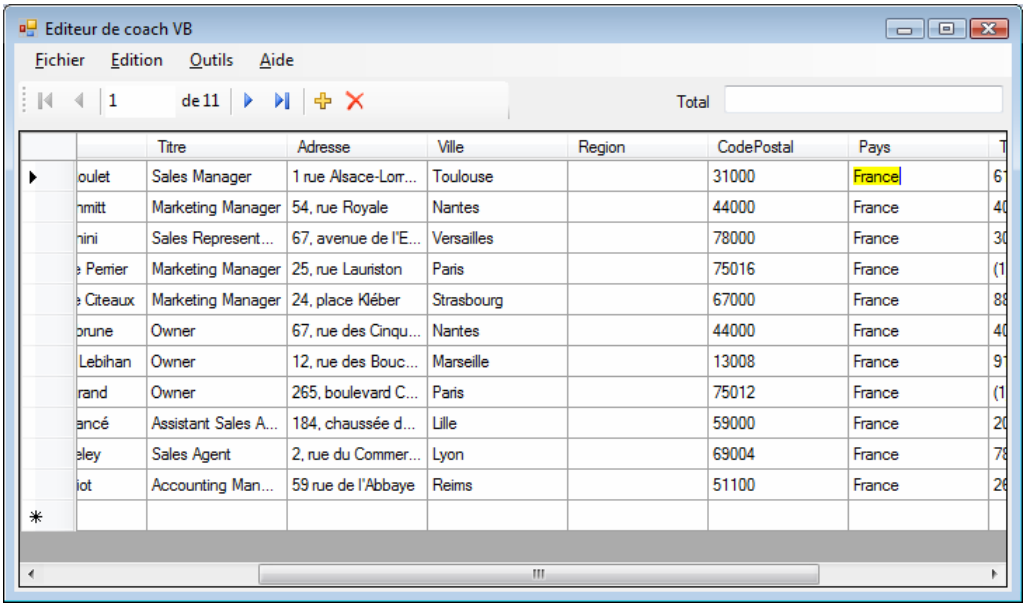
2. Pour utiliser ce type de requête paramétrée, suivez le guide :
   - Affichez le code de la classe **Main**.
   - Retrouvez le gestionnaire d'évènement **OuvrirToolStripMenuItem_Click**.
   - Modifiez la ligne d'extraction des données par l'ordre suivant :

Code VB

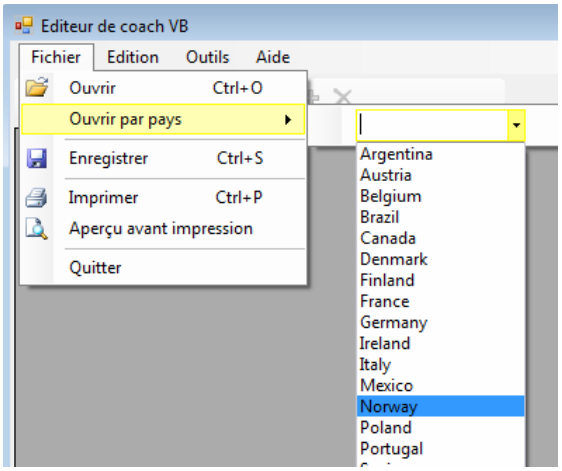
```
Private Sub OuvrirToolStripMenuItem_Click(ByVal sender As System.Object, _
                                          ByVal e As System.EventArgs) _
                                          Handles OuvrirToolStripMenuItem.Click
    'Lecture des données dans la base de données
    Dim ClientsTableAdapter = _
        New SQLClientsDataSetTableAdapters.ContactsTableAdapter()
    'Extraction des données et chargement du DataSet
    ClientsTableAdapter.Fill(ClientsDataSet.Contacts)
    ClientsTableAdapter.FillByCountry(ClientsDataSet.Contacts,"France")
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = ClientsDataSe
    'Liaison du gestionnaire avec les contrôles d'affic
    MainDataGridView.DataSource = MainBindingSo
    MainBindingNavigator.BindingSource = MainBin
End Sub
```

Paramètre de la requête que l'on retrouve en tant que paramètre de la méthode.

3. Testez le chargement des clients français :
   - Exécutez l'application en cliquant sur ▶ (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir** pour charger les données.



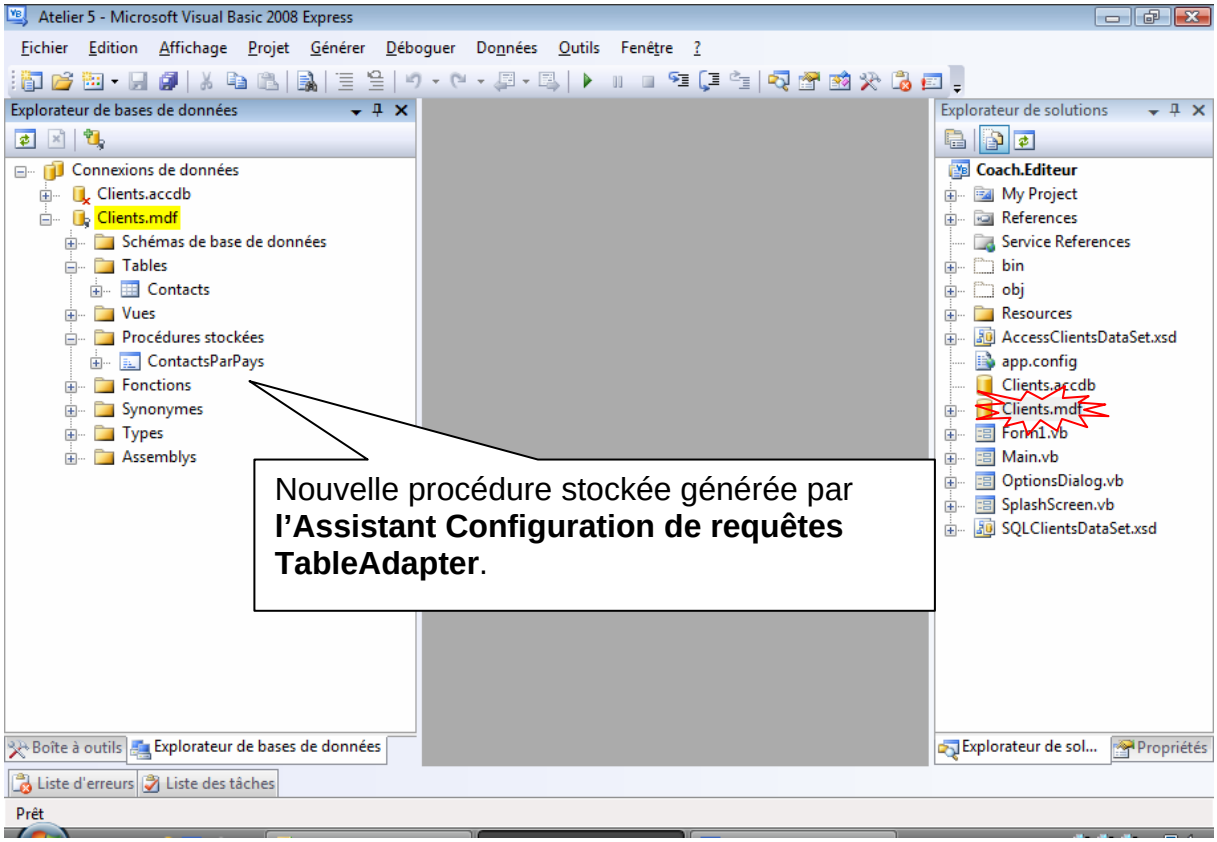
Evidemment, le pays codé en dur tel que nous venons de le faire n'est pas franchement idéal…
Pour créer une option de menu dynamique telle que :



reportez-vous à la solution de l'exercice dans le répertoire …**Atelier 5\Solution\3 – Pour aller plus loin**. Mais ce n'est qu'une solution parmi d'autres.

Et au fait, comment vérifiez que la base de données contient bien la procédure stockée que nous avons créée ?

- Double cliquez le fichier **Clients.mdf** dans l'Explorateur de solutions pour faire apparaître l'**Explorateur de bases de données**.
- Etendez le nœud **Procédures stockées** et vérifiez que la procédure **ContactsParPays** existe.



La fenêtre **Explorateur de bases de données** comprend de nombreux outils pour vous aider à administrer la base de données sans passer par l'outil de gestion de SQL Server (SQL Server Management Studio).

4.2 Ajouter une requête paramétrée à un formulaire dans une application Windows

Une autre alternative pour utiliser des requêtes paramétrées comme à l'exercice 4.1 précédent est d'utiliser l'assistant qui démarre sur l'option de menu **Données** > **Ajouter une requête** à partir du formulaire en mode **Design**. Cet assistant :

- Crée une requête supplémentaire (paramétrée) sur votre TableAdapter,
- Crée des contrôles d'affichage invitant l'utilisateur à fournir la valeur des paramètres de la requête,
- Crée un bouton pour exécuter la requête.

Pour consulter la marche à suivre :
http://msdn.microsoft.com/fr-fr/library/aeez5k4f(VS.80).aspx

4.3 Développer avec ADO.NET

Pour optimiser le code que nous venons de faire dans cet atelier, il faut bien sûr mettre les mains dans le cambouis c'est-à-dire apprendre à programmer ADO.NET.

Par exemple, pour afficher des données, il n'est pas utile de charger un DataSet. Il suffit d'utiliser un curseur matérialisé par l'objet DataReader. Cet objet fonctionne alors en mode *connecté*.

Voici deux conférences incontournables présentées par Mitsu Furuta de Microsoft France :

Les rencontres accès aux données 2005 :
http://www.microsoft.com/france/vision/WebcastMsdn.aspx?EID=c473ba88-8f22-4f68-afa4-e7b1640def8e

Les mêmes rencontres 3 ans plus tard avec toutes les nouveautés d'ADO.NET à savoir la technologie **LINQ** :
http://msdn.microsoft.com/fr-fr/data/bb743261.aspx

Développer ses objets

Sommaire

1 INTRODUCTION 3

1.1 CONTEXTE FONCTIONNEL 3

2 CREER ET UTILISER DES OBJETS PERSONNALISES 5

2.1 CREER UNE BIBLIOTHEQUE DE CLASSES 6
2.2 CREER UNE CLASSE 8
2.3 CREER UNE COLLECTION D'OBJETS 16
2.4 UTILISER LES CLASSES 20

3 UTILISER DES ATTRIBUTS 41

4 POUR ALLER PLUS LOIN… 66

4.1 LES TYPES GENERIQUES 66
4.2 LIBERER LA MEMOIRE 66
4.3 UTILISER DES INTERFACES 67

1 Introduction

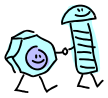
1.1 Contexte fonctionnel

Rappel du contexte fonctionnel du tutorial du coach VB

L'objectif du tutorial du Coach VB est d'accompagner les développeurs à la découverte et la prise en main du langage Visual Basic (VB) pour la construction d'applications avec une approche orientée objet.

Pour rappel, vous pouvez repérer facilement deux caractéristiques importantes du langage à l'aide des logos suivants en marge :

Ce logo marque une fonctionnalité de VB ou de Visual Studio qui permet de développer vite (et juste ☺).

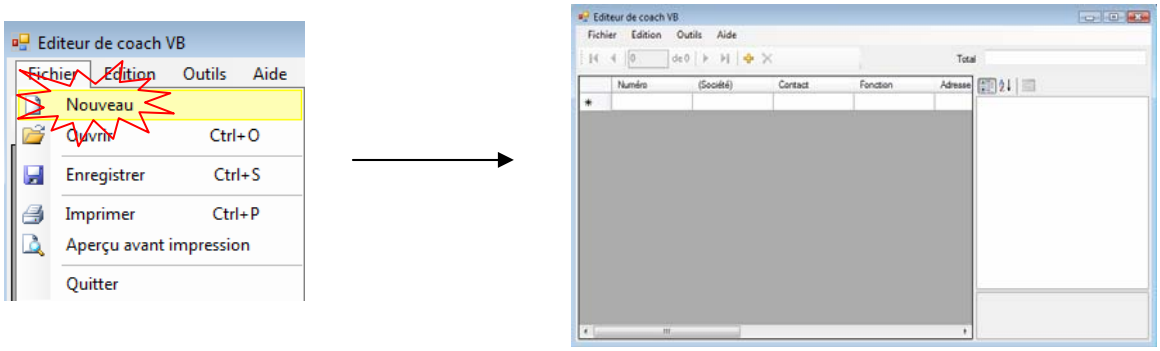


Ce logo met en évidence une caractéristique de la programmation orientée objet.

Contexte fonctionnel du sixième atelier

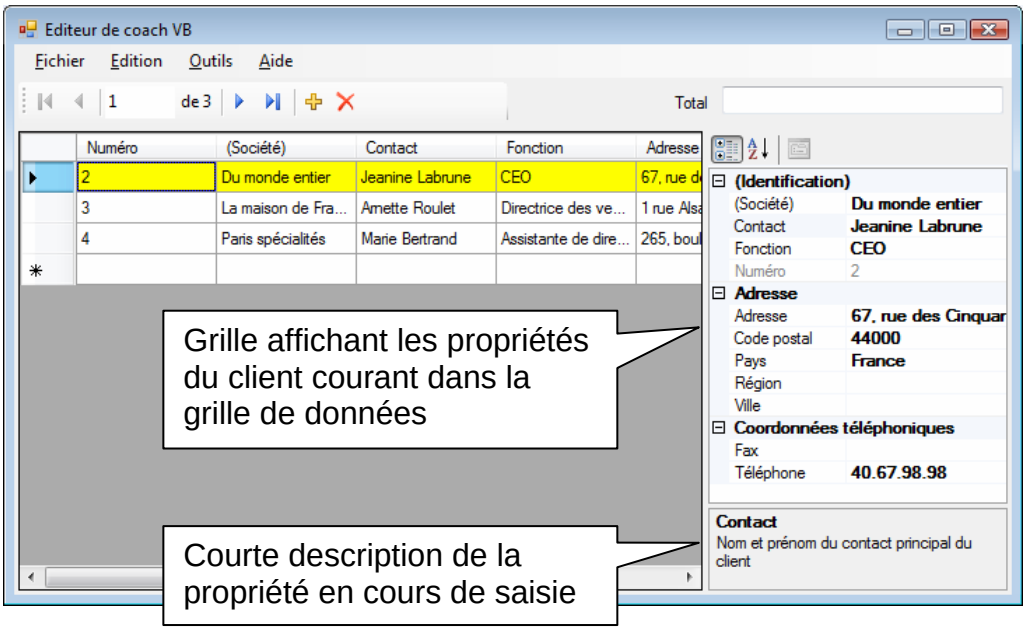
Dans ce sixième atelier, le besoin fonctionnel reste inchangé : il s'agit toujours de gérer une liste de clients.

La principale différence avec l'atelier précédent est que les données ne seront plus stockées en base de données. Dans cet atelier, nous nous contenterons d'ailleurs d'une fonctionnalité de création d'une nouvelle liste de clients vierge, qui n'est autre que la même fonctionnalité dont nous disposions jusqu'à l'atelier 4.



C'est dans l'atelier 7 que nous verrons comment rendre persistantes ces données à l'aide de la technologie XML.

La deuxième partie de cet atelier concerne l'ajout d'une grille de propriétés à droite de la grille de données de l'éditeur pour faciliter la saisie de l'utilisateur. Les propriétés d'un client y sont en effet rangées par catégories, et explicitées par une courte description.



Contexte technique

Si le contexte fonctionnel reste quasiment le même, en revanche, du point de vue programmation, nous allons adopter une approche plus « moderne » comparée à celle vue dans les ateliers précédents, qui consiste à développer à base d'objets. L'objectif est de simplifier le codage et d'augmenter la lisibilité, la maintenance et la réutilisation du code.

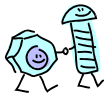
A la fin de cet atelier, vous saurez comment :

- Différencier un objet et une classe,
- Créer et manipuler des objets personnalisés,
- Utiliser l'héritage,
- Utiliser des attributs pour enrichir vos objets.

La solution de cet atelier est disponible dans le répertoire **..\Atelier 6\Solution**. Les fichiers utiles, auxquels font référence les exercices sont disponibles dans le répertoire **..Atelier 6\Fichiers utiles**.

2 Créer et utiliser des objets personnalisés

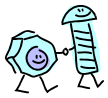
Objectif
L'objectif de ce premier exercice est d'apprendre à créer des objets personnalisés.



Qu'est ce qu'on entend par **objets personnalisés** ?
En fait, en tant que développeur, développer des **objets** est un abus de langage car ce n'est pas tant des objets que nous programmons, mais des **classes** qui serviront à la création des objets en mémoire lors de l'exécution de l'application. Les deux termes sont souvent confondus alors qu'ils sont pourtant bien distincts :

- Une *classe* décrit la structure d'un objet. C'est un *type de données* personnalisé.
  Par exemple une classe 2CV (deux chevaux) représente un type de voiture.
- Un *objet* est ce qu'on appelle une **instance** de la classe, c'est-à-dire une copie exacte et distincte de la classe en mémoire (comme une variable d'un type donné).
  C'est votre 2CV rangée au garage... et votre voisin en a peut-être un autre exemplaire (correspondant à une autre instance mémoire) dans le sien. Les deux sont des 2CV mais chacune a son existence propre.

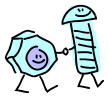
Développer ses objets personnalisés revient donc avant tout à développer des classes qui vont servir à représenter la définition des objets qui seront manipulés par l'application, comme une liste de clients par exemple dans notre cas.



Ca sert à quoi une **classe** au juste ?
L'un des principes de la programmation orientée objet qui permet de bien comprendre à quoi sert une classe (et donc les objets) est l'**encapsulation**.
Est-ce que vous avez besoin de connaître parfaitement le fonctionnement d'une 2CV pour la conduire ? Forte heureusement (pour moi), la réponse est non. Pourquoi ? Parce que les détails de fabrication et de fonctionnement vous sont masqués (ou *encapsulés*).
Une classe est l'unité qui regroupe toutes les données utiles au fonctionnement d'une 2CV pour nous permettre de la conduire.

De même, la programmation orientée objet consiste à organiser les données d'un programme et leur traitement sous la forme de classes de

façon à en faciliter la manipulation. C'est le rôle de l'architecte d'application de modéliser les objets de l'application en fonction des scénarios fonctionnels identifiés. Attention, pour concevoir les objets utiles d'une application, bien plus qu'au niveau des *données*, l'architecte raisonne en se basant sur les *services* que va consommer l'application pour répondre aux scénarios utilisateurs.



Le contenu d'une classe est organisé un peu comme un programme traditionnel : vous y trouvez des variables, des procédures, etc… enfin tout un tas de choses qu'on appelle les **membres** de la classe. Principalement ce sont :

Membre	***Définition***	***Exemple de la 2CV***
Propriétés et Champs	*Informations* caractérisant l'objet	Couleur de la voiture, type d'essence…etc
Méthodes	*Actions* que peut effectuer l'objet	Démarrer, reculer, stopper…etc
Evènements	*Evènements* qui se produisent sur l'objet	Tomber en panne, gagner un grand prix…etc

Nous aurons l'occasion de revenir plus longuement sur chacun de ces éléments dans cet atelier.

2.1 Créer une bibliothèque de classes

Une règle de bonne programmation est d'isoler toutes les définitions de types de données dans des bibliothèques de classes (ou ***dlls***), en regroupant ceux-ci généralement en fonction des services qu'ils fournissent. Par exemple, une bibliothèque de classe **Coach.Data.dll** contiendrait toutes les classes en relation avec la gestion des données. Dans notre cas, nous allons développer des types personnalisés que nous allons ajouter à une librairie nommée **Coach.Types.dll**.

Dans cet exercice, vous allez apprendre à :

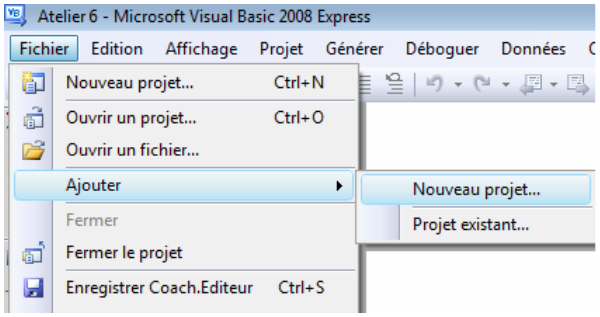
- Créer une bibliothèque de classes.

Déroulement de l'exercice :

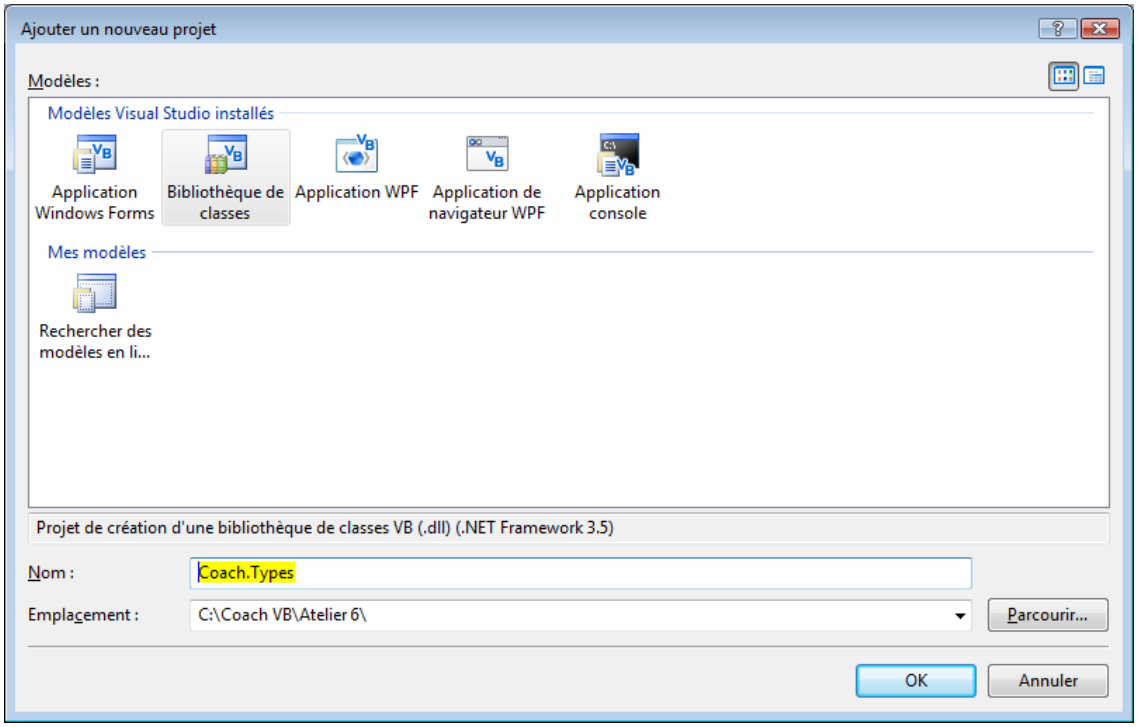
1. Ouvrez le projet précédent réalisé lors de l'atelier 5 :
   - Lancez Visual Studio à partir du menu **Démarrer** > **Tous les programmes** > **Microsoft Visual Basic 2008 Express Edition**.
   - Menu **Fichier** > **Ouvrir un projet**.

- Retrouvez le fichier **Atelier 5.sln** que vous avez créé lors de l'atelier 5 ou, si vous n'avez pas fait l'atelier précédent, récupérez le projet de démarrage dans le répertoire : **..\Atelier 6\Démarrage\Atelier 6.sln**.

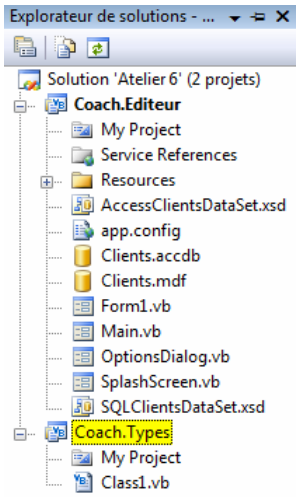
2. Créez un nouveau projet de type bibliothèque de classes au sein de la solution existante :
   - Dans le menu de Visual Studio, sélectionnez le menu **Fichier** > **Ajouter** > **Nouveau projet…** pour créer un nouveau projet dans la solution de développement en cours.



- Dans la boîte **Ajouter un nouveau projet**, sélectionnez le modèle de projet **Bibliothèque de classes** et indiquez le nom du projet, par exemple **Coach.Types**.



- Cliquez le bouton **OK**. L'explorateur de solutions affiche maintenant deux projets : **Coach.Editeur** et **Coach.Types** :



- Cliquez dans la **barre de menu Standard** de Visual Studio pour sauvegarder l'ensemble de la solution.

2.2 Créer une classe

Dans cet exercice, vous allez créer la toute première classe de la bibliothèque Coach.Types.

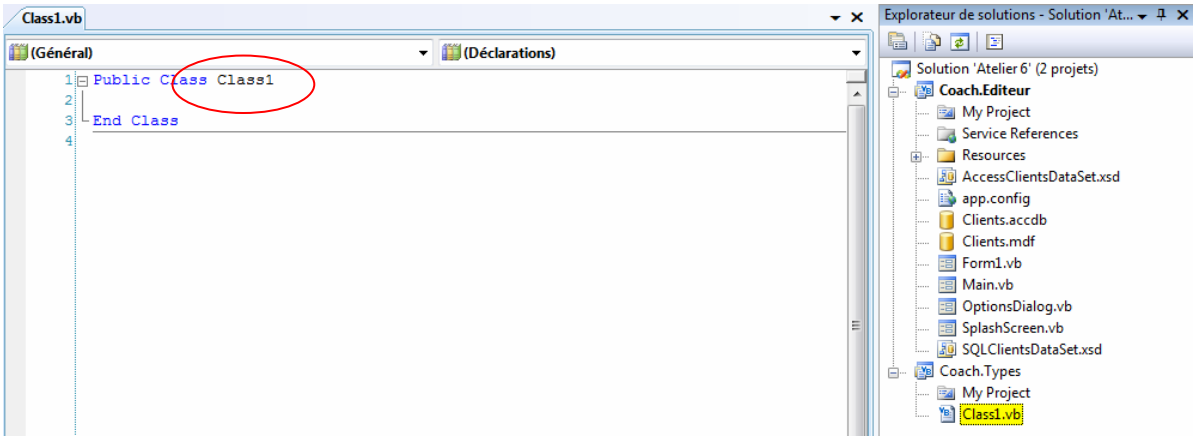
Dans cet exercice, vous allez apprendre à :

- Définir une classe et ses membres,
- Définir l'accessibilité d'un membre de classe.

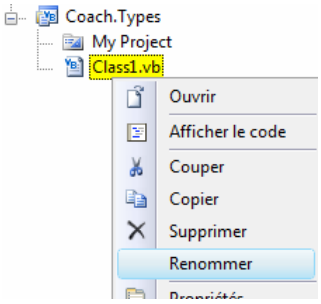
Déroulement de l'exercice :

1. Créez une classe Client pour gérer tout ce qui réfère à la notion de client :

La déclaration du type est presque toute prête (merci Visual Studio) puisque le modèle de projet **Bibliothèque de classes** sélectionné précédemment contient par défaut un fichier avec la déclaration d'une classe nommée **Class1** :

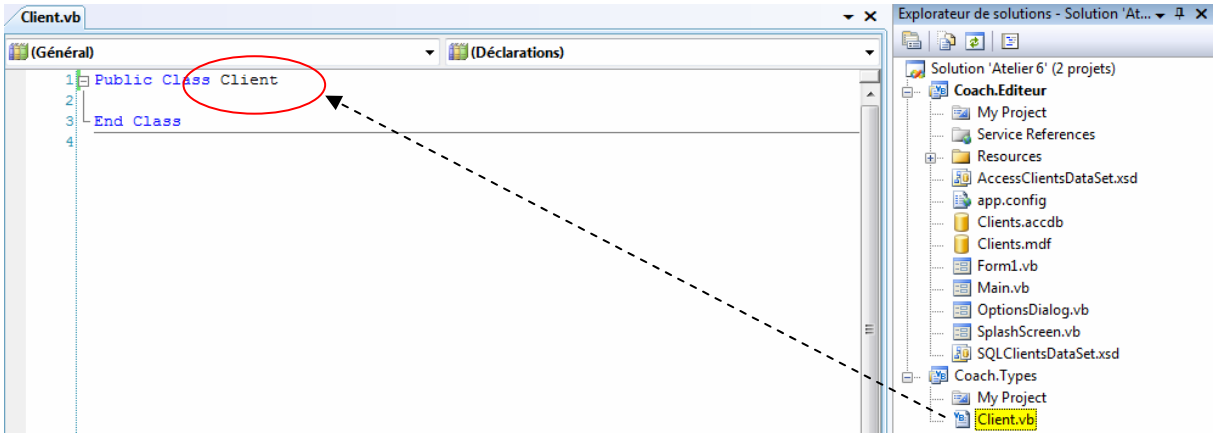


- Il ne nous reste qu'à renommer la classe. Pour cela, faites un clic droit sur le fichier **Class1.vb** dans l'**Explorateur de solutions** > **Renommer** :



- Nommez le fichier en : **Client.vb**.

Notez qu'en même temps que le fichier, Visual Studio renomme automatiquement la classe avec le même nom : très pratique !
Mais n'oubliez pas qu'un fichier peut contenir plusieurs éléments, dont plusieurs classes. A l'inverse une classe peut être définie dans plusieurs fichiers physiques distincts par le biais du mot clé **Partial** (on parle alors de classe *partielle*, cf. l'atelier 2).



Pour tout savoir sur l'instruction **Class** de déclaration d'une classe :
http://msdn.microsoft.com/fr-fr/library/wa0hwf23.aspx

Tout le code qui définit la classe doit donc être positionné entre les mots clés **Class** et **End Class** ! Tout ce que vous mettrez en dehors de ce bloc fera partie d'un autre type.

Peut-être avez-vous entendu parler d'un autre type de données similaire aux classes appelé **structure** ?
C'est vrai qu'il s'agit d'un type équivalent mais il présente quand même de grandes différences avec les classes. L'un des principaux inconvénients est qu'une structure est un type valeur alors qu'une classe est un type référence, et il y en a d'autres...

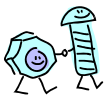
Pour tout savoir sur le type structure :
http://msdn.microsoft.com/fr-fr/library/cec05s9z.aspx
Pour comparer les classes et les structures :
http://msdn.microsoft.com/fr-fr/library/2hkbth2a.aspx

2. Créez les propriétés de la classe Client sur la base des informations associées à un client :

Il s'agit maintenant de définir le contenu de la classe, c'est-à-dire ce qu'on appelle ses **membres**.
Nous allons commencer par ajouter à la classe **Client** des **propriétés**. Elles constituent les « variables » de la classe, c'est-à-dire des données utiles destinées à être stockées au sein des objets de type Client.
Voici une description des propriétés dont nous avons besoin pour définir un objet Client avec leur type de données associé :

Nom	*Description*	*Type*
Id	Code d'indentification du client	integer
Entreprise	Raison sociale de l'entreprise	string
Nom	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

Vous reconnaissez les différentes colonnes d'un enregistrement **Client**, tel qu'il était défini en base dans l'atelier précédent. Nous allons juste modifier le type de l'identifiant (Id) d'un client pour manipuler des entiers (Integer) plutôt que des chaînes de caractères comme précédemment.
Mais attention, même si les propriétés de notre classe ressemblent parfaitement à la structure de la table **Client** en base de données, il n'y a aucun parallélisme direct. Dans notre cas, cela s'explique par le fait que nous sommes en train de construire une couche d'objets dit *métier* dans laquelle on retrouve notre modèle de données. Mais retenez que les classes sont des éléments pouvant définir toute sorte d'objets, en général alignés sur des services, plutôt que sur des données.



Faut-il faire une **propriété** ou **un champ** ?
Pour stocker une donnée d'une classe, il y a deux choix :

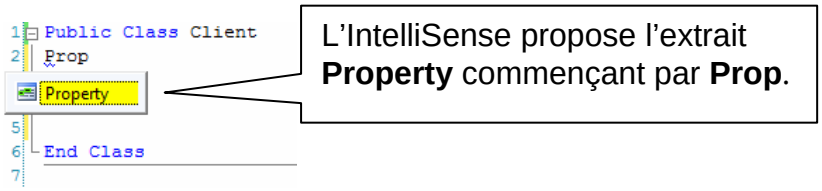
- Soit on définit un **champ** qui s'implémente exactement comme une variable.
- Soit on définit une **propriété** qui donne davantage de souplesse au moment de l'attribution et de l'extraction de la donnée. En effet, une propriété s'implémente via deux procédures conjointes **Get** et **Set** qui donnent un contrôle respectivement au moment de la *lecture* et de *l'écriture* de la variable. Comme ces deux procédures interviennent au

moment où on *accède* à la variable, on dit que ce sont des **accesseurs** de la propriété. C'est cette deuxième option que nous allons retenir pour notre classe.

Pour savoir quand utiliser une **propriété** ou un **champ** :
http://msdn.microsoft.com/fr-fr/library/9d65as2e.aspx

Pour créer une première propriété dans la classe, nous allons exploiter une fois de plus les extraits de code de Visual Studio.
L'extrait de code pour créer une propriété peut être chargé en tapant le mot **Property** suivi de la touche **Tabulation** (**TAB**).

- Positionnez le curseur quelque part entre le début et la fin de la classe **Client**.
- Tapez **Prop** :



- Tapez deux fois de suite la touche **TAB**, une première fois pour valider le mot complet **Property** proposé par l'IntelliSense et une seconde fois pour faire apparaître l'extrait de code.

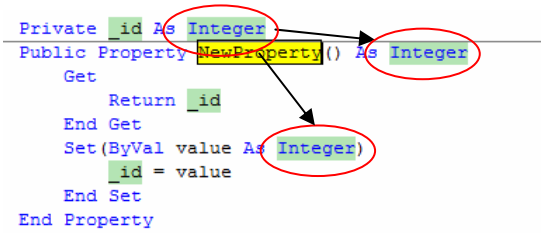
```
Public Class Client

    Private newPropertyValue As String
    Public Property NewProperty() As String
        Get
            Return newPropertyValue
        End Get
        Set(ByVal value As String)
            newPropertyValue = value
        End Set
    End Property


End Class
```

- Entrez **_id** dans la première zone en surbrillance pour remplacer **newPropertyValue**.
- Tapez une fois sur la touche **TAB** pour basculer sur le deuxième élément variable de l'extrait.

```
Private _id As String
Public Property NewProperty() As String
    Get
        Return _id
    End Get
    Set(ByVal value As String)
        _id = value
    End Set
End Property
```

Notez que l'extrait de code s'est modifié de façon à ce que les deux accesseurs **Set** et **Get** référencent maintenant **_id** à la place de **newPropertyValue**.

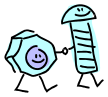
- Tapez **Integer** pour remplacer le type de données **String** proposé par défaut.
- Tapez une fois sur la touche **TAB** pour basculer sur le nom de la propriété.



```
Private _id As Integer
Public Property NewProperty() As Integer
    Get
        Return _id
    End Get
    Set(ByVal value As Integer)
        _id = value
    End Set
End Property
```

- Tapez **Id** pour remplacer **NewProperty**.
- Positionnez vous après **End Property**. Tapez une fois **Entrée** pour ajouter une ligne vide et désactiver ainsi l'extrait de code qui ne doit plus présenter de zones vertes.

```
Public Class Client

    Private _id As Integer
    Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)
            _id = value
        End Set
    End Property


End Class
```



Pour manipuler la propriété d'un objet, la syntaxe est la suivante :

<instance de la classe>.<Nom de la propriété>

Par exemple, pour manipuler la propriété **Id** d'un objet de type **Client** nommé **ContactPrincipal**, cela donne : **ContactPrincipal.Id**

Bravo ! Vous avez créé une première propriété **Id** au sein de la classe **Client**. Mais qu'avons-nous défini en somme ?

- L'accesseur **Get** est une procédure qui s'exécute avant que la valeur de la propriété ne soit extraite, c'est-à-dire chaque fois que vous utilisez l'écriture du type :

Lecture de la valeur de la propriété **Id** pour l'affecter à une variable X.

<une variable X quelconque> = MonObjet.Id

- L'accesseur **Set** est la procédure inverse qui s'exécute avant qu'une valeur ne soit attribuée à la propriété, c'est-à-dire chaque fois que vous utilisez l'écriture du type :

Ecriture dans la propriété **Id** d'une nouvelle valeur Y.

MonObjet.Id = <une valeur Y quelconque>

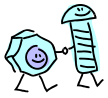
La valeur **Y** est transmise automatiquement à l'accesseur **Set** via le paramètre **value** de la procédure.

```
Set(ByVal value As Integer)
    _id = value
End Set
```

- **_id** est un champ privé de la classe, au cœur du mécanisme des accesseurs. C'est par lui que les accesseurs peuvent agir sur la propriété avant qu'elle ne soit attribuée ou extraite.

Que se passe-t-il si vous omettez un accesseur ?

- Si la propriété n'a pas d'accesseur **Set**, cela signifie qu'elle est en lecture seule. Pratique, non ?
- Si au contraire elle n'a pas d'accesseur **Get**, cela signifie que la propriété est en écriture seule !



C'est quoi cette histoire de membre *privé* ?
En effet, la définition du champ **_id** débute par le mot **Private** alors que celle de la propriété **Id** commence par **Public**.

Private et **Public** sont appelés **modificateurs d'accès** car ils définissent l'**accessibilité** du membre, c'est-à-dire s'il est visible depuis l'extérieur de la classe ou non. Ainsi :

- **_id** est un membre privé donc ne sert qu'à l'usage interne de la classe et ne peut pas être utilisé sur un objet de type **Client**.
- C'est la propriété **Id** qui est un membre public donc utilisable sur tout objet de type **Client**.

Ils existent plusieurs mots clés pour caractériser l'accessibilité d'un membre :

Mot clé	*Indique que le membre est :*
Public	Visible depuis l'assemblage qui le contient, et de toutes les assemblages référençant l'assemblage courant.
Private	Visible uniquement de l'intérieur de la classe courante.
Friend	Visible depuis l'assemblage qui le contient.
Protected Friend	Visible depuis l'assemblage qui le contient, ou de l'un des descendants de la classe courante.
Protected	Visible de l'intérieur de la classe courante, ou de l'un des descendants de la classe courante.

msdn

Pour connaître les niveaux d'accès dans Visual Basic :
http://msdn.microsoft.com/fr-fr/library/76453kax.aspx

- Recommencez l'opération d'insertion de propriétés pour toutes les autres propriétés à ajouter à la classe **Client.**

Deux règles de bonne programmation :

- Une fois toutes les propriétés insérées, vous pouvez regrouper en début de classe les déclarations des champs privés. Ce n'est techniquement pas obligatoire, mais vous y gagnerez en lisibilité du code.
- Une autre règle de bonne programmation consiste à assigner les champs privés à une valeur par défaut, lors de leur déclaration. Par exemple :

- les variables de type **Integer** à **0**,
- et les **String** à la propriété **Empty** de la classe **String**.

Vous devez obtenir le code suivant :

Code VB

```
Public Class Client

    Private _id As Integer = 0
    Private _entreprise As String = String.Empty
    Private _nom As String = String.Empty
    Private _titre As String = String.Empty
    Private _adresse As String = String.Empty
    Private _ville As String = String.Empty
    Private _region As String = String.Empty
    Private _codepostal As String = String.Empty
    Private _pays As String = String.Empty
    Private _telephone As String = String.Empty
    Private _telecopie As String = String.Empty

    Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)
            _id = value
        End Set
    End Property

    Public Property Enterprise() As String
        Get
            Return _entreprise
        End Get
        Set(ByVal value As String)
            _entreprise = value
        End Set
    End Property

    Public Property Nom() As String
        Get
            Return _nom
        End Get
        Set(ByVal value As String)
            _nom = value
        End Set
    End Property

    Public Property Titre() As String
```

Regroupement de toutes les déclarations de membres privés associés aux accesseurs de propriétés.

```
    Get
        Return _titre
    End Get
    Set(ByVal value As String)
        _titre = value
    End Set
End Property

Public Property Adresse() As String
    Get
        Return _adresse
    End Get
    Set(ByVal value As String)
        _adresse = value
    End Set
End Property

Public Property Ville() As String
    Get
        Return _ville
    End Get
    Set(ByVal value As String)
        _ville = value
    End Set
End Property

Public Property Region() As String
    Get
        Return _region
    End Get
    Set(ByVal value As String)
        _region = value
    End Set
End Property

Public Property CodePostal() As String
    Get
        Return _codepostal
    End Get
    Set(ByVal value As String)
        _codepostal = value
    End Set
End Property

Public Property Pays() As String
    Get
        Return _pays
    End Get
    Set(ByVal value As String)
        _pays = value
```

```
        End Set
    End Property

    Public Property Telephone() As String
        Get
            Return _telephone
        End Get
        Set(ByVal value As String)
            _telephone = value
        End Set
    End Property

    Public Property Telecopie() As String
        Get
            Return _telecopie
        End Get
        Set(ByVal value As String)
            _telecopie = value
        End Set
    End Property

End Class
```

- Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
- Compilez le projet pour vérifier que la génération se déroule sans souci.

Bravo ! Vous venez de créer la structure d'un objet de type **Client** personnalisé !

2.3 Créer une collection d'objets

En réalité, ce dont nous avons besoin, c'est de manipuler une liste de plusieurs clients…Comment gérer un ensemble d'objets d'un type donné ?

Dans cet exercice, vous allez apprendre à :

- Définir une liste d'objets,
- Utiliser un type générique du Framework .NET,
- Hériter d'une classe de base.

Déroulement de l'exercice :

Comme pour de nombreux autres langages, il est possible avec Visual Basic de regrouper plusieurs données d'un même type sous la forme d'un tableau indexé. Mais cette solution peut s'avérer couteuse en performance si le nombre d'éléments maximum que devra contenir le tableau ne peut pas être défini à l'avance.
Heureusement le Framework .NET est riche en types constituant une alternative aux tableaux, tels que les dictionnaires, les listes, les files d'attente ou encore les piles.

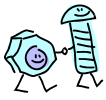
Pour tout savoir sur les tableaux dans Visual Basic :
http://msdn.microsoft.com/fr-fr/library/wak0wfyt.aspx
Pour connaître quelles sont les alternatives aux tableaux :
http://msdn.microsoft.com/fr-fr/library/e1ad18x6.aspx
Un lien vers l'espace de noms **System.Collections** pour connaître les différences entre les collections d'objets qu'il fournit :
http://msdn.microsoft.com/fr-fr/library/system.collections.aspx

Dans notre cas, ces objets ne sont pas tout à fait satisfaisant dans la mesure où ils manipulent des objets non typés c'est-à-dire de n'importe quel type. Cela demande un travail supplémentaire pour contrôler le type de l'objet chaque fois qu'il est lu dans la collection.
Or nous, nous connaissons parfaitement le type d'objets que nous voulons utiliser, à savoir **Client**.
Nous allons donc piocher dans une autre catégorie de collections du Framework .NET qu'on dit **génériques** parce qu'elles permettent de fonctionner sur un type de données précis.

Alors pourquoi le terme *générique* ?
La généricité vient du fait qu'au moment de la définition de la collection, on ne sait pas encore de quel type d'objet il s'agit. La collection se comporte d'ailleurs de manière identique quelque soit le type d'objets qu'elle manipule. C'est seulement lorsqu'on définit une instance de la collection qu'on indique le type de données qu'elle contient.
La collection qui nous intéresse est une classe **List(T)** où **T** représente le type d'objet que nous allons manipuler, ici le type **Client**.

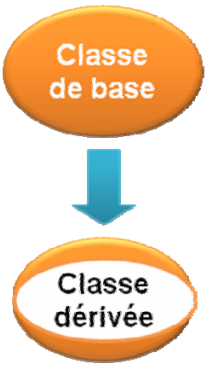
Pour tout savoir sur la classe **List(T)** :
http://msdn.microsoft.com/fr-fr/library/6sh2ey19.aspx
Pour creuser la question des génériques, rendez-vous au § 4.1 de cet atelier.

Dernier point avant de nous lancer à coder…
Nous n'allons pas utiliser le type **List(T)** tel quel car une telle liste ne fournit pas l'intégralité du comportement dont nous avons besoin pour gérer notre liste de clients. Par exemple, nous allons avoir besoin d'incrémenter de manière automatique l'identifiant (Id) d'un client à chaque nouveau client inséré dans la liste.
L'idéal serait de créer une classe nommée **Clients** qui reprendrait rigoureusement les caractéristiques du type **List(T)** moyennant quelques fonctionnalités complémentaires…
En programmation orientée objet, ce mécanisme est possible et s'appelle l'**héritage**.



C'est quoi l'**héritage** ?
C'est une association entre deux classes qui assure l'utilisation par l'une des membres déjà définis dans l'autre. La classe hérite de tous les membres, aussi bien des propriétés que des méthodes, des évènements etc… D'un point de vue terminologie, on dit que la classe qui hérite

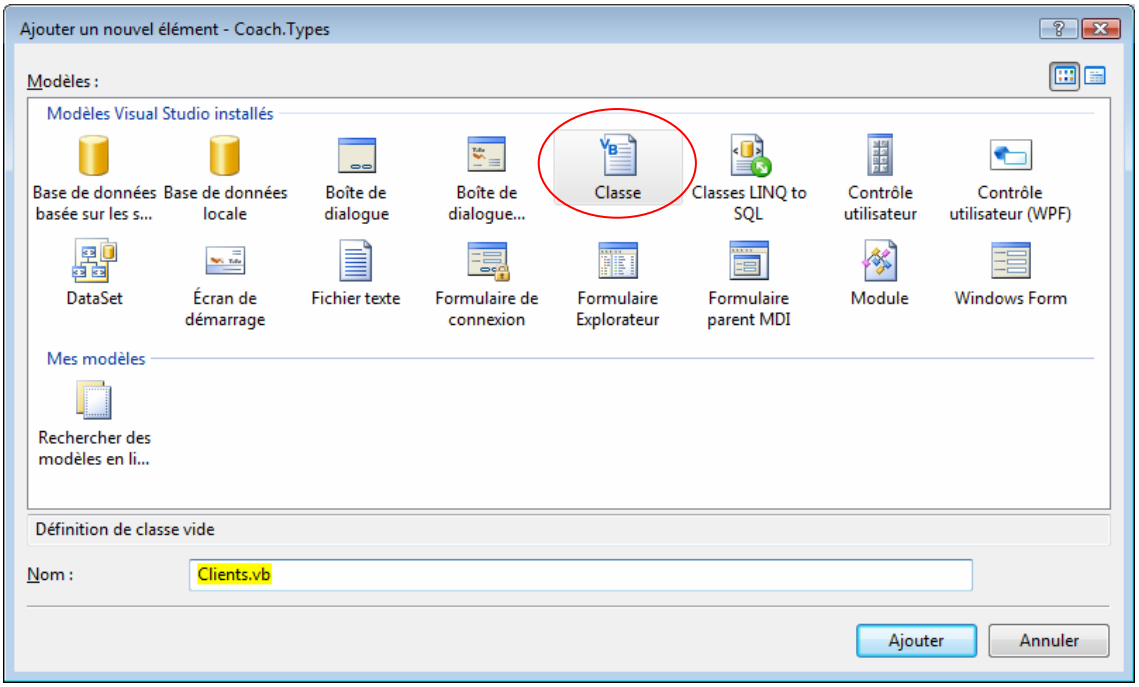
dérive de la première que l'on appelle la *classe de base*.



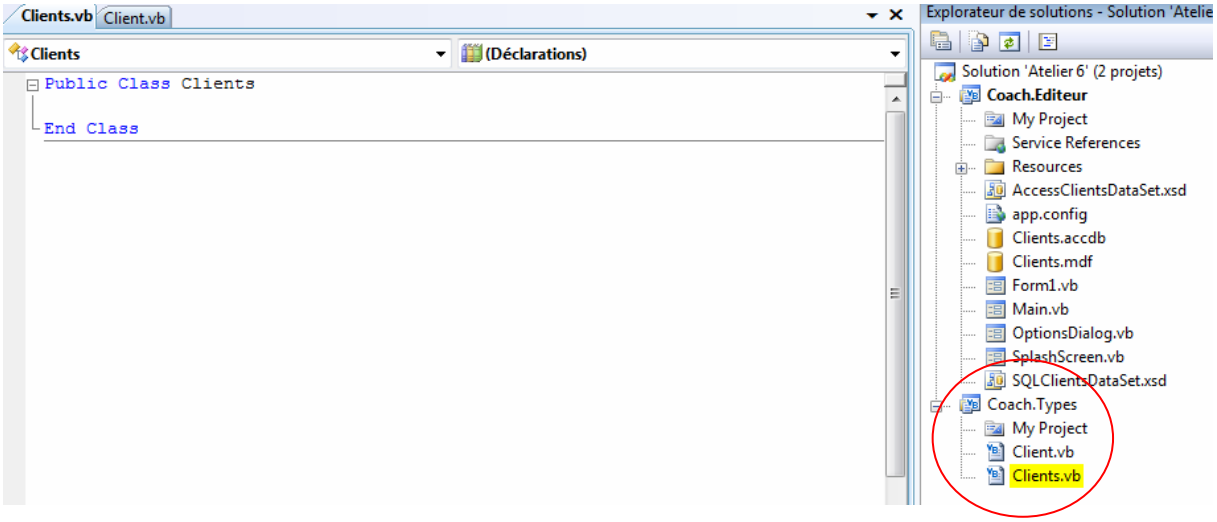
Attention ! Le langage Visual Basic (comme tous les langages .NET) n'autorise une classe qu'à **hériter d'une seule classe**. Lorsqu'aucune association d'héritage n'est précisée (ce qui est le cas par exemple pour notre classe **Client**), la classe hérite implicitement de la classe **Object** que vous trouverez dans le Framework .NET dans l'espace de nom **System**.

Pour en savoir plus sur la classe de base fondamentale **System.Object** à la racine de la hiérarchie des types du Framework .NET :
http://msdn.microsoft.com/fr-fr/library/system.object_members.aspx

1. Créez une nouvelle classe nommée Clients pour gérer le groupe d'objets Client dont nous avons besoin :
   - Dans l'**Explorateur de solutions**, faites un clic droit sur le projet **Coach.Types** > **Ajouter** > **Nouvel élément…**
   - Dans la fenêtre **Ajouter un nouvel élément**, sélectionnez le modèle Classe.
   - Nommez la classe **Clients** (avec un s) :



- Cliquez **Ajouter** pour ajouter la nouvelle classe au projet **Coach.Types** :

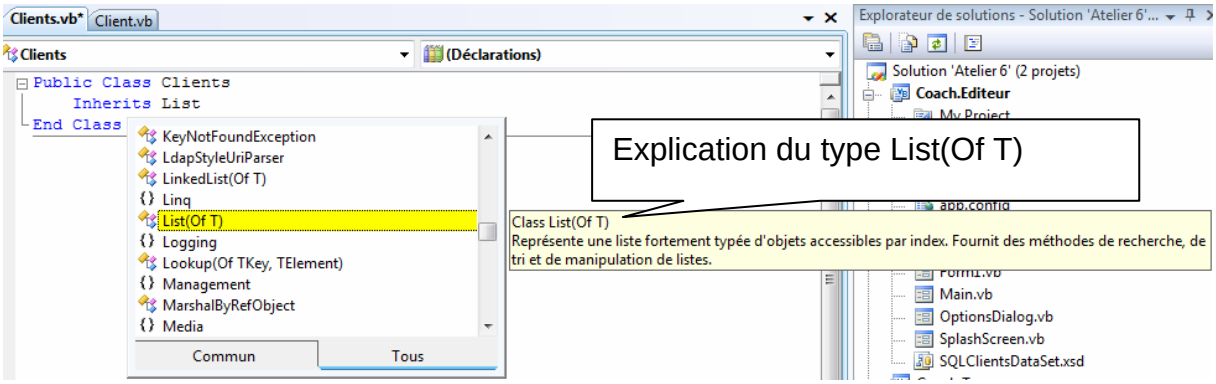


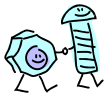
Une règle de bonne programmation consiste à ne mettre un **s** qu'aux classes destinées à créer des listes ou des collections d'objets. Ainsi, il n'y a pas d'ambiguïté lors de la relecture du code (notamment en cas de maintenance). **Client** désigne donc un objet simple, alors que **Clients** est une liste de **Client**.

Avez-vous remarqué que par défaut une classe est créée avec le modificateur d'accès **Public** ? Pensez à corriger l'accessibilité de la classe si au contraire vous ne souhaitez pas exposer des objets internes à l'application…

- Utilisez le mot clé **Inherits** de Visual Basic pour faire *hériter* la classe **Clients** de la classe **List(T)** du Framework .NET.

Attention : l'héritage se définit sur la ligne immédiatement après la ligne de déclaration de la classe !
N'hésitez pas une fois de plus à vous reposer sur l'IntelliSense qui vous guide à chaque frappe :





Pour rappel, le principe de la liste générique **List(T)** est de fournir des méthodes pour manipuler une liste regroupant des objets d'un type **T**. Au moment de l'utilisation de la liste, vous devez indiquer au Framework .NET quel est le type d'éléments dont sera composée votre liste. Dans notre cas, il s'agit d'une liste d'objets de type **Client**.

Il faut donc écrire :

Code VB

```
Public Class Clients
    Inherits List(Of Client)
End Class
```

Bon ! Beaucoup d'explications pour trois malheureuses lignes de code au final… Mais c'est ça la magie de la programmation orientée objet et du principe de l'héritage. La classe **Clients** propose tous les membres usuels dont nous avons besoin pour manipuler une liste d'objet **Client**.

2.4 Utiliser les classes

Dans cet exercice, vous allez apprendre à :

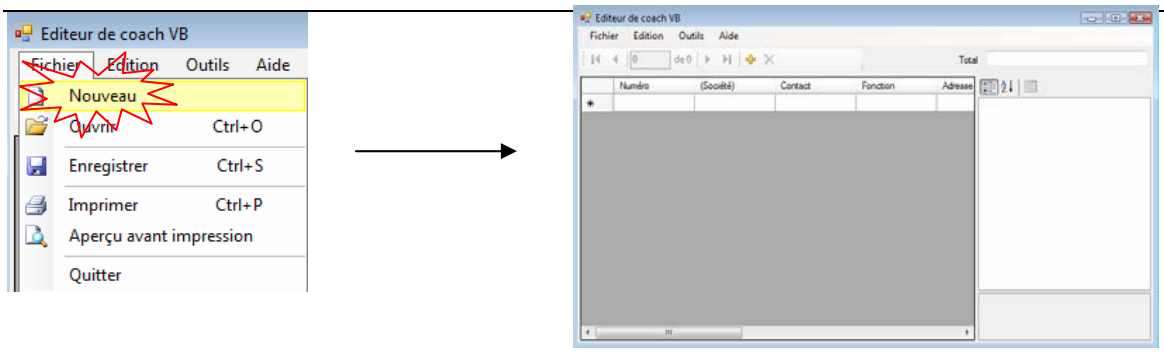
- Créer puis utiliser une instance d'un objet personnalisé,
- Définir et utiliser un membre partagé dans une classe,

- Personnaliser les accesseurs d'une propriété membre d'une classe.

Contexte fonctionnel

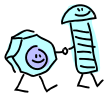
L'objectif de cet exercice est de modifier l'éditeur pour utiliser un objet de type **Clients** à la place d'un objet de type **DataTable** en tant que source de données de l'application. L'idée consiste à retirer tout le code inutile référençant la **DataTable** (retirer des lignes de code, ça fait toujours plaisir …), puis rajouter tout ce qu'il faut pour couvrir l'étendue fonctionnelle de l'application d'origine.

Dans cet atelier, nous allons momentanément mettre de côté la récupération d'un jeu de données existant ainsi que le processus de sauvegarde de celles-ci. Nous allons plus simplement rétablir le menu **Fichier** > **Nouveau** qui permet de construire un jeu de données clientes vierge.



Rassurez-vous, dans le prochain atelier, nous verrons comment rendre persistantes les informations saisies par l'utilisateur dans la grille de données de façon à pouvoir les recharger si besoin.

Déroulement de l'exercice :

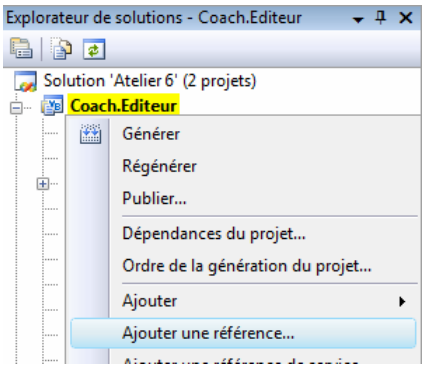


Pour utiliser les classes **Clients** et **Client** de la bibliothèque de classes **Coach.Types** à partir de notre application éditeur **Coach.Editeur**, il faut ajouter à cette dernière une référence vers la bibliothèque de classes. C'est à cette seule condition que nous pourrons accéder aux types de l'assemblage référencé.

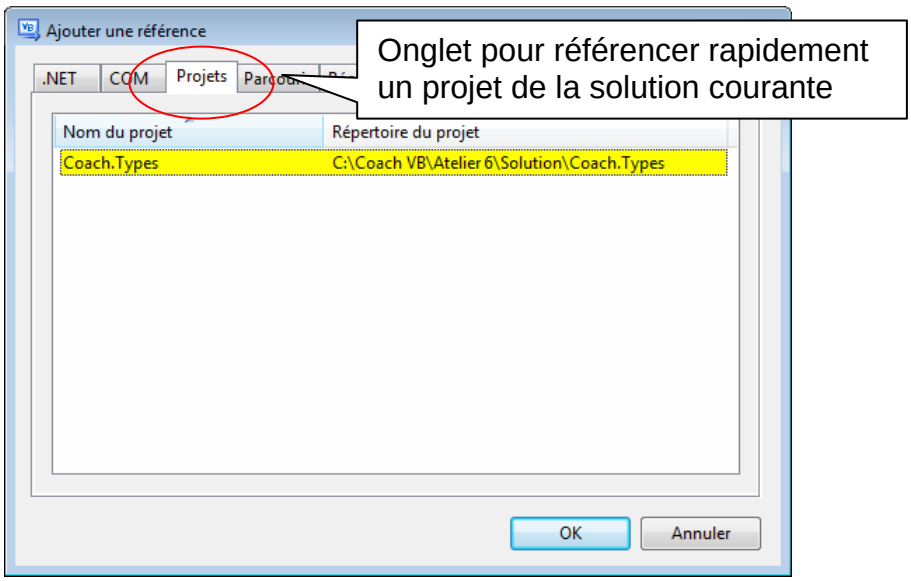
msdn

Pour tout savoir sur la gestion des références dans un projet :
http://msdn2.microsoft.com/fr-fr/library/s5zwxxdw.aspx

1. Ajoutez au projet Coach.Editeur une référence au projet Coach.Types :
   - Dans l'**Explorateur de solutions**, faites un **clic-droit** à la racine du projet **Coach.Editeur** > **Ajouter une référence** :



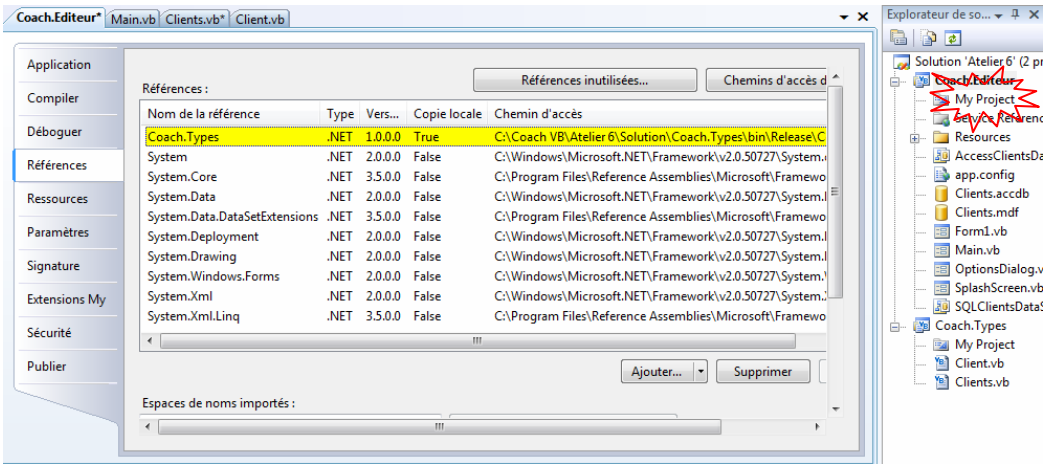
- Dans la fenêtre **Ajouter une référence**, sélectionnez l'onglet **Projets**.
- Dans la liste des projets de la solution courante proposés (normalement il n'y en a qu'un), sélectionnez le projet **Coach.Types** :



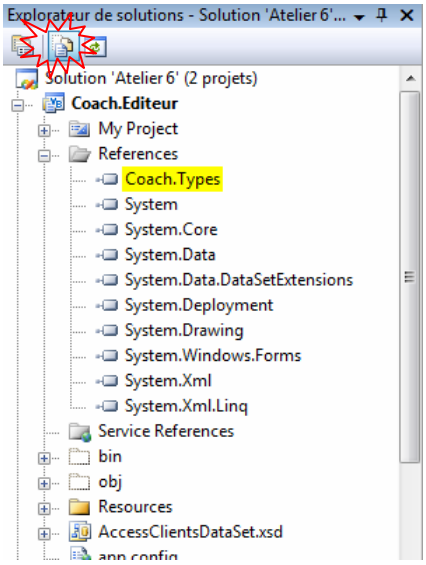
- Cliquez le bouton **Ok**.

Vous pouvez contrôler la liste des références d'un projet :

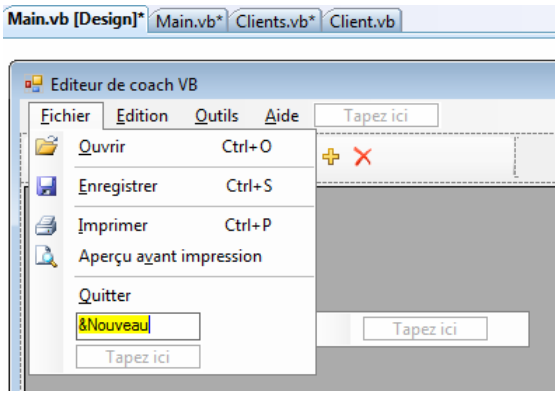
- Soit en affichant le **Concepteur de projet** en double cliquant sur le nœud **My Project** du projet **Coach.Editeur** dans l'**Explorateur de solutions**. Cliquez ensuite l'onglet **Références** :



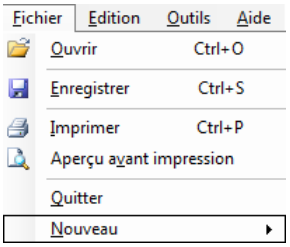
- Soit en cliquant l'icône **Afficher tous les fichiers** dans la barre d'outils de l'**Explorateur de solutions** pour faire apparaître le répertoire **Références** du projet.



2. Rajoutez maintenant une option de menu **Fichier** > **Nouveau** au menu de l'application :
   - Dans l'**Explorateur de solutions**, double cliquez sur le fichier **Main.vb** pour afficher le formulaire dans le **Concepteur de vue**.
   - Cliquez le menu **mainMenuStrip** puis l'option **Fichier** pour faire apparaître ses sous options.
   - Remplacez la zone **Taper ici** par **&Nouveau**.



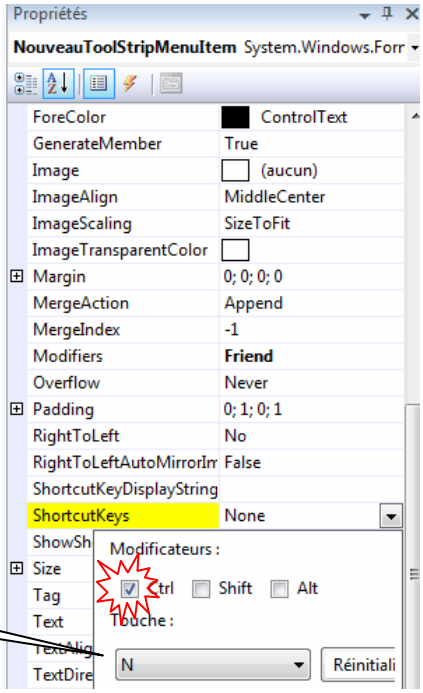
Le sigle **&** juste devant le **N** de **Nouveau** permet de définir la lettre comme *touche d'accès rapide*. La lettre s'affiche avec un souligné.



Pour connaître le fonctionnement d'une touche d'accès rapide ou apprendre comment améliorer un **ToolStripMenuItem** :
http://msdn.microsoft.com/fr-fr/library/ms171651(VS.80).aspx

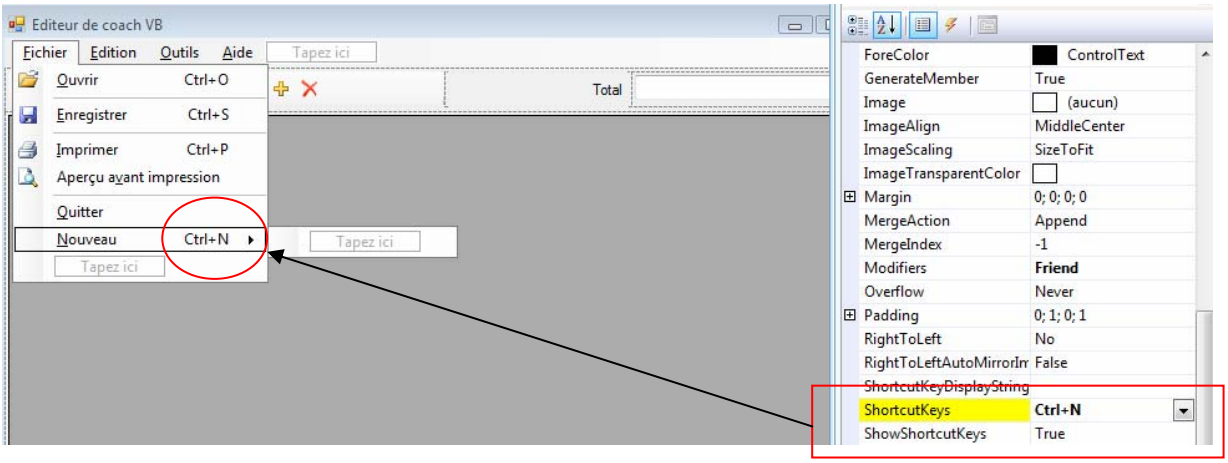
Vous pouvez aussi associer l'option de menu à un raccourci clavier en configurant ses propriétés **ShortcutKeys** et **ShowShortcutKeys**.

- Affichez la fenêtre de propriétés de l'élément **NouveauToolStripMenuItem** ainsi créé en cliquant **F4**.
- Vérifiez que la propriété **ShowShortcutKeys** est configurée à la valeur **True**.
- Configurez la propriété **ShortcutKeys** à **Ctrl+N** en vous aidant de l'éditeur de propriétés associé qui vous montre les combinaisons de touches disponibles :

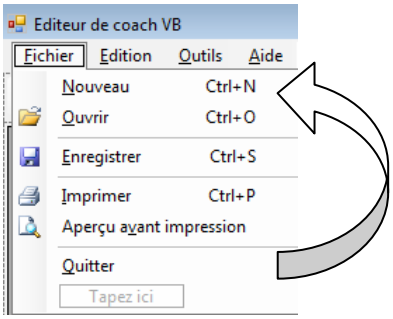


Sélectionnez la lettre N pour l'associer à la touche Ctrl

La combinaison de touches est automatiquement ajoutée à droite du texte de l'option de menu pour informer l'utilisateur qu'un raccourci clavier est disponible sur cette option :

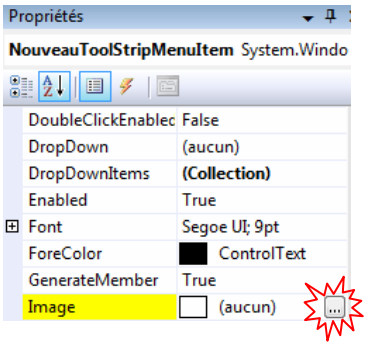


- Faites un glisser déplacer avec la souris de l'option **Nouveau** pour la déplacer juste au dessus de l'option **Ouvrir** du menu **Fichier**.

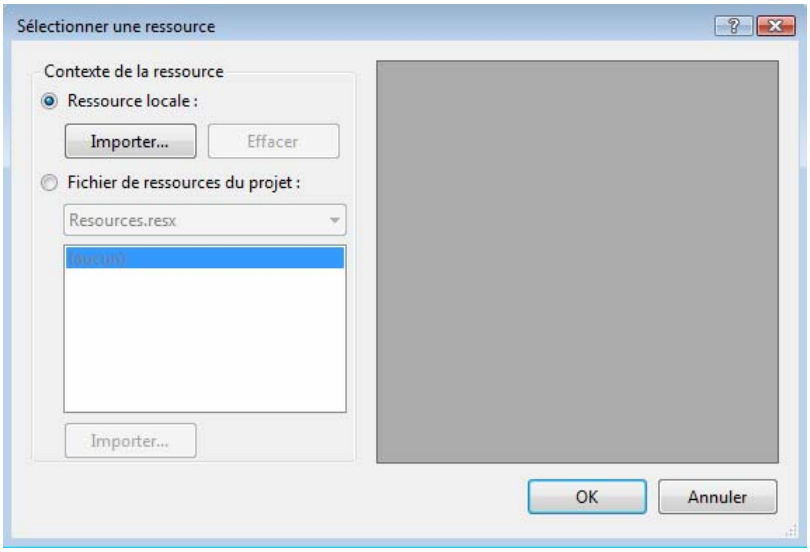


Il nous reste à ajouter également l'icône standard de l'option **Nouveau** que nous vous fournissons dans le répertoire **Fichiers utiles** de cet atelier.

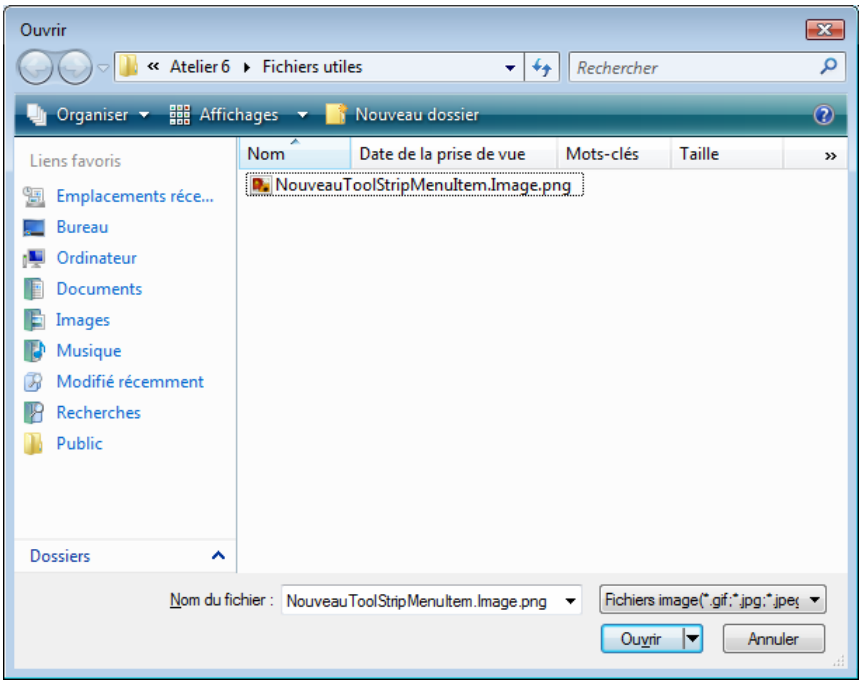
- Toujours à partir de la fenêtre de propriétés de l'option **NouveauToolStripMenuItem**, cliquez [...] dans la zone de saisie de la propriété **Image**.



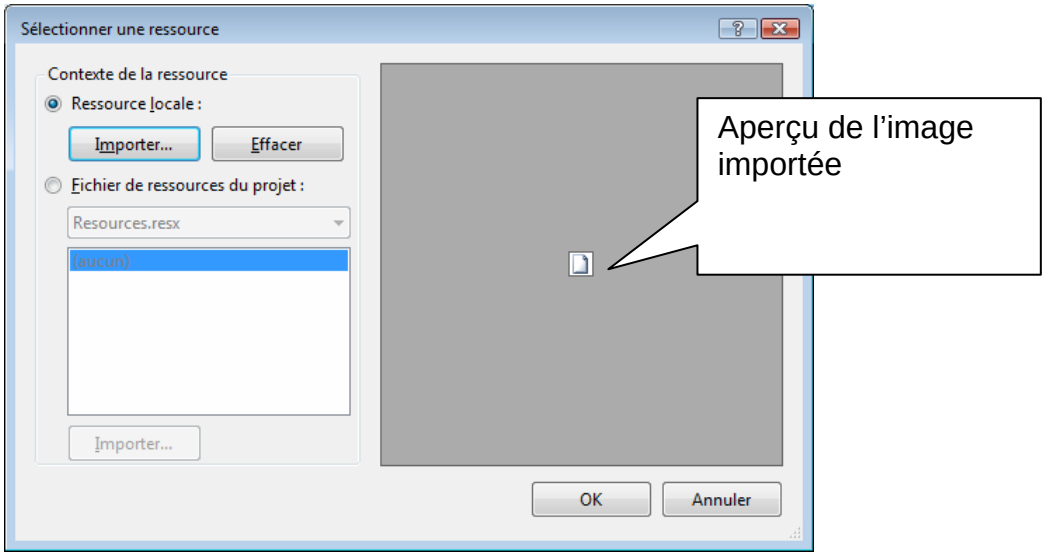
- Dans la fenêtre **Sélectionner une ressource**, cochez l'option **Ressource locale** pour inclure une nouvelle image au sein du fichier de ressource local du formulaire.



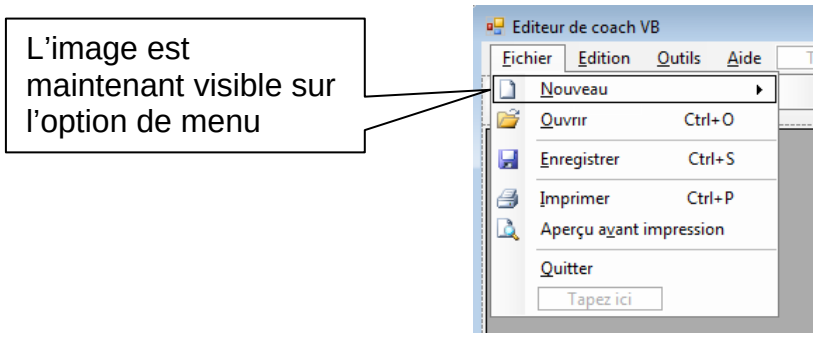
- Cliquez le bouton **Importer…** pour récupérer le fichier image sur le disque.
- Dans la fenêtre **Ouvrir**, retrouvez et sélectionnez le fichier **NouveauToolStripMenuItem.Image.png** fourni dans le dossier **…Atelier 6\Fichiers utiles** de cet atelier :



- Validez par le bouton **Ouvrir**.

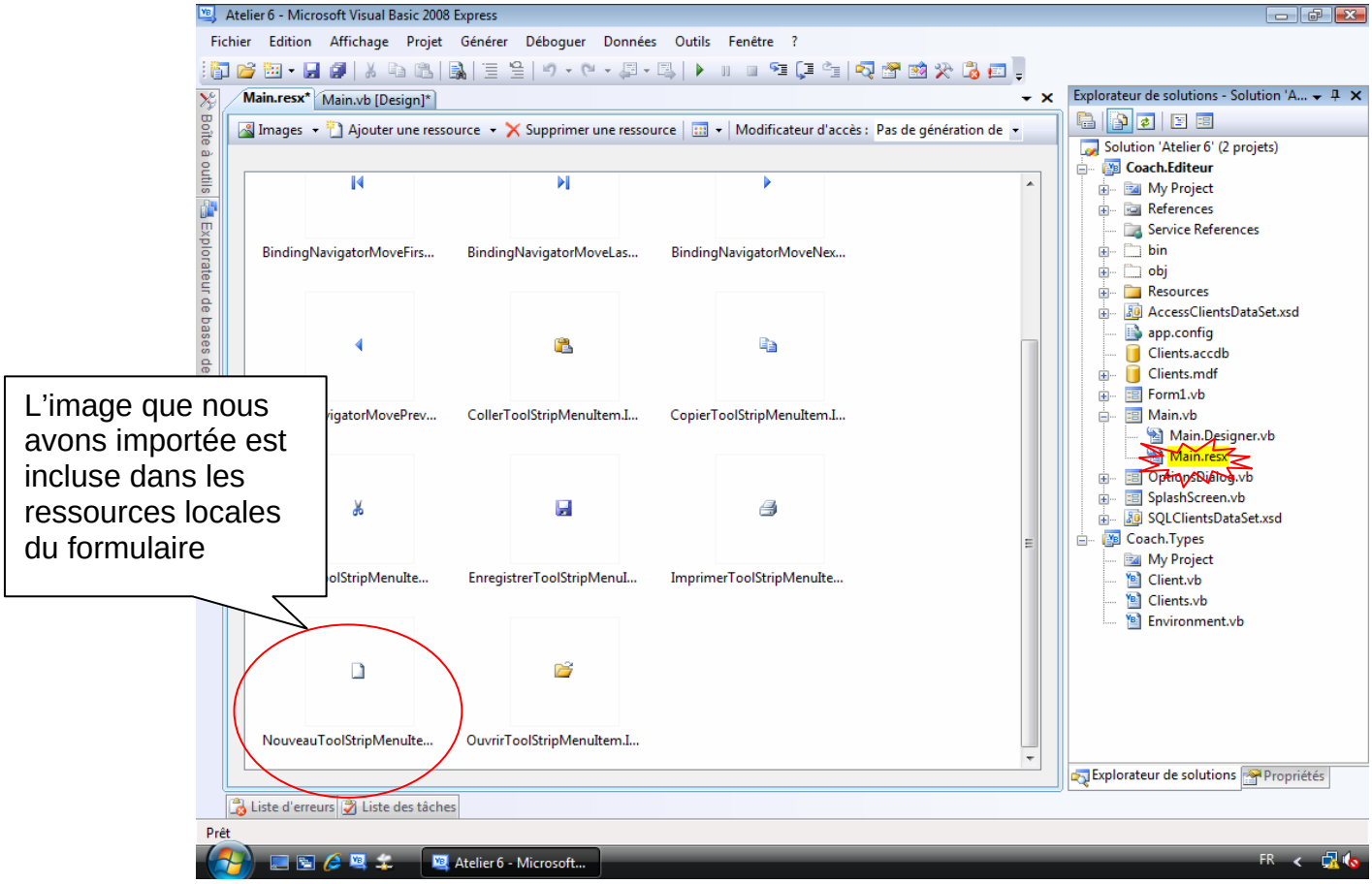


- Validez par le bouton **OK**.



Le fichier a été ajouté aux ressources locales du formulaire. Pour voir les ressources :

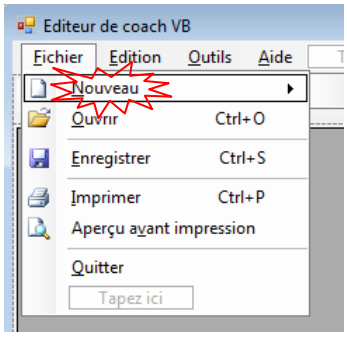
- cliquez **Afficher tous les fichiers** dans la barre d'outils de l'**Explorateur de solutions**.
- étendez le nœud **Main.vb** pour voir les fichiers directement liés.
- Double cliquez sur **Main.resx** pour voir les ressources du formulaire :



L'image que nous avons importée est incluse dans les ressources locales du formulaire

Ne pas confondre avec le dossier **Resources** du projet qui contient les ressources globales à l'ensemble du projet.

3. Codez la création d'un jeu de données clients vierge à partir du menu Fichier > Nouveau :
   - A partir du **Concepteur de formulaire**, double cliquez sur l'élément **NouveauToolStripMenuItem** pour générer un gestionnaire d'évènement associé à l'évènement **Click**.



   - Visual studio bascule sur le fichier de code **Main.vb** sur la nouvelle procédure créée :

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                                                                        Handles NouveauToolStripMenuItem.Click

End Sub
```

Reprenons le code que vous aviez écrit à l'atelier 4 de ce tutorial en réponse à ce même évènement click sur l'option **Fichier** > **Nouveau**. Il s'agissait de créer un objet **DataTable** neuf puis de le lier aux contrôles d'affichage :

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click
    'Création d'une nouvelle table mémoire vide
    Dim newDataTable As DataTable
    newDataTable = CreateDataTable()
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newDataTable
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.BindingSource = MainBindingSource
    'Mettre à jour le chemin de fichier à vide
    DataFilePath = String.Empty
    'Remettre à jour le titre du formulaire
    UpdateFormTitle(DataFilePath)
End Sub
```

- Remplacez la création du **DataTable** par notre nouvel objet **Clients** :

Code VB

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                           Handles NouveauToolStripMenuItem.Click
    Dim newClients As Clients = New Clients()
End Sub
```

Que fait cette ligne ?

- La partie à gauche de l'opérateur **=** déclare un objet de type **Clients**. Pour rappel, en utilisant les nouvelles capacités d'inférence de type de Visual Basic, vous pouvez réduire l'écriture à :

  Dim newClients = New Clients()

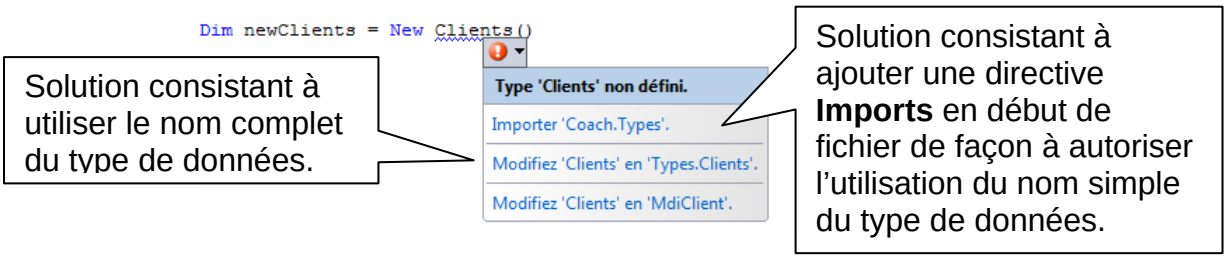
  Le type de données de la variable **newClients** est automatiquement déduit du type de l'objet créé par l'assignation (c'est-à-dire par ce qui se trouve à droite du signe **=**).
- La partie à droite de l'opérateur **=** créé une nouvelle instance d'un objet de type **Clients** via l'utilisation du mot clé **New**. Cette instance constitue une copie exacte et distincte de la classe **Clients** en mémoire (c'est votre 2CV au garage). C'est un *objet*.

Notez que le type **Clients** n'est pas reconnu par Visual Studio qui affiche un trait surligné bleu sur le mot complet ainsi qu'un trait surligné rouge sur la dernière lettre s :

```
Private Sub NouveauToolStripMenuItem_Click(ByV

    Dim newClients = New Clients()

End Sub
```

L'éditeur vous indique qu'il a bien trouvé une classe du même nom dans l'ensemble des assemblages référencés du projet mais qu'il n'est pas sûr que ce soit le bon car le nom n'est pas complet.

- Positionnez le curseur de la souris juste au dessus du caractère de soulignement rouge pour faire apparaître le menu de la balise active dans laquelle Visual Studio vous propose des solutions au problème rencontré :



Pour faire apparaître rapidement le menu de la balise active, vous pouvez aussi vous positionner directement sur le mot souligné et appuyez en même temps sur les touches **Contrôle** (Ctrl) et le **point**.

- Sélectionnez le premier choix **Importer 'Coach.Types'** pour autoriser l'usage du nom simple tel que nous l'avions marqué initialement.
- Les surlignés disparaissent et vous devez retrouver une nouvelle directive **Imports** en en-tête du fichier **Main.vb** :

```
Imports System.IO
Imports Coach.Types

Public Class Main
    Public Sub New()

        ' Cet appel est requis par le Concepteur Windows Form.
        InitializeComponent()
```

Il ne vous reste plus qu'à réaliser la liaison de données (databinding) avec les contrôles d'affichage de la grille. Heureusement pour nous, la propriété **DataSource** des contrôles **BindingSource** ou **DataGridView** peut avoir pour valeur plusieurs sources de données dont des objets, des types et listes de types.
Nous allons donc pouvoir directement remplacer le type **DataTable** par notre type d'objet personnalisé **Clients** sans perturber nullement le fonctionnement du binding.

Pour consulter une liste de quelques sources de données courantes :
http://msdn.microsoft.com/fr-fr/library/system.windows.forms.bindingsource.datasource.aspx

- Ajoutez la liaison de données à l'objet **newClients** comme suit :

Code VB

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                   Handles NouveauToolStripMenuItem.Click
    Dim newClients = New Clients()
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newClients
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.BindingSource = MainBindingSource
End Sub
```

Ce n'est pas la première (ni la dernière) fois que nous utilisons ce bloc de code réalisant la liaison d'une source de données et des contrôles d'affichage. Une bonne pratique est de l'isoler dans une procédure dédiée à la liaison de données séparée.

4. Isolez le code de databinding dans une procédure séparée :
   - Ajoutez une nouvelle procédure nommée **Bind** à la classe **Main.** Définissez un paramètre nommé **Source** de type **Object** pour transmettre à la procédure la source de données à lier :

Code VB

```
Private Sub Bind(ByVal Source As Object)
End Sub
```

   - Ajoutez le code de liaison des contrôles d'affichage :

Code VB

```
Private Sub Bind(ByVal Source As Object)
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = Source
    'Liaison du gestionnaire avec les contrôles d'affichage de données
    MainDataGridView.DataSource = MainBindingSource
    MainBindingNavigator.BindingSource = MainBindingSource
End Sub
```

   - Modifiez le gestionnaire d'évènement **NouveauToolStripMenuItem _Click** pour qu'il utilise la procédure **Bind** :

Code VB

```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                   Handles NouveauToolStripMenuItem.Click
    Dim newClients = New Clients()
    Bind(newClients)
    'Configuration du gestionnaire de liaison sur la source de données
    MainBindingSource.DataSource = newClients
```

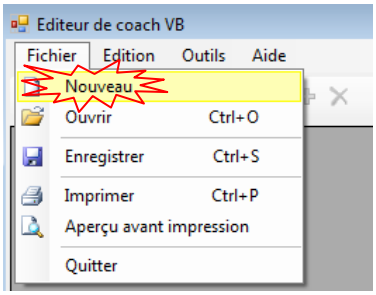
```
        'Liaison du gestionnaire avec les contrôles d'affichage de données
        MainDataGridView.DataSource = MainBindingSource
        MainBindingNavigator.BindingSource = MainBindingSource
End Sub
```

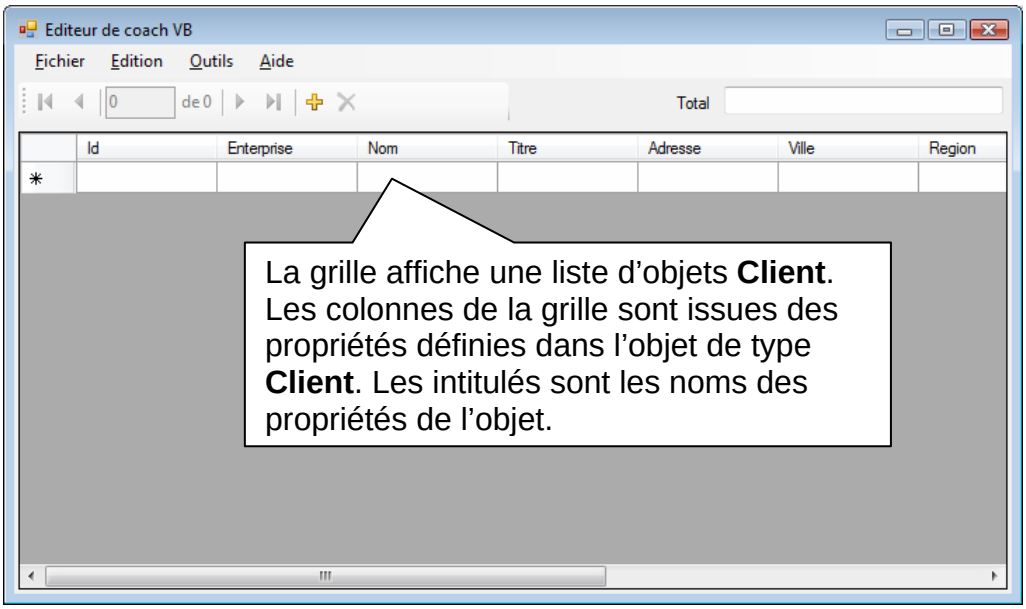
Visual Basic autorise l'utilisation du mot **New** partout où un objet est attendu. Vous pouvez donc réduire le code à l'écriture suivante :

Code VB

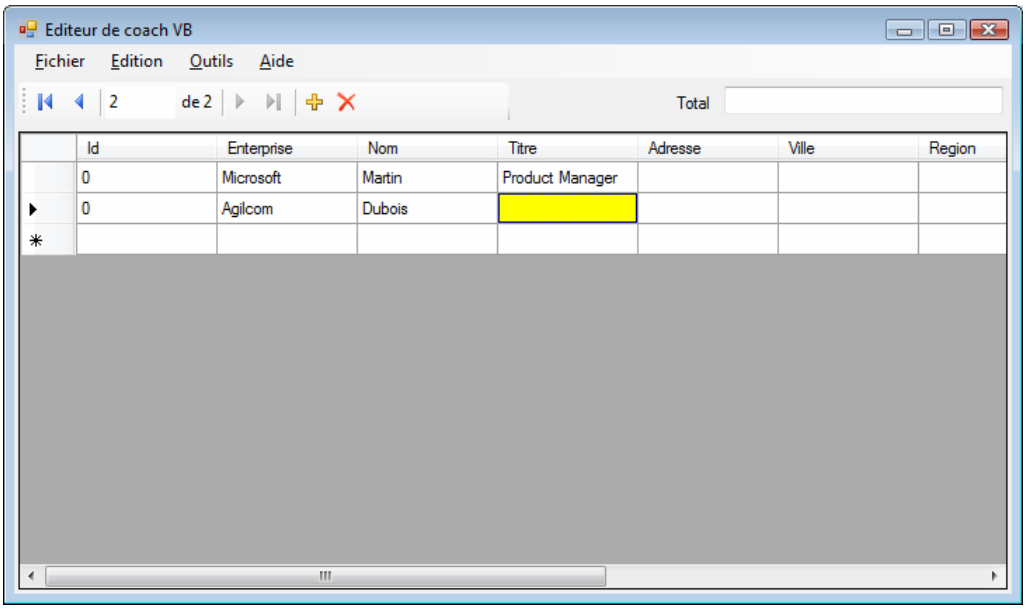
```
Private Sub NouveauToolStripMenuItem_Click(ByVal sender As System.Object, _
                                           ByVal e As System.EventArgs) _
                                   Handles NouveauToolStripMenuItem.Click
    Dim newClients = New Clients()
    Bind(New Clients())
End Sub
```

5. Testez la nouvelle fonctionnalité de création d'une liste de client vide :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Nouveau**. Vous devez obtenir une liste de clients vide :



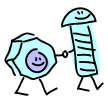


- Vous pouvez saisir de nouveaux clients en utilisant le bouton :



La grille fonctionne donc parfaitement. Vous constatez que la colonne **Id** s'initialise automatiquement à la valeur **0**. C'est normal puisqu'il s'agit d'une propriété de type **Integer** de notre objet que nous avons explicitement initialisée à 0.

```
Public Class Client

    Private _id As Integer = 0
    Private _entreprise As String = String.Empty
    Private _nom As String = String.Empty
    Private _titre As String = String.Empty
    Private _adresse As String = String.Empty
    Private _ville As String = String.Empty
    Private _region As String = String.Empty
    Private _codepostal As String = String.Empty
    Private _pays As String = String.Empty
    Private _telephone As String = String.Empty
    Private _telecopie As String = String.Empty

    Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)
            _id = value
        End Set
    End Property
```



L'idéal serait d'incrémenter automatiquement l'**Id** pour chaque nouveau client. Mais pour cela, il faudrait disposer d'un compteur défini en tant que variable globale commune à toutes les instances d'objets de type **Client** qui s'incrémenterait de 1 à chaque nouvel objet **Client**.
Ce type de membre de classe s'appelle un membre *partagé* en Visual Basic (dans certains langages, on parle de membre *statique*).

Qu'est qu'un membre **partagé** ?
Comme son nom l'indique ☺, il s'agit d'un membre *partagé* par toutes les instances d'une classe, c'est-à-dire par tous les objets du type donné.

Comme il n'est pas lié à une instance de classe particulière, il n'est donc pas nécessaire de créer une instance de la classe pour y accéder. On y accède en utilisant le nom de la classe (et non de l'objet) suivi d'un point puis du nom du membre partagé :

<nom de la classe>.<nom du membre partagé>

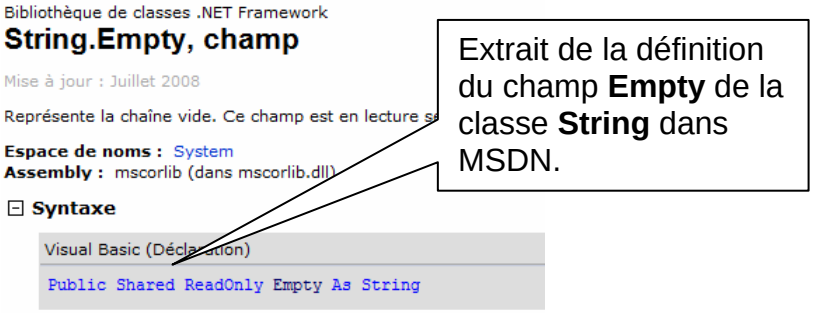
Ce peut être des propriétés mais aussi des méthodes de la classe.

Notez que vous avez déjà rencontré à plusieurs reprises ce type de membre qui est présent régulièrement dans les classes du Framework .NET.
Par exemple, la méthode **Concat** de la classe **String** que vous avez utilisée à l'atelier 4, est une méthode *partagée* qui ne nécessite pas la création d'une instance de la classe **String** pour être utilisée :

```
Me.Text = String.Concat("Editeur de coach VB", _
                        " - ",
                        fileInformation.Name)
```

Ou encore la propriété **Empty** de cette même classe **String** que vous avez utilisée au point 2 du § 2.2 de cet atelier, pour initialiser les membres privés de type **String** de la classe **Client**.

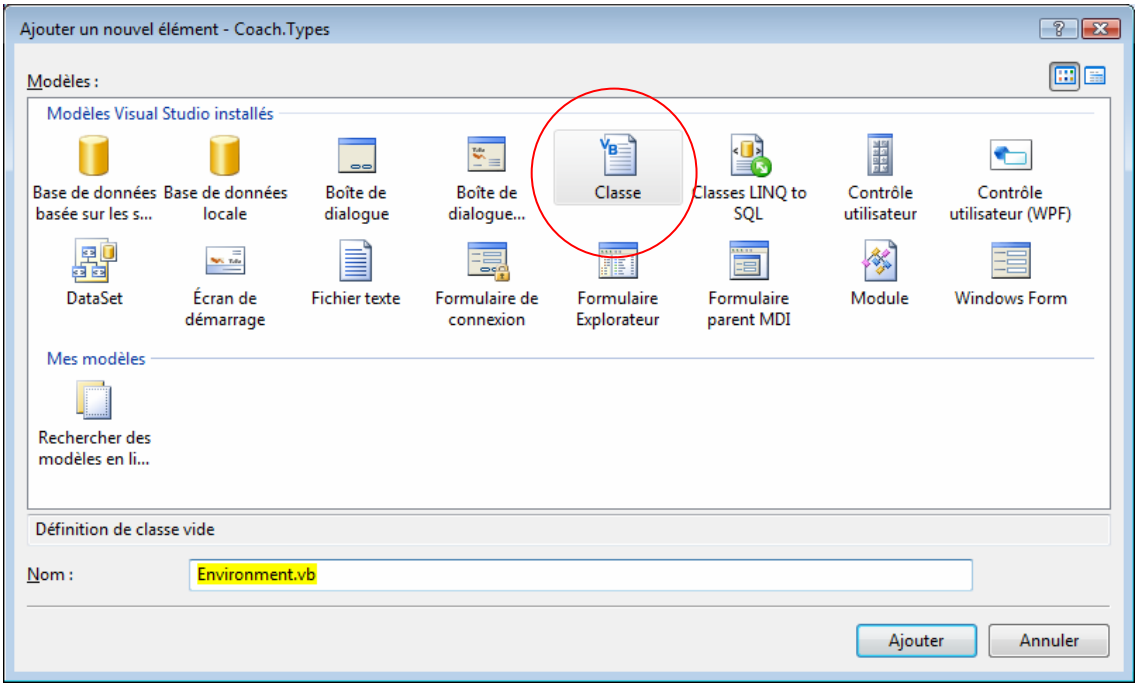
Pour déclarer un membre partagé d'une classe, utiliser le mot clé **Shared** comme ceci :



Pour tout savoir sur les membres partagés en Visual Basic :
http://msdn.microsoft.com/fr-fr/library/4hbsxy95.aspx

Il suffirait donc d'ajouter un membre partagé à la classe **Client** pour matérialiser le compteur d'incrément dont nous avons besoin.
Une bonne pratique consiste à isoler les membres partagés dont nous avons besoin dans l'application dans une classe séparée. Plutôt que de définir le compteur au sein de la classe **Client**, nous allons donc créer une nouvelle classe dans l'assemblage **Coach.Types**, que nous allons appelée par exemple **Environment**.
Comme cette classe aura pour vocation de contenir des membres partagés à usage interne de nos objets personnalisés, nous allons marquer la classe avec le modificateur d'accès **Friend** pour la limiter à l'usage interne de l'assemblage **Coach.Types** (reportez vous au § 2.2 de cet atelier pour retrouver la définition d'un modificateur d'accès).

6. Créez une classe Environment dans le projet Coach.Types contenant une propriété nommé IdCounter :
   - Dans l'**Explorateur de solutions**, faites un **clic-droit** sur la solution **Coach.Types** et sélectionnez le menu **Ajouter > Nouvel élément**…
   - Dans la fenêtre **Ajouter un élément**, sélectionner le modèle **Classe**.
   - Nommez cette classe **Environment**.



   - Supprimez le mot clé **Public** dans la définition de la classe. Indiquez une accessibilité de classe interne (en utilisant le modificateur **Friend**).

Code VB

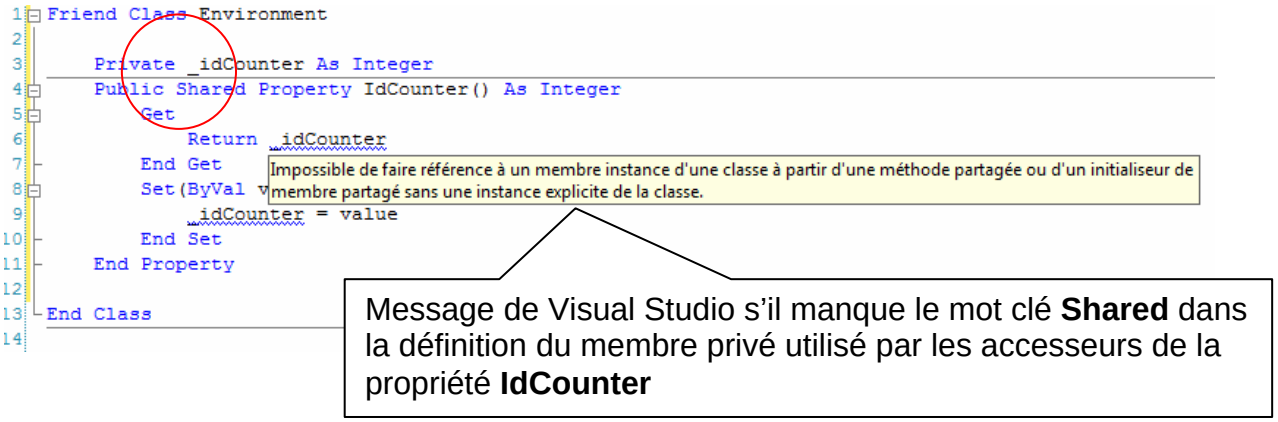
```
~~Public~~ Friend Class Environment

End Class
```

   - Ajouter une propriété **IdCounter** de type **Integer** en utilisant l'extrait de code **Property** vu au point 2 du § 2.2 de cet atelier :

```
1  Friend Class Environment
2
3      Private _idCounter As Integer
4      Public Property IdCounter() As Integer
5          Get
6              Return _idCounter
7          End Get
8          Set(ByVal value As Integer)
9              _idCounter = value
10         End Set
11     End Property
12
13 End Class
14
```

   - Marquez cette propriété comme membre partagé à l'aide du mot clé **Shared** (sans oublier le membre privé de soutien de la propriété) :

N'oubliez pas d'ajouter le mot clé **Shared** également sur le membre privé **_idCounter**. En effet, il ne serait pas cohérent de laisser les méthodes **Get** et **Set** utiliser un membre qui serait dépendant d'une instance de la classe. Quoiqu'il en soit vous pouvez compter sur Visual Studio pour vous avertir de l'incohérence :

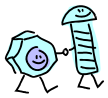


Au final vous devez obtenir le code suivant :

Code VB

```
Friend Class Environment
    Private Shared _idCounter As Integer = 0
    Public Shared Property IdCounter() As Integer
        Get
            Return _idCounter
        End Get
        Set(ByVal value As Integer)
            _idCounter = value
        End Set
    End Property
End Class
```

Pensez à initialiser le compteur.



Maintenant il s'agit d'incrémenter ce compteur à chaque nouvelle instance d'objet **Client** créée dans l'application.

Quel est le membre d'une classe qui correspond justement à la création d'une instance d'objet ?
Le **constructeur** de la classe bien sûr ! Nous avons déjà rencontré cette notion à plusieurs reprises au cours de ce tutorial. Dans notre cas, il nous faut donc définir un constructeur dans la classe personnalisée **Client** qui réalise l'incrément. Cette méthode nommée **New()** sera appelée au moment de la *construction* de l'objet.

7. Ajoutez un constructeur à la classe Client qui réalise l'incrément du compteur donné par la classe Environment :
   - Ouvrez le fichier **Client.vb** du projet **Coach.Types**.
   - Ajoutez un constructeur sans paramètre à la classe **Client** comme suit :

Code VB

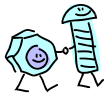
```
Public Class Client

   Public Sub New()

   End Sub
…
End Class
```

- Initialiser le membre privé _**id** de la classe avec la valeur en cours du compteur :

Code VB

```
Public Class Client

   Public Sub New()
      _id = Environment.IdCounter
   End Sub
…
End Class
```

Le membre partagé **IdCounter** est directement accessible en le précédant simplement du nom de la classe le contenant.



Pour incrémenter le compteur, il suffit maintenant d'ajouter une ligne directement à la suite du constructeur qui ajoute 1 à la propriété **IdCounter**.
Une autre solution plus élégante consisterait à inclure l'auto incrément au sein de la propriété **IdCounter** elle-même, dans la classe **Environment**. En effet, il s'agit d'un fonctionnement inhérent au compteur, complètement indépendant du code qui l'utilise…

Mais où placez le code d'incrément au sein de la définition de la propriété **IdCounter** ?
Pourquoi pas dans l'accesseur **Get** de la propriété **IdCounter** qui s'exécute justement à chaque fois que le compteur est utilisé ☺.

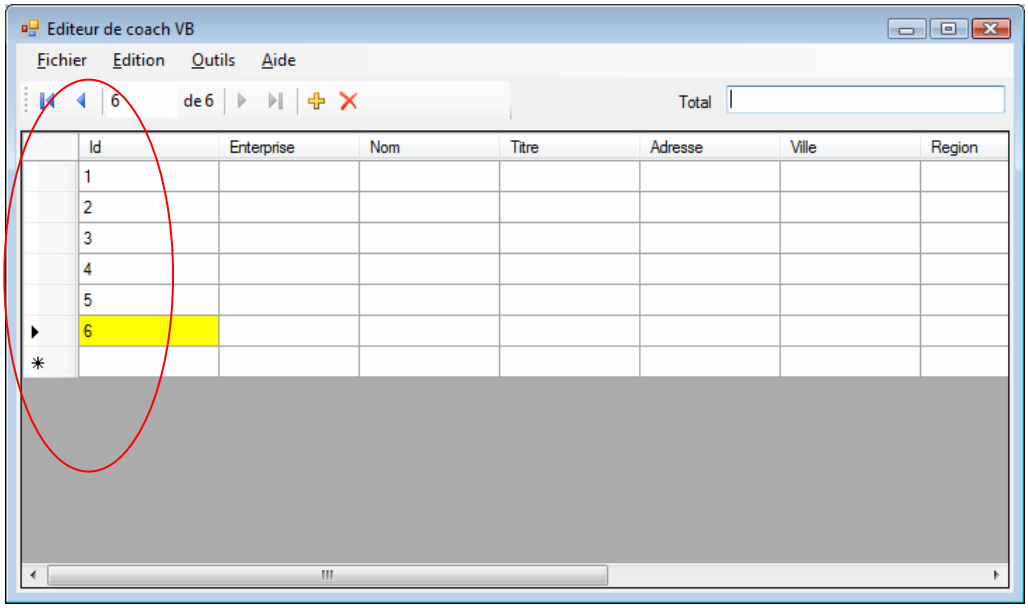
- Basculez sur la définition de la propriété **IdCounter** de la classe **Environment**.
- Modifiez l'accesseur **Get** pour incrémenter automatiquement le compteur avant de retourner une valeur :

Code VB

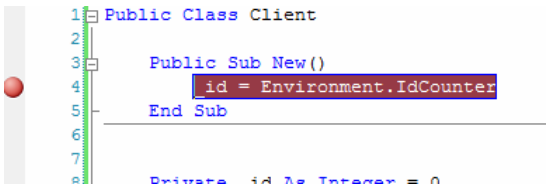
```
friend Class Environment

   Private Shared _idCounter As Integer = 0
   Public Shared Property IdCounter() As Integer
      Get
         _idCounter = _idCounter + 1
         Return _idCounter
      End Get
      Set(ByVal value As Integer)
         _idCounter = value
```

```
    End Set
  End Property

End Class
```

8. Testez l'auto incrément du compteur d'Id :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Nouveau**. Vous devez obtenir une liste de clients vide.
   - Saisissez de nouveaux clients en utilisant le bouton . La colonne **Id** doit s'incrémenter automatiquement de 1 à chaque nouvelle ligne :



   - Fermez l'application en cliquant .
   - Positionnez un point d'arrêt (touche F9) dans le constructeur de la classe **Client** comme suit :

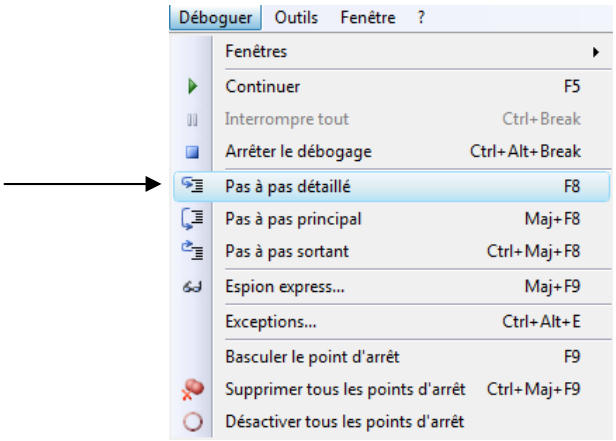


   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Nouveau**.
   - Saisissez un nouveau client en utilisant le bouton .
   - Visual Studio stoppe l'application au niveau du point d'arrêt. Vous êtes sur le point de lire la valeur du compteur **IdCounter** :

```
Public Class Client

    Public Sub New()
        _id = Environment.IdCounter
    End Sub
```

C'est maintenant que vous allez voir tout l'intérêt des **accesseurs** que vous avez définis sur la propriété **IdCounter** de la classe **Environment**. Comme vous tentez de *lire* la valeur de la propriété **IdCounter**, le runtime va appeler l'accesseur **Get** de la propriété et exécuter son contenu.

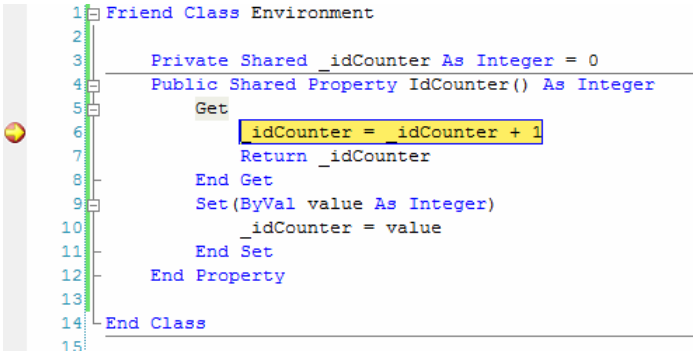
- Appuyez la touche **F8** ou sélectionnez le menu **Déboguer** > **Pas à pas détaillé** dans la barre de menu de Visual Studio pour avancer l'exécution pas à pas de manière la plus détaillée possible.



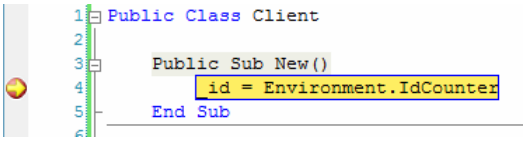
- Le pointeur d'exécution bascule aussitôt sur l'accesseur **Get** de la propriété **IdCounter** :

```
Friend Class Environment

    Private Shared _idCounter As Integer = 0
    Public Shared Property IdCounter() As Integer
        Get
            _idCounter = _idCounter + 1
            Return _idCounter
        End Get
        Set(ByVal value As Integer)
            _idCounter = value
        End Set
    End Property

End Class
```

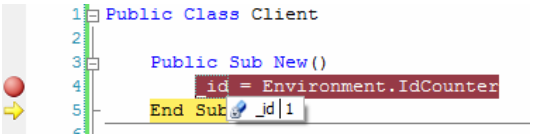
- Appuyez à nouveau sur la touche **F8**. Le pointeur se décale sur la ligne qui va incrémenter le compteur de 1.



- Appuyez trois fois sur la touche **F8** pour revenir sur la ligne de lecture de la propriété.



- Appuyez encore une fois sur la touche **F8** pour terminer l'exécution de la ligne.
- Passez avec la souris sur la variable **_id** pour consulter sa valeur. Vérifiez qu'elle a la valeur 1 (=0+1):



- Terminez l'exécution en cliquant **F5**.
- Terminez l'application.
- Supprimez le point d'arrêt en appuyant la touche **F9** sur la ligne correspondante.

Bravo ! Vous savez maintenant à quoi peuvent servir les accesseurs des propriétés membres des classes.

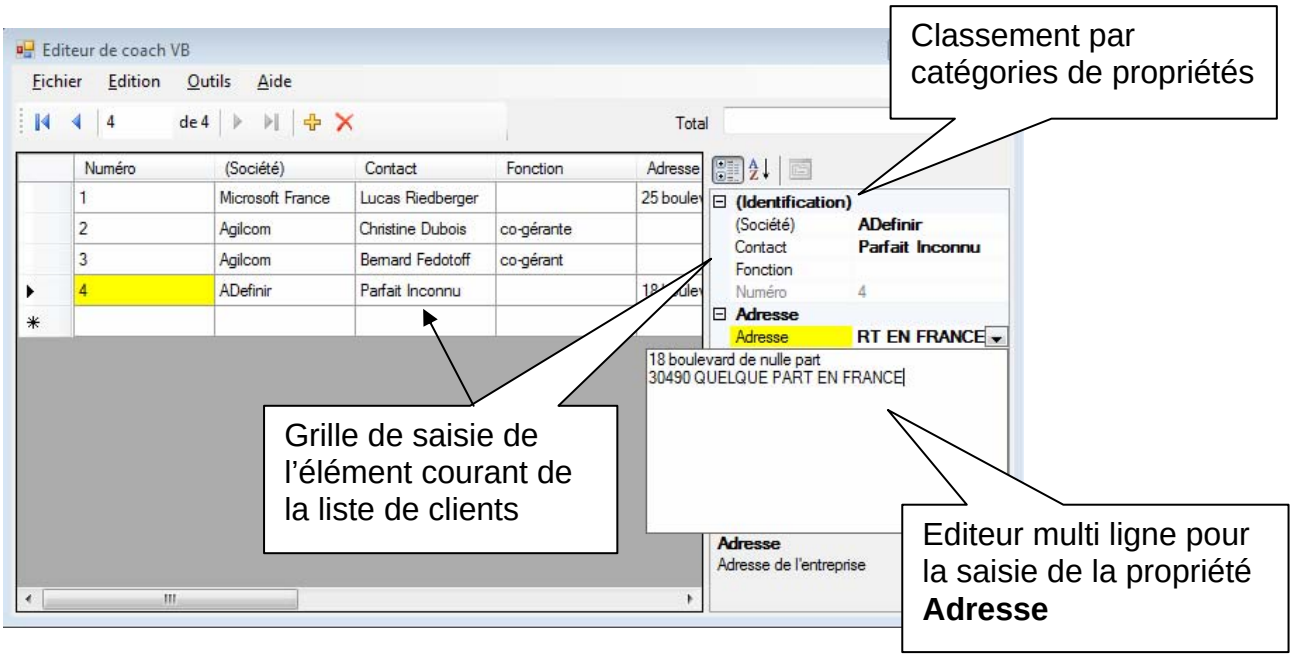
3 Utiliser des attributs

Dans cet exercice, vous allez apprendre à :

- Utiliser le contrôle PropertyGrid,
- Utiliser des attributs pour manipuler des objets personnalisés,
- Utiliser des éditeurs de propriétés personnalisés.

Objectif
L'objectif de cet exercice est d'apprendre à manipuler un objet personnalisé en le décorant à l'aide d'attributs.

Contexte fonctionnel
L'objectif fonctionnel de cet exercice est de rajouter à droite de l'éditeur une grille de propriétés pour aider l'utilisateur lors de la saisie des informations d'un client. Cette fenêtre propose une saisie plus conviviale que celle disponible dans la grille de données, que nous configurerons donc à partir de maintenant en lecture seule.

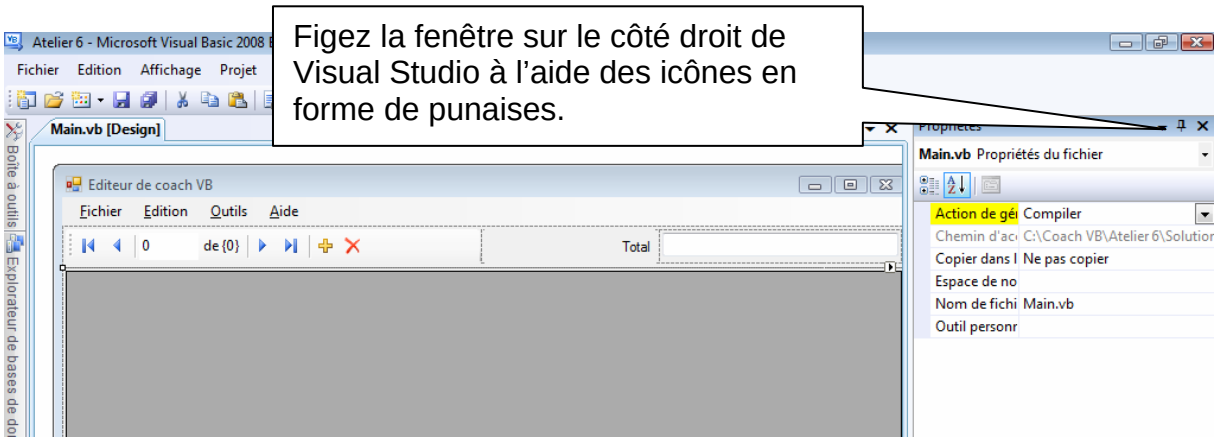


Pour dessiner une grille de propriétés à la manière de la fenêtre de propriétés dont nous disposons en tant que développeur dans Visual Studio, nous allons tout simplement nous appuyer sur un composant existant du Framework .NET appelé **PropertyGrid**. Il s'agit d'un contrôle d'affichage puissant fournissant une interface à l'utilisateur pour explorer (afficher et/ou écrire) les propriétés d'un objet.

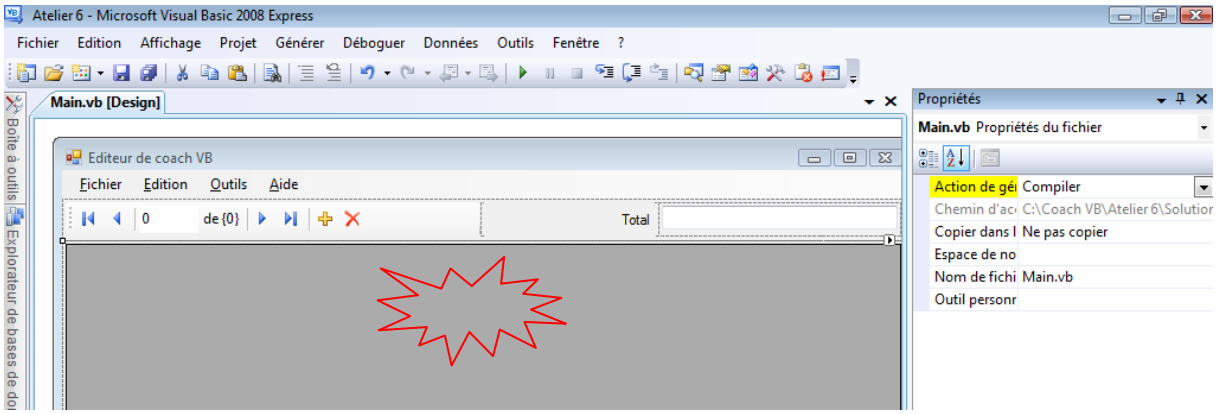
Déroulement de l'exercice :

1. Dans un premier temps, vous allez faire un peu de place dans le formulaire Main de l'éditeur pour y inclure le nouveau composant PropertyGrid :

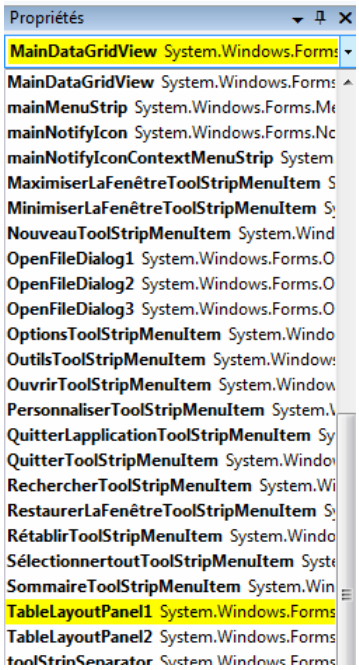
- Dans l'**Explorateur de solutions**, double cliquez sur le fichier **Main.vb** pour l'afficher en mode **Design**.
- Si elle n'est pas déjà visible, affichez la fenêtre de propriétés en cliquant **F4**.



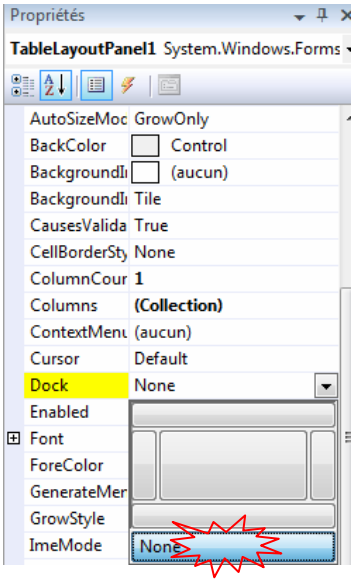
- Cliquez n'importe où sur le **Concepteur de formulaire** pour amener le focus sur le formulaire.



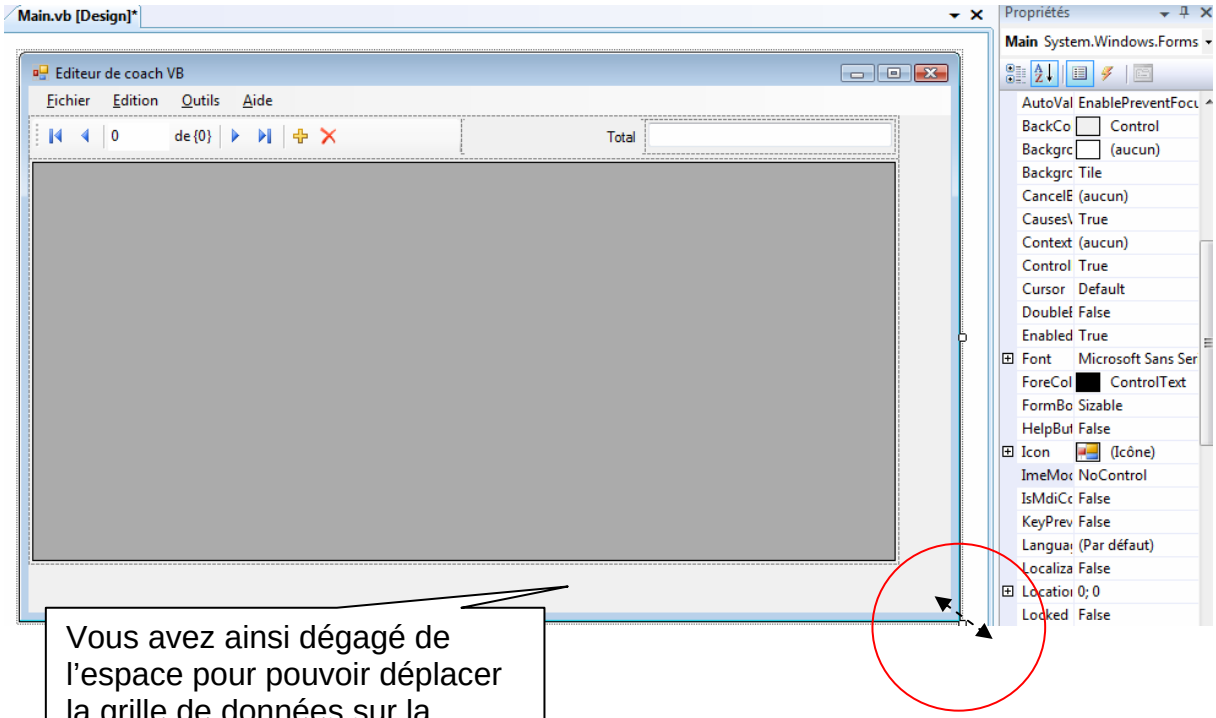
- Dans la liste déroulante située tout en haut de la fenêtre de propriétés, retrouvez puis sélectionnez le contrôle nommé **TableLayoutPanel1** qui représente le panneau principal du formulaire **Main** :



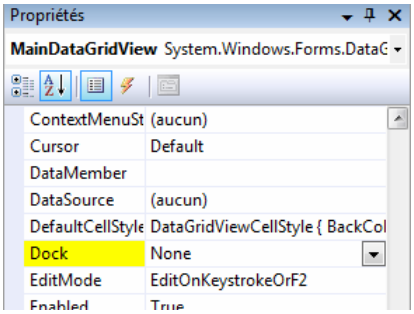
- Configurez sa propriété **Dock** à la valeur **None** pour désolidariser le contrôle de son conteneur (à savoir le formulaire) :



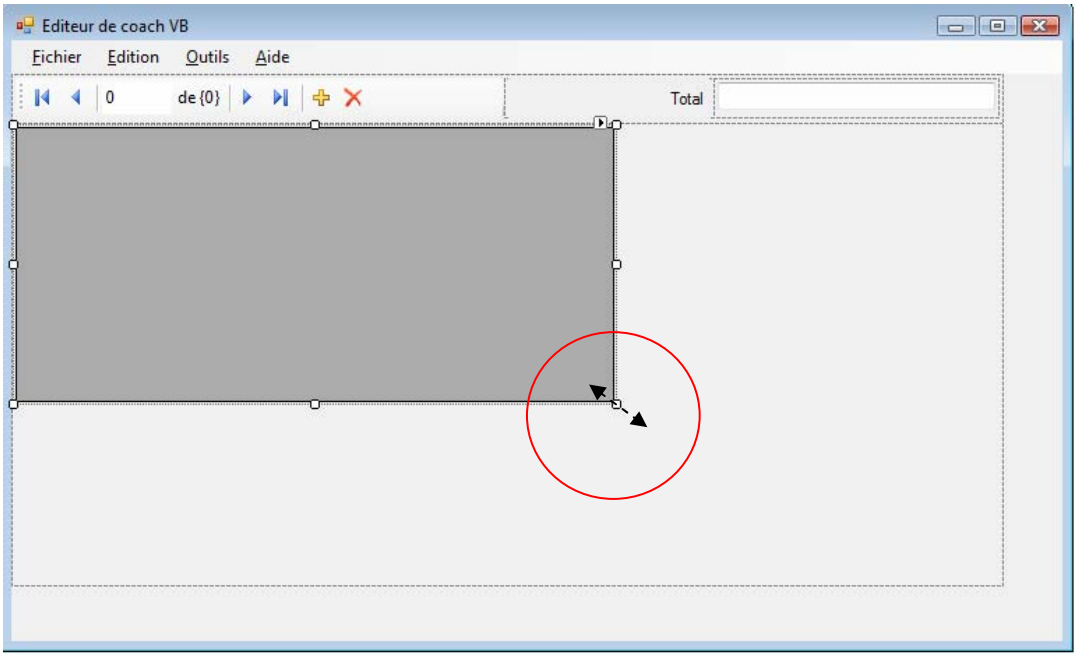
- Cliquez maintenant sur la barre de titre du formulaire **Main** pour le sélectionner.
- Agrandissez la taille du formulaire en positionnant la souris sur le coin bas droit du formulaire et en tirant doucement vers le bas :



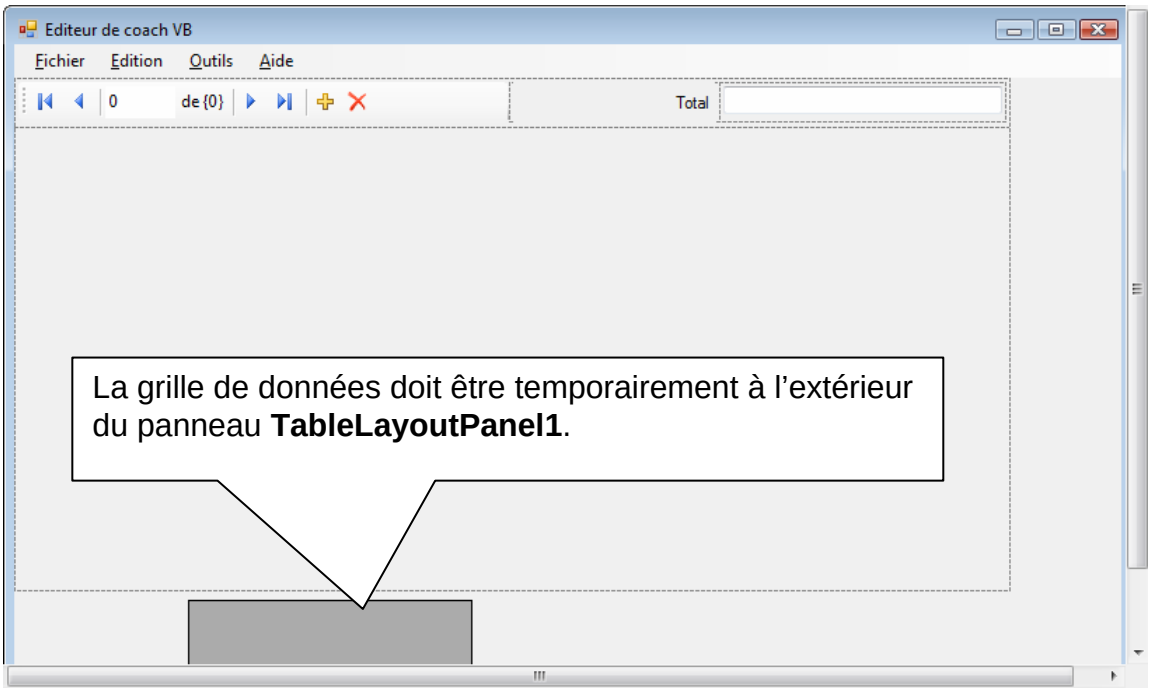
- Dans la liste déroulante de la fenêtre de propriétés, sélectionnez la grille de données **mainDataGridView**.
- Configurez la propriété **Dock** du contrôle **mainDataGridView** à la valeur **None** pour la désolidariser de son conteneur (à savoir le panneau **TableLayoutPanel1**)



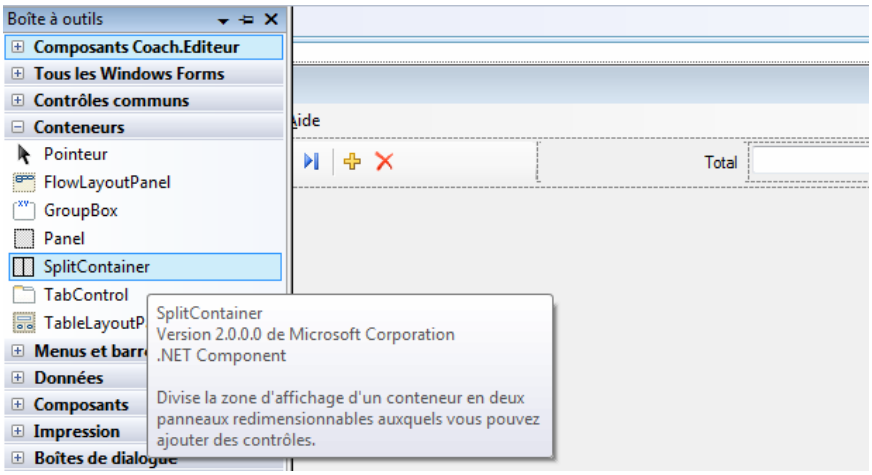
- Pour pouvoir déplacer plus facilement la grille, réduisez sa taille en faisant glisser le coin bas droit avec la souris :



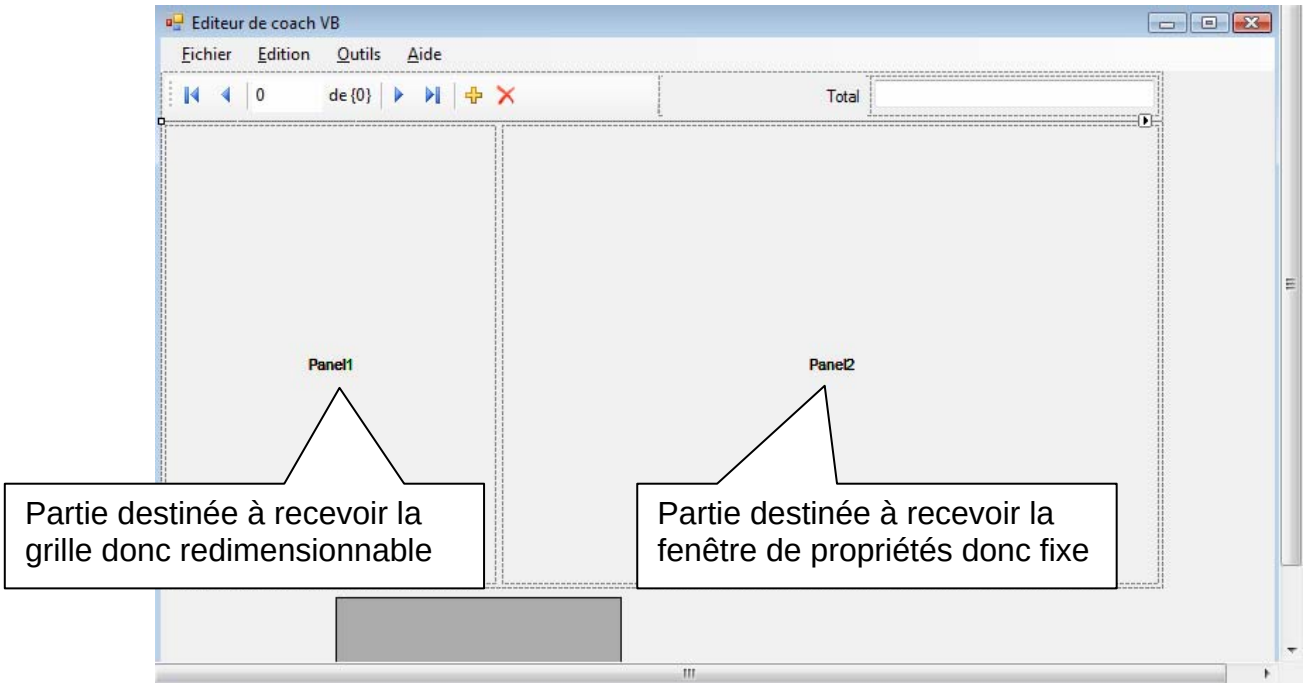
- Déplacez la grille en dehors du panneau en la faisant glisser déplacer sur l'espace laissé libre par l'agrandissement du formulaire **Main** :



- Depuis la **boite à outils**, faites un glisser déplacer de la rubrique **Conteneurs** > du contrôle **SplitContainer** sur l'emplacement initial de la grille de données.

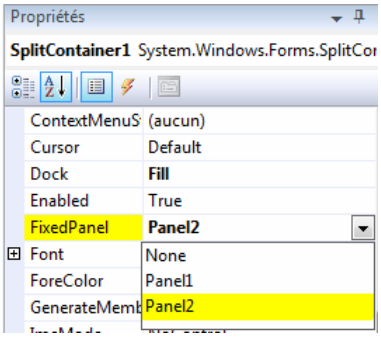


L'intérêt du contrôle **SplitContainer** est qu'il divise la zone conteneur en deux parties redimensionnables. En plus d'aider au positionnement des contrôles sur le formulaire, le **SplitContainer** est intéressant pour gérer le comportement des différentes zones lorsque l'utilisateur redimensionne la fenêtre. Par exemple, vous pouvez figer la taille d'une des zones en cas de redimensionnement.
Dans notre cas, cela va s'avérer très utile pour autoriser l'utilisateur à redimensionner la grille de données à sa guise tout en lui imposant une dimension fixe pour la partie qui comprendra la fenêtre de propriétés.

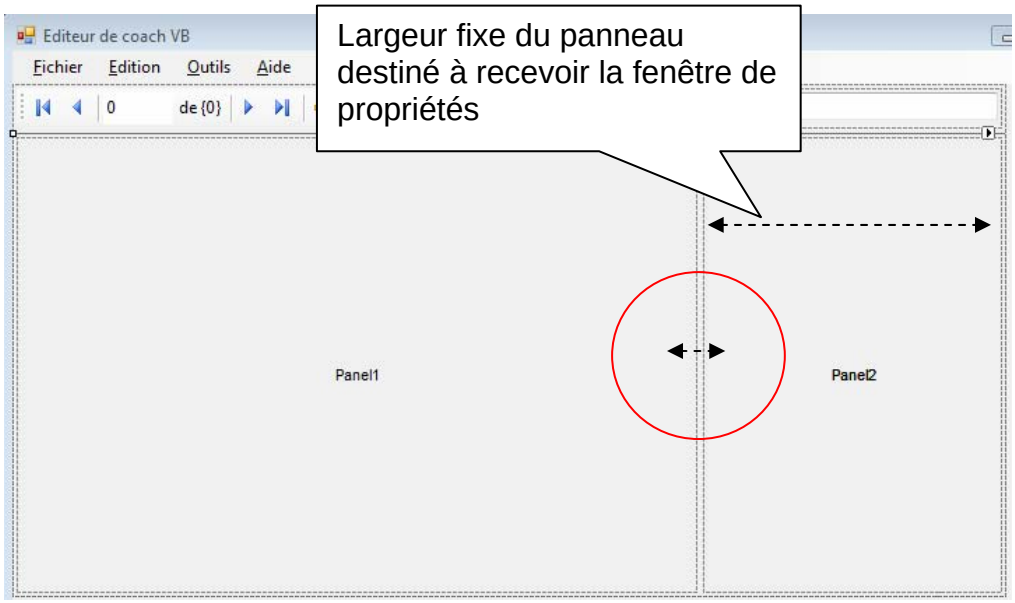


Pour tout savoir sur les **SplitContainer** :
http://msdn.microsoft.com/fr-fr/library/system.windows.forms.splitcontainer.aspx

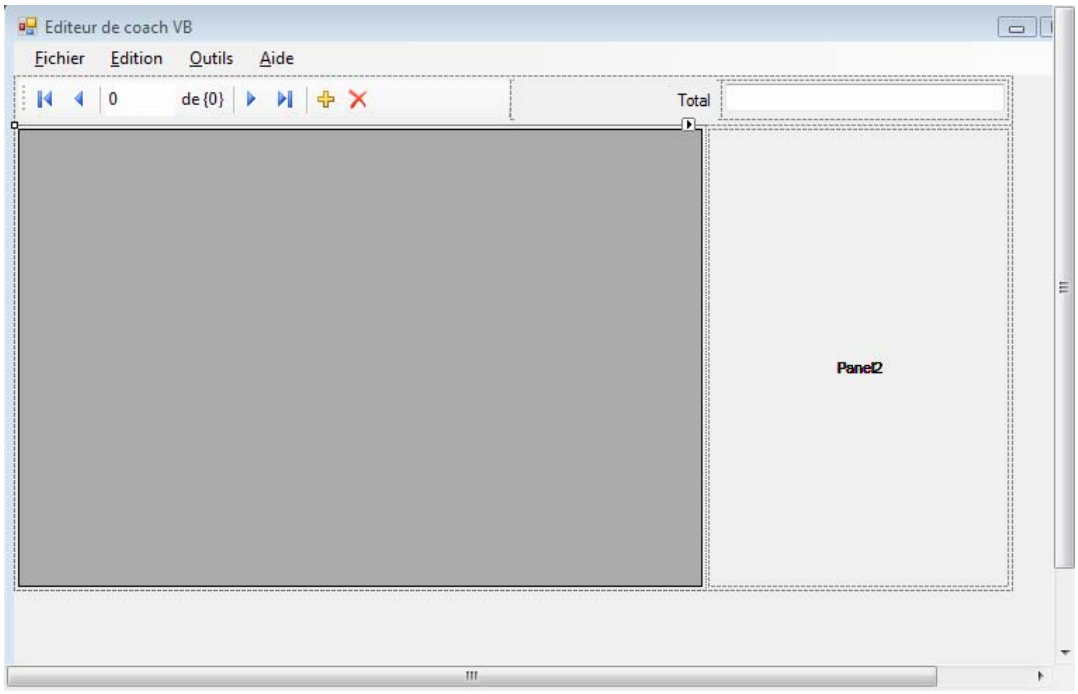
- Dans la fenêtre de propriétés, vérifiez que la propriété **Dock** du contrôle **SplitContainer1** est initialisée à la valeur **Fill** pour qu'il occupe tout l'espace disponible dans son conteneur.
- Configurez la propriété **FixedPanel** à la valeur **Panel2** pour indiquer que vous souhaitez figer les dimensions du deuxième panneau (donc celui de droite).



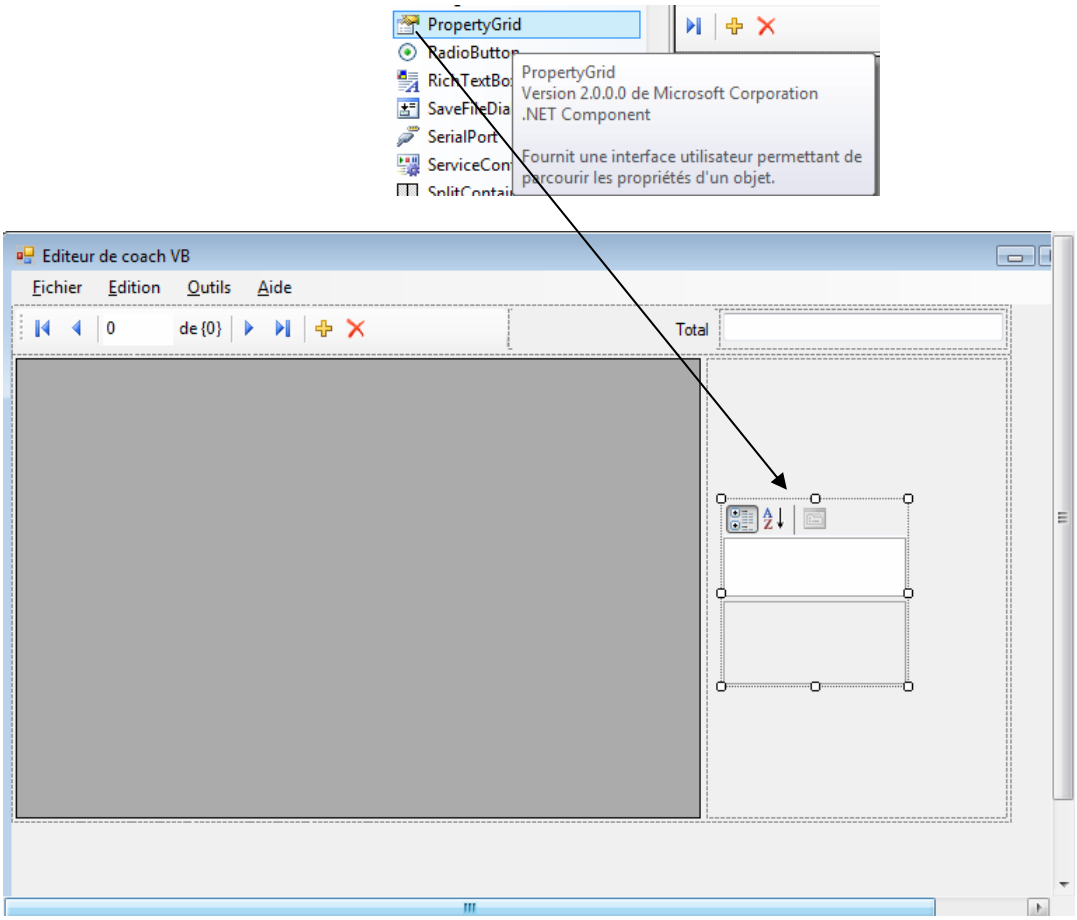
- A l'aide de la souris, faites un glisser déplacer de la barre centrale matérialisant le **SplitContainer** pour réduire la taille du deuxième panneau de façon à obtenir les dimensions usuelles d'une fenêtre de propriétés sur le panneau de droite :



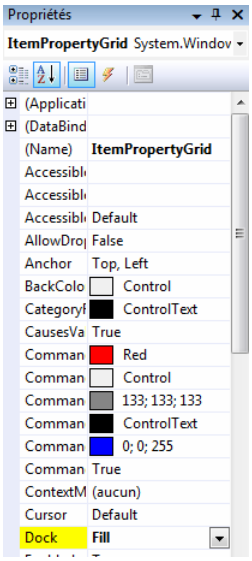
- Sélectionnez de nouveau la grille de données **mainDataGridView**.
- Faites un glisser-déplacer de la grille **mainDataGridView** dans le panneau **Panel1** de gauche du contrôle **SplitContainer1**.
- Reconfigurez la propriété **Dock** de la grille à la valeur **Fill** pour qu'elle reprenne les dimensions totales de son conteneur (à savoir le **Panel1** de **SplitContainer1**) :



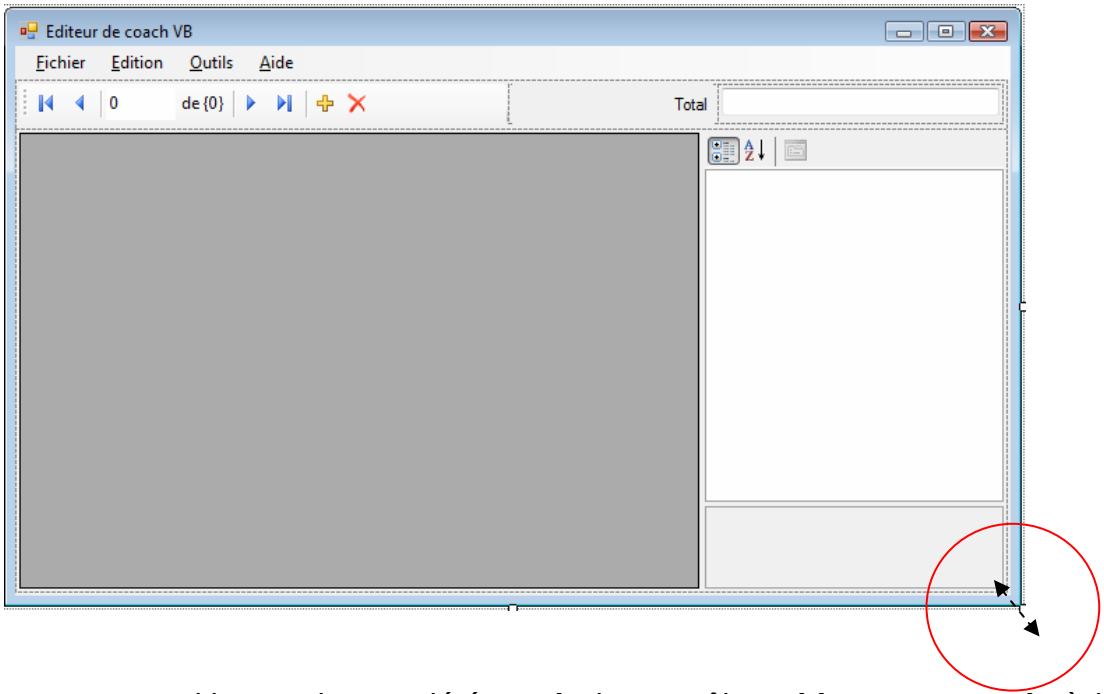
- Depuis la **boite à outils**, faites un glisser déplacer de la rubrique **Tous les Windows Forms** > du contrôle **PropertyGrid** sur le panneau de droite du contrôle **SplitContainer1**.



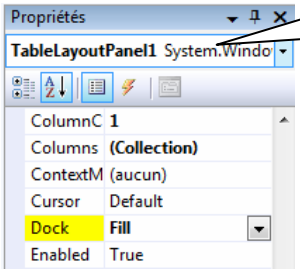
- Dans la **Fenêtre de Propriétés**, configurez la propriété **Name** à la valeur **ItemPropertyGrid**.
- Configurez la propriété **Dock** à la valeur **Fill** pour que le contrôle prenne tout l'espace disponible dans le **Panel2**.



- Redimensionnez maintenant le formulaire pour qu'il reprenne sa taille initiale, calquée sur celle du contrôle **TableLayoutPanel1**.



- Repositionnez la propriété **Dock** du contrôle **TableLayoutPanel1** à la valeur **Fill** pour le lier à son conteneur.



Retrouvez le contrôle en vous aidant de la liste déroulante dans la fenêtre de propriétés.

- Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.

Voilà ! Le contrôle de grille de propriétés est en place sur le formulaire ! Il ne reste plus qu'à le lier à notre objet personnalisé de type **Client** .

2. Configurez le contrôle PropertyGrid pour le lier à la grille de données MainDataGridView afin qu'il affiche les propriétés de l'enregistrement client courant :

Le fonctionnement du contrôle **PropertyGrid** repose sur la propriété **SelectedObject** qui doit référencer l'objet dont on veut afficher les propriétés.

msdn

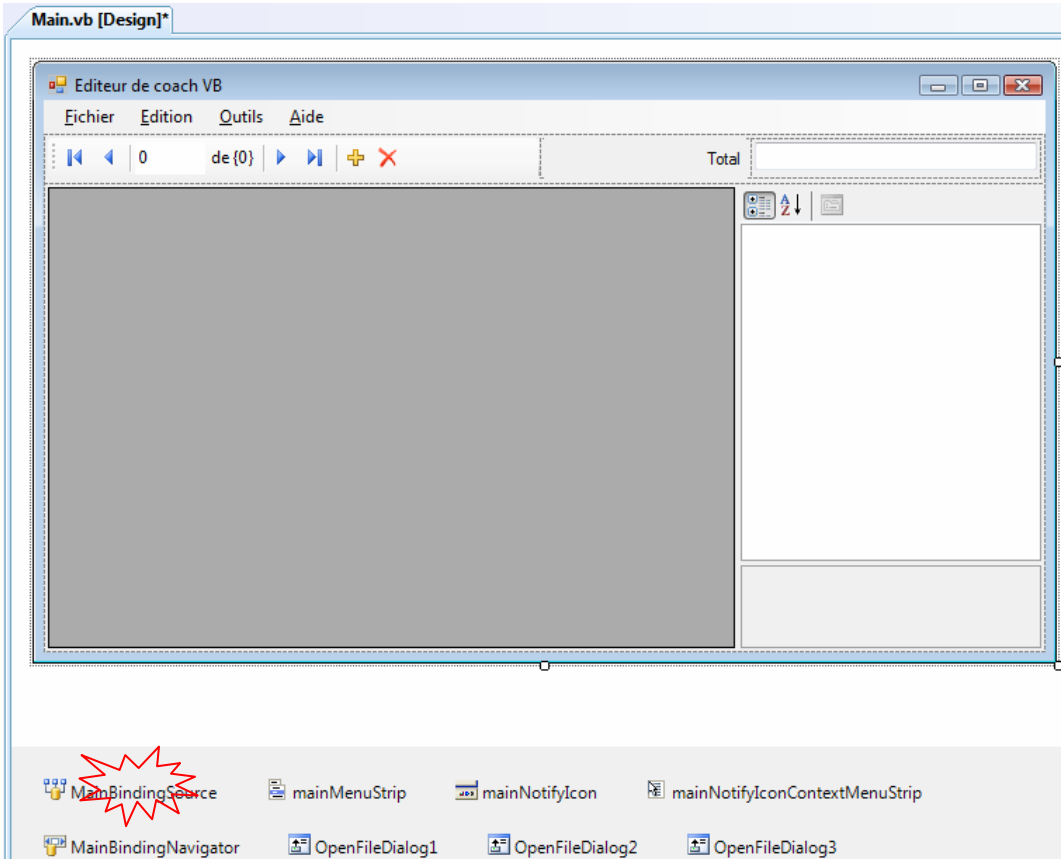
Pour en savoir davantage sur le contrôle **PropertyGrid** :
http://msdn2.microsoft.com/fr-fr/library/x495ht92.aspx
Voici un lien pour apprendre à personnaliser le contrôle **PropertyGrid** :
http://msdn2.microsoft.com/en-us/library/aa302326.aspx (lien en anglais)

Dans notre cas, l'objet en question est l'enregistrement **Client** courant dans la grille de données **MainDataGridView**, ce qui implique de mettre à jour la propriété **SelectedObject** du contrôle **ItemPropertyGrid** systématiquement *quand* la sélection change dans la grille de données.

Tiens, on utilise la conjonction *quand*… ça ne vous rappelle rien ?
Ca ressemble à un évènement (cf. point 4 du §2.1 de l'atelier 3) ! La conjonction *quand* est généralement suivie du verbe qui donne l'évènement déclencheur : **la sélection change** .

Sur quel objet va porter l'évènement ?
Ce pourrait être la grille de données… mais qui fournit les informations des clients à la grille ?
La source de données est gérée par le contrôle **MainBindingSource** donc c'est sur ce contrôle que nous allons chercher l'évènement qui va nous intéresser. Il suffira ensuite d'écrire un gestionnaire d'évènement dans lequel nous coderons l'affichage de l'élément courant dans le contrôle **ItemPropertyGrid** !

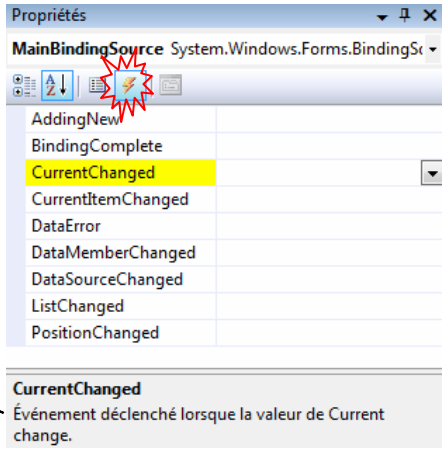
- Rebasculez sur le fichier **Main.vb** pour l'afficher en mode **Design**.
- Sélectionnez le contrôle **MainBindingSource** dans la zone de dépôt de contrôles du formulaire.



- Dans la fenêtre de propriétés du contrôle, cliquez sur l'icône de la barre d'outils pour lister tous les évènements disponibles sur ce contrôle.

Voyez-vous un évènement correspondant à notre besoin ?
CurrentChanged est justement l'évènement déclenché lorsque l'élément courant a changé !

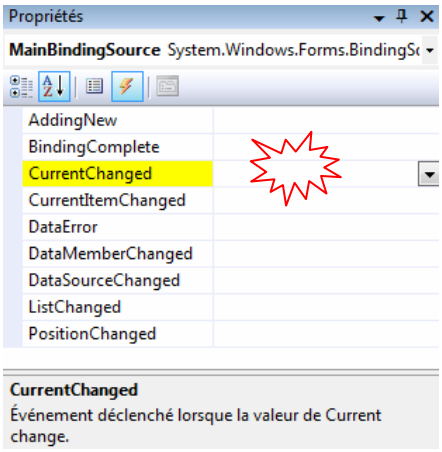
Aidez-vous de l'explication proposée au bas de la fenêtre de propriétés



Attention à ne pas confondre avec l'évènement **CurrentItemChanged** qui se déclenche lorsque n'importe laquelle des propriétés de l'élément

courant a changé.

- Double cliquez dans la zone de texte à droite de l'évènement **CurrentChanged** pour générer automatiquement la création d'un gestionnaire d'évènement associé.




```
Private Sub MainBindingSource_CurrentChanged(ByVal sender As System.Object, _
                                             ByVal e As System.EventArgs) _
                                             Handles MainBindingSource.CurrentChanged

End Sub
```

Il nous faut maintenant assigner à la propriété **SelectedObject** de type **Object** du contrôle **ItemPropertyGrid** l'objet **Client** en cours. Celui-ci est fourni par la propriété **Current** du contrôle **MainBindingSource**, elle aussi de type **Object**.

- Ajoutez le code de liaison du contrôle **ItemPropertyGrid** avec l'élément courant de la source de données :

Code VB

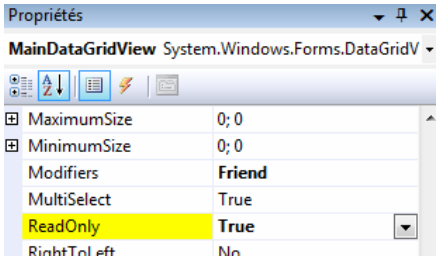
```
Private Sub MainBindingSource_CurrentChanged(…) _
                                Handles MainBindingSource.CurrentChanged
    ItemPropertyGrid.SelectedObject = MainBindingSource.Current
End Sub
```

Une autre écriture possible serait la suivante :
ItemPropertyGrid.SelectedObject = _
CType(sender,BindingSource).Current
Elle consiste à utiliser le paramètre **sender** de type **Object** du gestionnaire d'évènement, qui renvoie l'instance de l'objet sur lequel a été déclenché l'évènement. Pour rappel, le second paramètre **e**, de type **EventArgs** (ou d'un type dérivé), transporte des informations sur le contexte d'exécution de l'évènement.
Dans cette écriture il faut toutefois réaliser une conversion de type pour

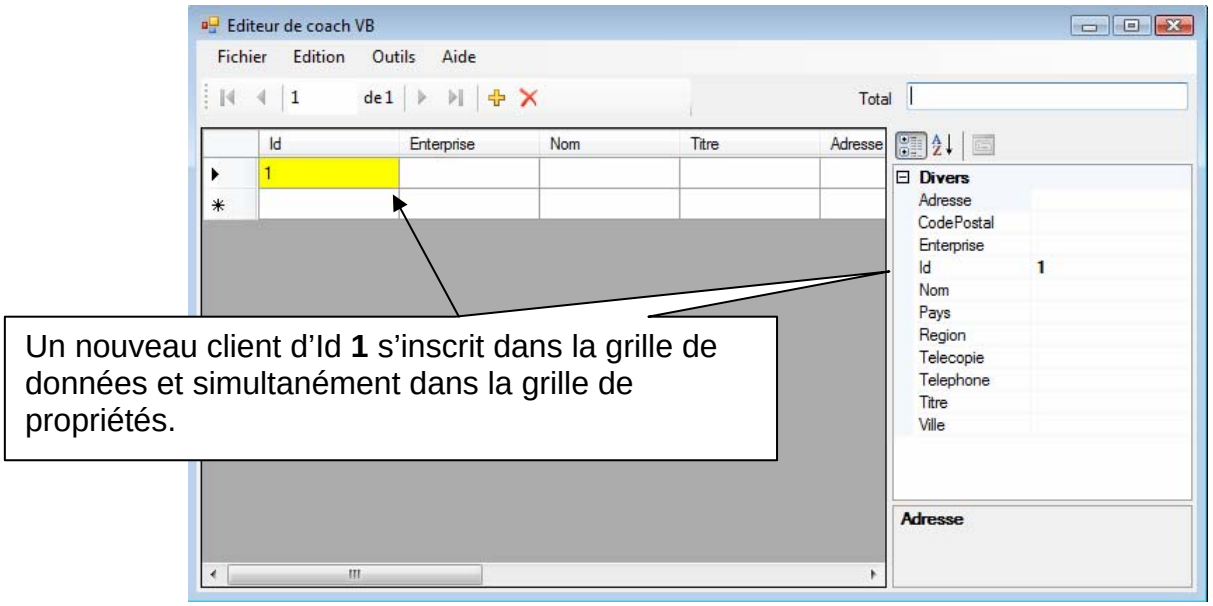
convertir l'élément en cours en un objet de type **BindingSource** avant de pouvoir prétendre à l'usage de la propriété **Current**.

Même si elle paraît plus compliquée, cette écriture garantie que vous pointez bien sur l'instance de l'objet sur lequel s'est déclenché l'évènement.
Pensez donc à utiliser ce paramètre **sender** dans vos gestionnaires d'évènement…

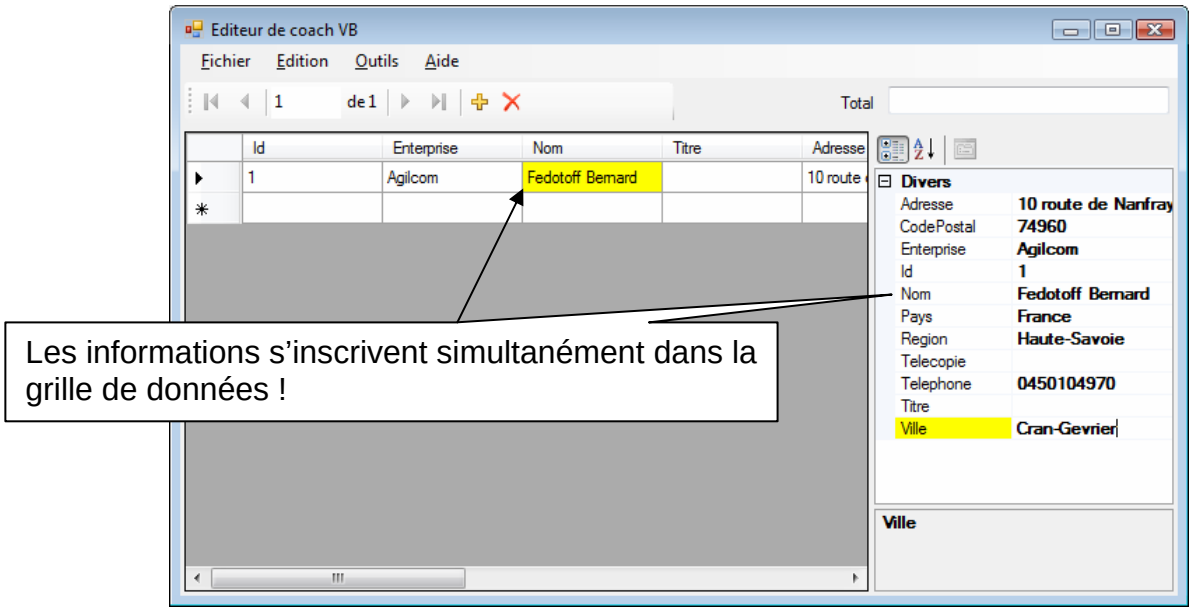
- Pour terminer, configurez la grille de données en lecture seule. Pour cela, rebasculez en mode **Design** du formulaire.
- Sélectionnez le contrôle **MainDataGridView** et affichez sa fenêtre de propriétés (F4).
- Configurez la propriété **ReadOnly** à la valeur **True**.



3. Testez le fonctionnement du contrôle **ItemPropertyGrid** :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Nouveau** pour générer une liste de clients vide.
   - Ajoutez un nouveau client en cliquant le bouton .

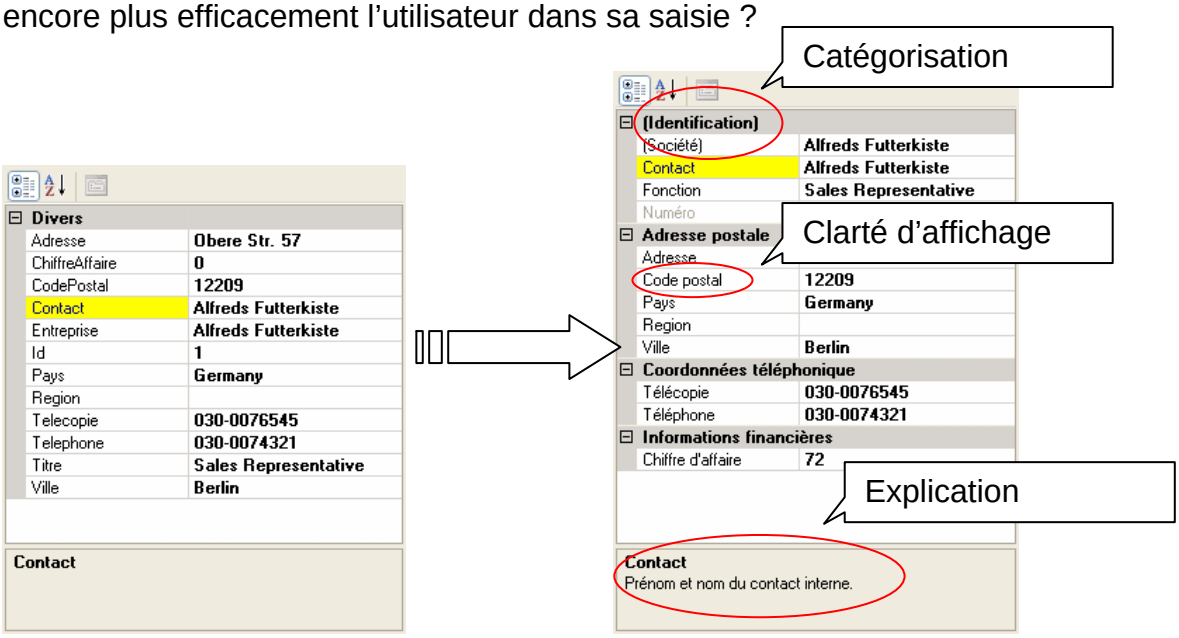


- Utilisez la grille de données pour saisir les informations du client :



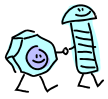
C'est pas mal, non ? On se croirait dans une fenêtre de propriétés de Visual Studio…enfin presque.

Comment faire pour personnaliser l'affichage de la grille de façon à aider encore plus efficacement l'utilisateur dans sa saisie ?



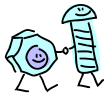
L'objectif est de :

- catégoriser les propriétés.
- afficher un commentaire d'explication en bas de la grille.
- clarifier les intitulés de propriété en ajoutant notamment des espaces et des accents.



Dans une démarche de code classique, ou « à l'ancienne », on chargerait le code de présentation (i.e. la classe Main) de travailler la façon dont l'information va être affichée à l'utilisateur. Mais dans une démarche de

programmation objet, notamment avec .Net, nous allons enrichir la description des objets de façon à ce que l'interface utilisateur n'ait qu'à afficher les propriétés de la manière dont le concepteur de l'objet l'a décidé. L'intérêt est que quelque soit le nombre d'endroit où l'objet est affiché, il le sera toujours de la même manière.



Comment enrichir la description d'un objet ?
Avec .NET, il est possible de spécifier des informations supplémentaires sur un objet qui permettent au runtime de mieux savoir comment utiliser cet objet. Ces informations « descriptives » sont aussi appelées **métadonnées** (**metedata** dans la littérature anglaise). Elles sont directement enregistrées dans les assemblages.
Pour spécifier ces données, on utilise des balises semblables à des mots clés nommées **attributs**.

C'est quoi un **attribut** ?
Un attribut est utilisé pour étendre la description d'un élément dans un assemblage. Ce peut être tout type d'information susceptible d'apporter une information au runtime. Il intervient un peu comme un adjectif pour annoter l'assemblage lui-même ou une classe, une méthode, une propriété, ou tout autre élément constitutif d'un assemblage.
Le Framework propose un ensemble d'attributs standards, qui pour la plupart ne sont pas spécifiques au langage de programmation, mais il est aussi possible de développer vos propres attributs afin de prendre en charge vos propres extensions dans la description des types.

msdn Pour tout savoir sur la programmation des attributs avec Visual Basic :
http://msdn.microsoft.com/fr-fr/library/39967861.aspx

Comment utiliser un **attribut** ?
Les attributs s'ajoutent au niveau de la déclaration de l'élément qu'ils décrivent. Sur le plan de la syntaxe, vous pouvez spécifier un attribut simplement en mettant son nom entre les signes inférieur et supérieur **<*attribut*>**, devant la déclaration de l'entité à laquelle il s'applique. Par exemple, une propriété avec l'attribut **ReadOnly** (limitant son utilisation en lecture uniquement dans une grille de propriétés) est déclarée de la manière suivante :

Espace de nommage qui fournit l'attribut

Nom de l'attribut

```
<System.ComponentModel.ReadOnly(True)>
Public Property Id() As Integer
    Get
        Return _id
    End Get
    Set(ByVal value As Integer)
        _id = value
    End Set
End Property
```

Attention à ne pas omettre le surligné _ en fin de ligne

Paramètre du (ou de l'un des) constructeur(s) de l'attribut

Quelques remarques :

- Notez que l'attribut fait partie de la ligne de déclaration d'où l'usage d'un surligné _ en fin de ligne pour indiquer qu'il s'agit bien d'un

élément de la ligne de déclaration.

- Un attribut standard du Framework .NET est nécessairement fourni par un assemblage et est donc nécessairement à l'intérieur d'un espace de nom. Vous pouvez donc utiliser soit son nom complet soit son nom court. Dans ce dernier cas, il est nécessaire d'ajouter une directive **Imports** sur l'espace de nommage de l'attribut en début de code. Dans les deux cas, une référence sur l'assemblage de l'attribut doit être ajoutée à la liste des références du projet.
- Un attribut n'est ni plus ni moins qu'un type de .NET défini par une classe. C'est pourquoi il se déclare en invoquant son constructeur, qui peut lui-même attendre des paramètres. Par rapport aux autres types que nous avons déjà utilisés dans ce coach, il existe une limitation importante : les paramètres utilisables dans le constructeur de l'attribut sont nécessairement des éléments constants (comme la constante **True** dans l'exemple ci-dessus). Cela s'explique par le fait qu'un attribut particulier fait partie des métadonnées de description du type, et qu'il n'est, par la même, pas lié au cycle de vie de *l'instance* de l'objet.

Quel rapport entre les attributs et notre grille de propriétés que nous cherchons à personnaliser ?
Il existe dans le Framework .NET des attributs standards avec lesquels vous allez décorer les propriétés de l'objet **Client**. Le runtime les interprète au moment de l'affichage de la grille de propriétés pour savoir de quelle manière afficher les données de l'objet.
Les attributs que nous allons ici utiliser sont dans l'espace de nom **System.ComponentModel**, et se trouvent dans l'assemblage **System.dll** :

- **ReadOnly** : cet attribut spécifie si la propriété à laquelle est lié cet attribut est en lecture seule ou en lecture/écriture lorsqu'elle est affichée dans une grille de propriétés.
- **DisplayName** : cet attribut spécifie le libellé à afficher pour la propriété.
- **Category** : cet attribut spécifie le nom de la catégorie dans laquelle grouper la propriété lorsque la grille de propriétés est en mode **Par catégorie** (bouton de la grille).
- **Description** : Cet attribut spécifie la description de la propriété qui s'affiche en bas de la grille de propriétés.

4. Décorez les propriétés de la classe **Client** avec les attributs adéquats :
   - Dans l'**Explorateur de solutions**, faites un double-clique sur le fichier **Clients.cs** du projet **Coach.Types** pour afficher le code de la classe **Client**.
   - Tout en haut du fichier, au dessus de la ligne de déclaration de la classe **Client**, ajoutez une directive **Imports** de référence à l'espace de nommage **System.ComponentModel**.

Code VB

```
Imports System.ComponentModel
```

```
Public Class Client
…
End Class
```

Notez qu'il n'est pas utile de référencer l'assemblage correspondant à cet espace de noms car il l'est déjà par défaut dans notre modèle de projet.

- Positionnez le curseur sur la déclaration de la propriété **Id**.
- Indiquez une description succincte pour la propriété à l'aide de l'attribut **Description**, par exemple : **Identification du client** :

Code VB

```
<Description("Identification du client")> _
Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)
            _id = value
        End Set
End Property
```

- Ajoutez à la suite du précédent, l'attribut **Category** qui définit la catégorie de la propriété. Donnez-lui la valeur **(Identification)** :

Code VB

```
<Description("Identification du client"), Category("(Identification)")> _
Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)
            _id = value
        End Set
End Property
```

L'utilisation des parenthèses dans le nom de la catégorie n'est pas anodine. Elles vont assurer le positionnement de la catégorie en tout début de grille de propriétés qui est classée par ordre alphabétique ☺.

- Indiquez **Numéro** pour libellé d'affichage de la propriété à l'aide de l'attribut **DisplayName** :

Code VB

```
    <Description("Identification du client"), Category("(Identification)"), _
      DisplayName("Numéro")> _
    Public Property Id() As Integer
        Get
```

```
        Return _id
    End Get
    Set(ByVal value As Integer)
        _id = value
    End Set
End Property
```

- Enfin, indiquez que la propriété est en lecture seule à l'aide de l'attribut **ReadOnly**.

Code VB

```
  <Description("Identification du client"), Category("(Identification)"), _
   DisplayName("Numéro"), [ReadOnly](True)> _
  Public Property Id() As Integer
    Get
      Return _id
    End Get
    Set(ByVal value As Integer)
      _id = value
    End Set
End Property
```

Attention, cet attribut est un peu spécial dans la mesure où il porte à confusion avec le mot clé **ReadOnly** de Visual Basic.
L'attribut **ReadOnly** marque la propriété en lecture seule à l'attention de *l'affichage* dans la grille de propriétés alors que le mot clé **ReadOnly** indique que la propriété ne peut être que lue et interdit de ce fait l'usage d'un accesseur **Set**.
Deux solutions pour éviter toute confusion :

- Soit vous utilisez le nom complet de l'attribut **System.ComponentModel.ReadOnly**,
- Soit vous encadrez le nom de crochets : **[ReadOnly]**.

- Répétez l'opération pour toutes les autres propriétés, en respectant le tableau ci-dessous. Notez que l'attribut **ReadOnly** est inutile sur les autres propriétés.

Propriété	Description	DisplayName	Category
Entreprise	Raison sociale de l'entreprise.	(Société)	(Identification)
Nom	Nom et prénom du contact principal du client.	Contact	(Identification)
Titre	Fonction du contact principal du client.	Fonction	(Identification)
Adresse	Adresse de l'entreprise.	Adresse	Adresse postale
Ville	Ville de résidence de l'entreprise.	Ville	Adresse postale
Region	Région de résidence de l'entreprise.	Région	Adresse postale
CodePostal	Code postal du bureau postal	Code postal	Adresse postale

	distributeur.		
Pays	Pays de résidence de l'entreprise.	Pays	Adresse postale
Telephone	Numéro de téléphone du standard de l'entreprise.	Téléphone	Coordonnées téléphoniques
Telecopie	Numéro de la télécopie principale de l'entreprise.	Fax	Coordonnées téléphoniques

La classe résultante est la suivante :

Code VB

```
Imports System.ComponentModel
Public Class Client

    Public Sub New()
        _id = Environment.IdCounter
    End Sub

    Private _id As Integer = 0
    Private _entreprise As String = String.Empty
    Private _nom As String = String.Empty
    Private _titre As String = String.Empty
    Private _adresse As String = String.Empty
    Private _ville As String = String.Empty
    Private _region As String = String.Empty
    Private _codepostal As String = String.Empty
    Private _pays As String = String.Empty
    Private _telephone As String = String.Empty
    Private _telecopie As String = String.Empty

    <Description("Identification du client"), Category("(Identification)"), _
    DisplayName("Numéro"), [ReadOnly](True)> _
    Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)
            _id = value
        End Set
    End Property

    <Description("Raison sociale de l'entreprise"), Category("(Identification)"), _
     DisplayName("(Société)")>
     Public Property Enterprise() As St
        Get
            Return _entreprise
        End Get
        Set(ByVal value As String)
            _entreprise = value
```

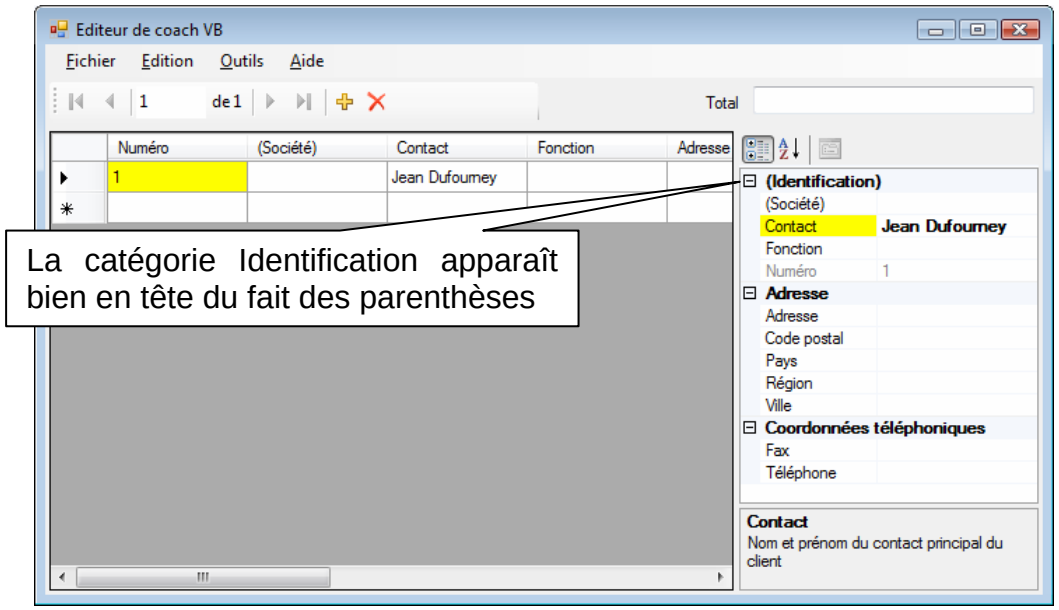
Entourez le mot **Société** de parenthèses pour faire apparaître cette propriété avant toutes les autres dans la catégorie **Identification**.

```
        End Set
    End Property
    <Description("Nom et prénom du contact principal du client"),
Category("(Identification)"), _
     DisplayName("Contact")> _
     Public Property Nom() As String
        Get
            Return _nom
        End Get
        Set(ByVal value As String)
            _nom = value
        End Set
    End Property

    <Description("Fonction du contact principal du client"),
Category("(Identification)"), _
     DisplayName("Fonction")> _
     Public Property Titre() As String
        Get
            Return _titre
        End Get
        Set(ByVal value As String)
            _titre = value
        End Set
    End Property

    <Description("Adresse de l'entreprise"), Category("Adresse"), _
    DisplayName("Adresse")> _
    Public Property Adresse() As String
        Get
            Return _adresse
        End Get
        Set(ByVal value As String)
            _adresse = value
        End Set
    End Property

    <Description("Ville de résidence de l'entreprise"), Category("Adresse"), _
     DisplayName("Ville")> _
     Public Property Ville() As String
        Get
            Return _ville
        End Get
        Set(ByVal value As String)
            _ville = value
        End Set
    End Property

    <Description("Région de résidence de l'entreprise"), Category("Adresse"), _
    DisplayName("Région")> _
```

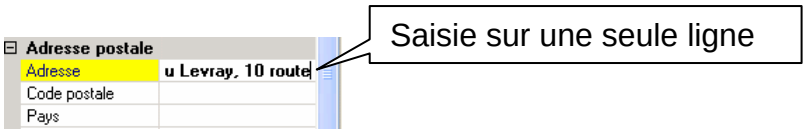
```
    Public Property Region() As String
        Get
            Return _region
        End Get
        Set(ByVal value As String)
            _region = value
        End Set
    End Property

    <Description("Code postal de l'entreprise"), Category("Adresse"), _
    DisplayName("Code postal")> _
    Public Property CodePostal() As String
        Get
            Return _codepostal
        End Get
        Set(ByVal value As String)
            _codepostal = value
        End Set
    End Property

    <Description("Pays de résidence de l'entreprise"), Category("Adresse"), _
    DisplayName("Pays")> _
    Public Property Pays() As String
        Get
            Return _pays
        End Get
        Set(ByVal value As String)
            _pays = value
        End Set
    End Property

    <Description("Numéro de téléphone du standard de l'entreprise"),
Category("Coordonnées téléphoniques"), _
    DisplayName("Téléphone")> _
    Public Property Telephone() As String
        Get
            Return _telephone
        End Get
        Set(ByVal value As String)
            _telephone = value
        End Set
    End Property

    <Description("Numéro de télécopie principal de l'entreprise"),
Category("Coordonnées téléphoniques"), _
   DisplayName("Fax")> _
   Public Property Telecopie() As String
        Get
            Return _telecopie
        End Get
```

```
        Set(ByVal value As String)
            _telecopie = value
        End Set
    End Property

End Class
```

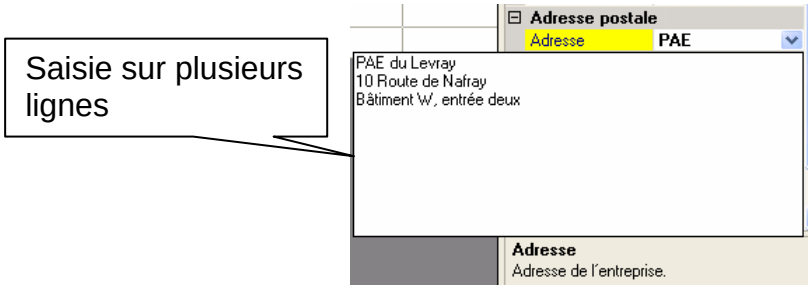
5. Testez le fonctionnement de la grille de données :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Nouveau**.
   - Ajoutez un nouveau client en cliquant le bouton :



Alors ? Ca le fait non ?
Et si on améliorait encore un peu la chose ? Dans l'état actuel, quand l'utilisateur saisit une adresse dans la grille de propriétés, il ne dispose que d'une seule ligne :



Ce serait nettement plus confortable s'il disposait de plusieurs lignes de saisie, comme suit :

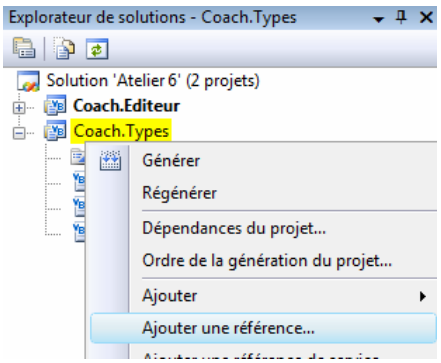


Comme d'habitude, on ne va pas réinventer la poudre... Le Framework .NET a des solutions toutes prêtes et propose plusieurs éditeurs de propriétés personnalisés. Par défaut l'éditeur d'une chaine de caractères est mono ligne, mais il est possible de préciser que nous souhaitons éditer la propriété avec plusieurs lignes. Ces éditeurs se trouvent dans l'assemblage **System.Design.dll**, et pour les utiliser, il faut aussi référencer l'assemblage **System.Drawing.dll**.

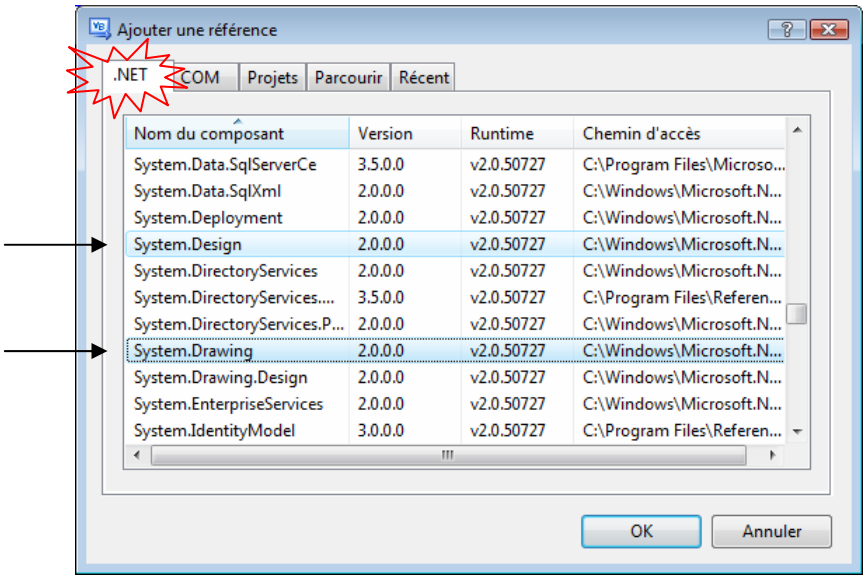
Pour avoir une vue d'ensemble sur les éditeurs de propriétés :
http://msdn.microsoft.com/fr-fr/library/ms171839.aspx

Vous pouvez bien sûr créer vos propres éditeurs personnalisés, par exemple une boîte de dialogue complexe.
Pour apprendre à implémenter un éditeur de propriétés personnalisé :
http://msdn.microsoft.com/fr-fr/library/ms171840.aspx

6. Configurez un éditeur multi ligne sur la propriété Adresse de l'objet Client :
   - Dans l'**Explorateur de solutions**, faites un **clic-droit** sur la racine du projet **Coach.Types** > **Ajouter une référence**.



   - Dans la fenêtre **Ajouter une référence**, sélectionnez l'onglet **.NET** :
   - Dans la liste des assemblages, sélectionnez **System.Design** et **System.Drawing** :



- Validez par **Ok** pour ajouter les deux assemblages à la liste des références du projet **Coach.Types**.
- En en-tête du fichier **Client.vb**, ajoutez deux directives **Imports** de référence aux espaces de nommage **System.ComponentModel.Design** et **System.Drawing.Design** :

Code VB

```
Imports System.ComponentModel
Imports System.ComponentModel.Design
Imports System.Drawing.Design

Public Class Client
…
End Class
```

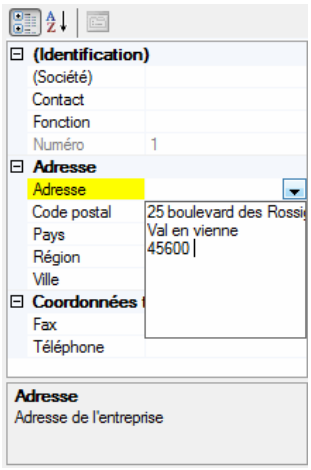
- Sur la déclaration de la propriété **Adresse** de la classe **Client**, indiquez que vous souhaitez utiliser un éditeur multi ligne à l'aide de l'attribut **Editor** :

Code VB

```
<Description("Adresse de l'entreprise"), Category("Adresse"), _
 DisplayName("Adresse"), _
 Editor(GetType(MultilineStringEditor), GetType(UITypeEditor))> _
 Public Property Adresse() As String
 …
 End Property
```

7. Testez le fonctionnement de l'éditeur multi ligne :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Nouveau**.

- Ajoutez un nouveau client en cliquant le bouton .
- Cliquez sur la ligne d'adresse pour faire apparaître l'éditeur multi ligne.



4 Pour aller plus loin…

4.1 Les types génériques

Dans cet atelier, nous avons utilisé la classe générique **List(T)** du Framework .NET. Sachez que vous pouvez définir vous aussi des types génériques personnalisés. Par exemple au lieu de définir une classe **Client<u>s</u>** manipulant une liste de **Client**, vous pourriez définir une classe **Item<u>s</u>** manipulant une liste d'objets dont vous ne connaissez pas encore le type au moment de la définition de la classe.
Votre classe **Items** doit être capable de s'adapter pour exécuter les mêmes fonctionnalités quelque soit le type de données qu'elle manipule. Le type manquant est défini en tant que paramètre de la classe, un peu comme pour une procédure.
Cela donnerait une classe du type :

Code VB

```
Public Class Items(Of T)
    Inherits List(Of T)
End Class
```

Type de données passé en paramètre de la classe.

Exemple d'une utilisation du paramètre T dans la définition de la classe Items.

Pour tout savoir sur les types génériques, voici un lien intéressant :
http://msdn.microsoft.com/fr-fr/library/w256ka79.aspx

4.2 Libérer la mémoire

Le Framework .NET utilise en standard un mécanisme de libération de la mémoire que l'on nomme le **garbage collector** (sorte de « collecteur de déchets » appelé aussi ramasse-miettes).
Le garbage collector se comporte un peu comme le service de ménage d'une entreprise. En effet, si un employé renverse du café sur la moquette de son bureau, la tâche sera nettoyée de manière certaine mais pas immédiatement après l'incident. De même, le garbage collector nettoie systématiquement la mémoire allouée aux objets qui ne sont plus utilisés par votre application mais on ne sait pas exactement quand (en réalité au meilleur moment déterminé par le moteur d'optimisation du garbage collector).
Bref, vous n'avez donc pas à vous soucier de la libération de la mémoire allouée à vos objets. Sauf que le garbage collector ne s'occupe que de la mémoire allouée dans le contexte du runtime du Framework.NET, c'est-à-dire sur des objets dits *managés*. Or vos objets manipulent très souvent des ressources qui ne sont pas managées (par le runtime) telles que les handles de fichier ou les connexions aux bases de données…etc.
C'est pour cette raison que se poser quelques questions concernant la libération de la mémoire peut s'avérer utile…

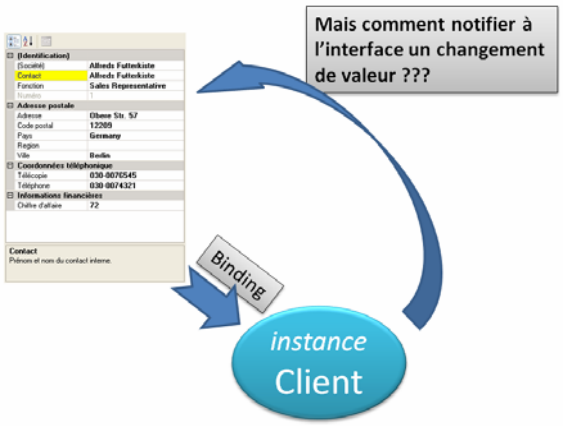
msdn Pour en savoir plus sur comment nettoyer des ressources non managées :

http://msdn.microsoft.com/fr-fr/library/b1yfkh5e.aspx
Pour tout savoir sur le Garbage Collector :
http://msdn.microsoft.com/fr-fr/library/0xy59wtx.aspx

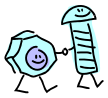
4.3 Utiliser des interfaces

Il se trouve que l'objet **Client** que nous avons écrit présente une petite limitation. Cette limitation ne concerne pas l'usage que nous avons fait de l'objet dans cet atelier aussi est-il difficile de la remarquer.

Le problème concerne la mise à jour de l'information qui ne se fait que dans un seul sens. En effet, les contrôles d'affichage comme la grille de propriétés sont capables de mettre à jour automatiquement (en utilisant les mécanismes de binding) toute propriété des objets. Mais que se passerait-il si une propriété d'objet (pas nécessairement l'objet courant donné par la propriété **MainBindingSource.Current** de la source de données d'ailleurs) était mise à jour, sans passer par un contrôle d'affichage (par programmation par exemple) ?



Par quel mécanisme cette propriété pourrait-elle notifier à tous les contrôles qui s'en servent que sa valeur vient de changer et qu'ils doivent donc se rafraîchir ?
La solution est d'implémenter sur l'objet **Client** ce qu'on appelle une interface (dans notre cas fournie par le Framework.NET) nommée **INotifyPropertyChanged**. Cette dernière fournit un évènement nommé **PropertyChanged** qu'il suffit de déclencher dans chaque accesseur **Set** des propriétés de l'objet **Client** pour notifier les contrôles d'un changement de valeur sur la propriété associée.



C'est quoi une **interface** ?
Une interface ressemble à une classe mais diffère par le fait qu'elle contient uniquement les signatures de ses membres. C'est-à-dire que dans une interface vous trouvez les noms, paramètres et autres informations de définition des propriétés, méthodes et évènements. En revanche, vous n'y trouvez pas leur contenu (leur *implémentation*).
Une interface est donc destinée à être *implémentée* à un moment ou à

un autre, dans une ou plusieurs classes qui reprendront rigoureusement la même signature pour chaque membre défini dans l'interface et en implémenteront le contenu (corps).
Dans le Framework, une convention de nommage veut que toutes les interfaces commencent par un **I** majuscule.

A quoi sert une interface alors ?
Si une classe implémente une interface donnée, alors vous êtes à même d'affirmer qu'elle implémente les membres de l'interface avec rigoureusement la même signature.

Pour en savoir un peu plus sur l'interface **INotifyPropertyChanged** :
http://msdn.microsoft.com/fr-fr/library/system.componentmodel.inotifypropertychanged.aspx
Pour apprendre à utiliser et définir des interfaces :
http://msdn.microsoft.com/fr-fr/library/28e2e18x.aspx

Manipuler des données XML

Sommaire

1 INTRODUCTION 3

1.1 CONTEXTE FONCTIONNEL 3

2 SAUVEGARDER DES DONNEES EN XML 5

2.1 SERIALISER DES DONNEES EN XML 6
2.2 DE SERIALISER DES DONNEES XML 18

3 MANIPULER DES DOCUMENTS XML 30

3.1 CREER UN DOCUMENT XML 30
3.2 UTILISER LES LITTERAUX XML 39
3.3 MANIPULER LES DONNEES 42
3.4 UTILISER LES PROPRIETES D'AXE XML 48
3.5 RECHERCHER DES DONNEES 51

4 POUR ALLER PLUS LOIN… 60

1 Introduction

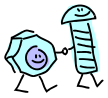
1.1 Contexte fonctionnel

Rappel du contexte fonctionnel du tutorial du coach VB

L'objectif du tutorial du Coach VB est d'accompagner les développeurs à la découverte et la prise en main du langage Visual Basic (VB) pour la construction d'applications avec une approche orientée objet.

Pour rappel, vous pouvez repérer facilement deux caractéristiques importantes du langage à l'aide des logos suivants en marge :

Ce logo marque une fonctionnalité de VB ou de Visual Studio qui permet de développer vite (et juste ☺).

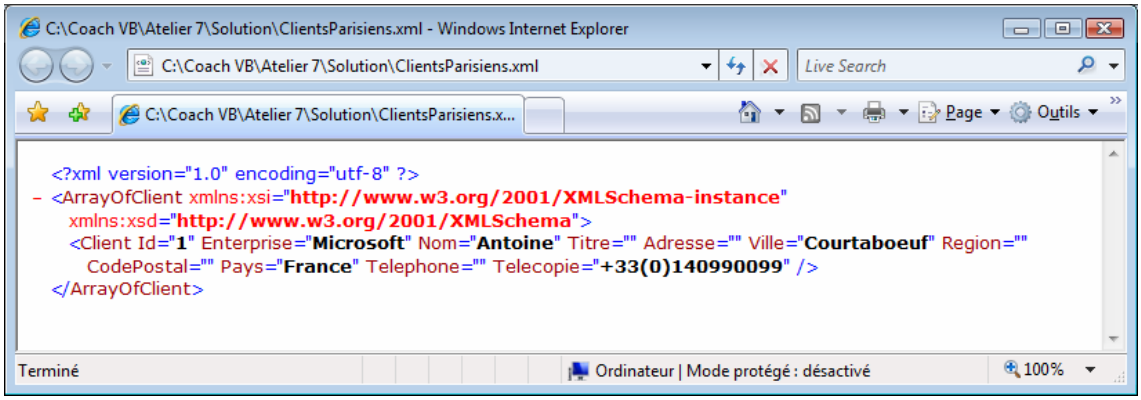


Ce logo met en évidence une caractéristique de la programmation orientée objet.

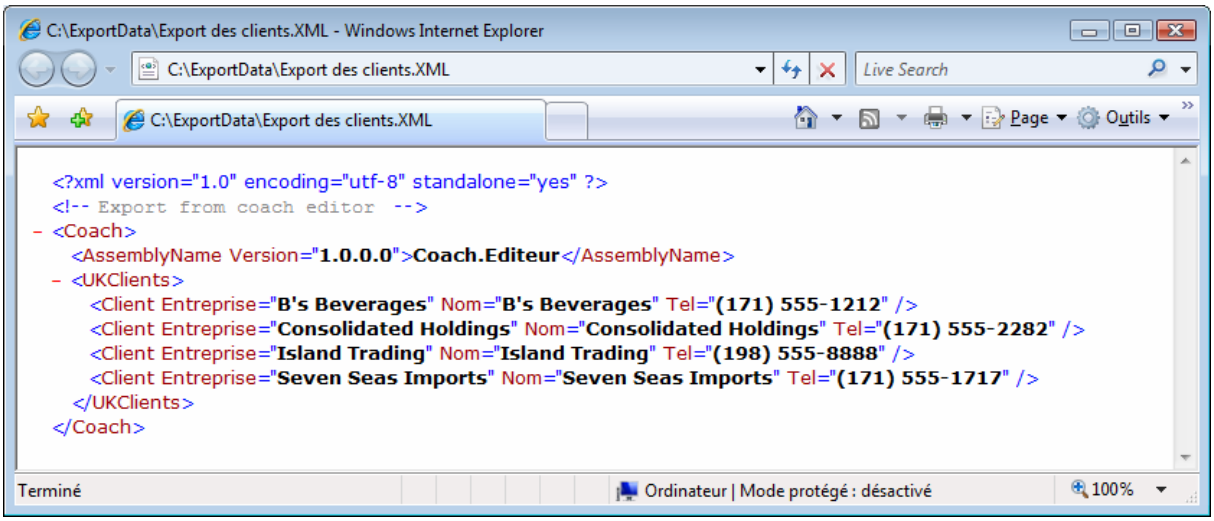
Contexte fonctionnel du septième atelier

Dans ce septième atelier, nous allons abandonner la sauvegarde des données en base de données pour adopter un scénario différent qui consiste à rendre persistantes les informations sur disque, au moyen du format XML. Cette approche est en effet à considérer dans des scénarios plus complexes, tels que les scénarios déconnectés.

Le principe consiste à réutiliser les options **Ouvrir**, **Enregistrer** et **Enregistrer sous** du menu **Fichier** que nous avions déjà exploitées lors de l'atelier 4. L'utilisateur peut choisir l'emplacement du fichier et enregistrer ou rouvrir les données à volonté. Le format du fichier de sauvegarde sera du type :



Dans un second temps, nous implémenterons une fonctionnalité d'export pour permettre à l'utilisateur de n'exporter que toute ou une partie des informations clientes. Le fichier ressemblera à ceci :



Contexte technique
Dans cet atelier, l'objectif est d'apprendre à manipuler des données XML. La première partie illustre le procédé de *sérialisation XML* d'un objet mémoire. La seconde partie explique comment créer un document XML et le manipuler avec la technologie *LINQ to XML*.

A la fin de cet atelier, vous saurez comment :

- Construire et manipuler un document XML,
- Rechercher des données dans un document XML,
- Sérialiser un objet en XML,
- Dé-sérialiser des données XML en objet.

La solution de cet atelier est disponible dans le répertoire **..\Atelier 7\Solution**. Les fichiers utiles, auxquels font référence les exercices sont disponibles dans le répertoire **..Atelier 7\Fichiers utiles**.

2 Sauvegarder des données en XML

Dans cet exercice, vous allez apprendre à :

- Reconnaître un document XML,
- Utiliser les processus de sérialisation et dé-sérialisation du Framework .NET,
- Utiliser les propriétés d'objet.

Objectif
L'objectif de cet exercice est d'implémenter les fonctions de sauvegarde et de lecture d'une liste de clients sur disque c'est-à-dire dans un état *persistant* (dit aussi *transportable*) dans le but de le stocker ou de le transférer à une application tierce.
Entre nous, c'est exactement ce que nous avions fait lors de l'atelier 4 dédié à la manipulation de fichier. La seule nouveauté réside ici dans le fait que nous allons adopter un format de sauvegarde beaucoup plus puissant que le format Texte délimité, à savoir le format XML.

Justement, c'est quoi **XML** ?
XML est l'abréviation de e**X**tensible **M**arkup **L**anguage, ce qui donne littéralement en français *langage de marquage extensible*. Il s'agit d'une norme ouverte, maintenant très largement répandue, utilisée pour décrire le contenu d'un document (c'est-à-dire des données).

On parle de langage de *marquage* parce qu'il est constitué de balises qui *marquent* l'information. Tout élément XML est délimité par une *balise de début* et une *balise de fin*. Entre les deux, on inscrit les données.

Ce langage est *extensible* parce qu'on peut l'étendre comme on veut. Le nom des balises peut être issu d'une grammaire bien déterminée (c'est le cas par exemple du protocole SOAP qui permet d'interagir avec un service web) mais vous pouvez vous-même décidé du format de vos données en créant votre propre **schéma** de document.

Voici un exemple de document XML :

Directive de déclaration

```
<?xml version="1.0" encoding="utf-8"?>
<Data>
        <Client id="1">Microsoft</Client>
        <Client id="2">Agilcom</Client>
</Data>
```

Attribut d'un Elément

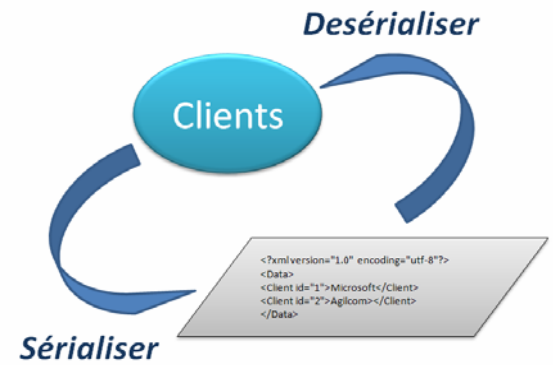
C'est quoi un *attribut XML* ?
Les attributs d'un élément XML contiennent des informations uniques sur l'élément, par exemple l'id d'un client. Ils sont directement accrochés à la balise ouvrante d'un élément XML sous la forme :

<Element **NomDeLAttribut = "ValeurDeLAttribut"** >…</Element>

En somme, un document XML n'est rien d'autre qu'une chaîne de caractères, c'est-à-dire du texte. Voilà pourquoi c'est un format qui facilite les échanges de données, quelque soit la plateforme et les protocoles d'échange en présence. Tout le monde sait manipuler une chaîne de caractères…

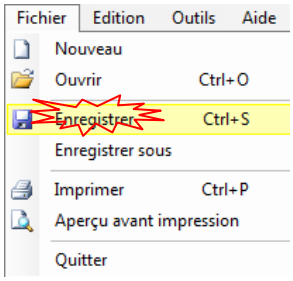
Pour tout savoir sur XML, voici l'adresse du centre de développement XML sur MSDN :
http://msdn2.microsoft.com/fr-fr/xml/default(en-us).aspx

Mais comment faire pour transformer une liste d'objets **Clients** en XML ?
Nous disposons d'une liste d'objets de type **Client**, représentée par une instance d'objet de type **Clients**. Cet objet réside en mémoire RAM de l'ordinateur. Le processus qui consiste à rendre un objet en mémoire sous une forme persistante est appelé la *sérialisation*. Le processus inverse, c'est-à-dire « réhydrater » une instance d'objet à partir de sa forme persistante est appelé *dé-sérialisation* (fallait y penser …). Nous allons donc sérialiser notre instance d'objet **Clients** en **XML** !

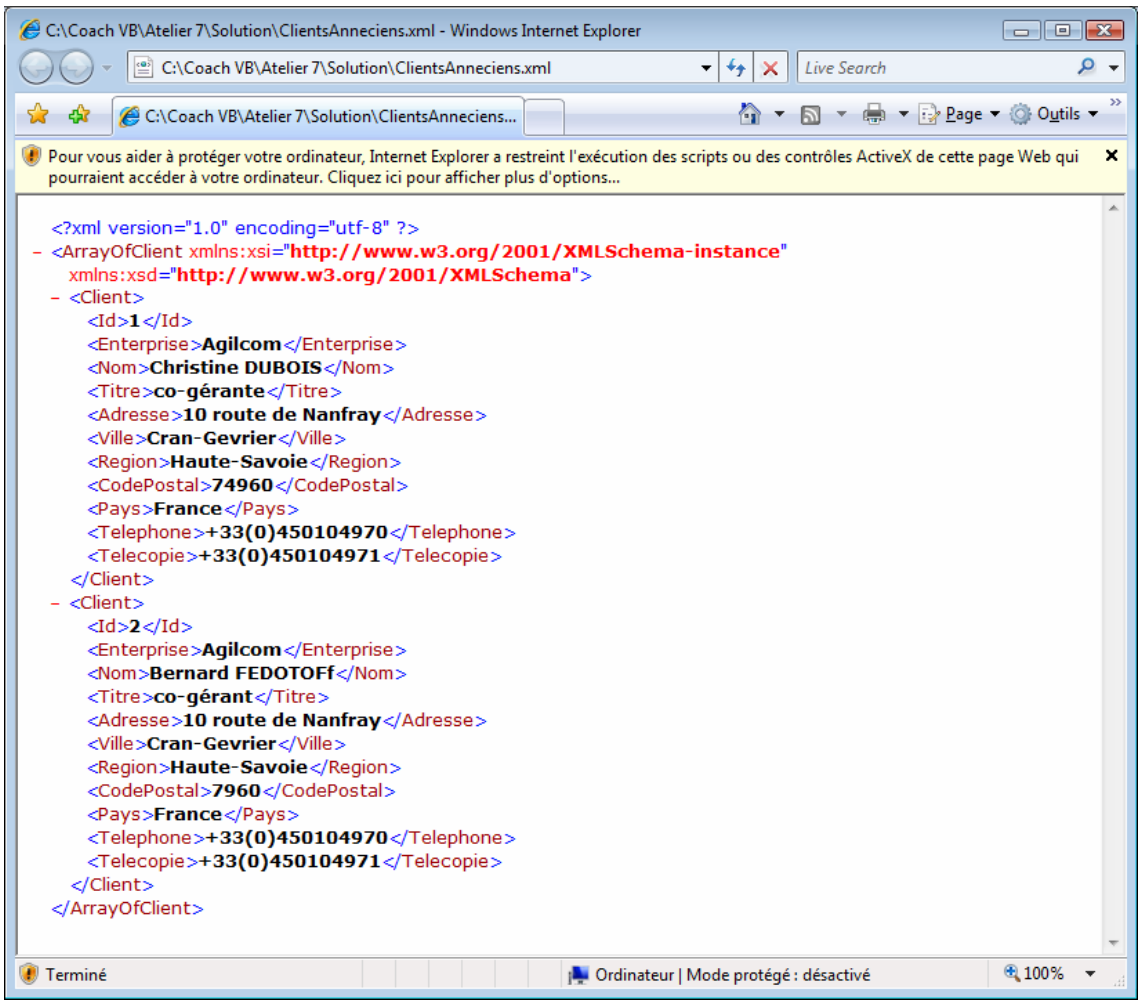


2.1 Sérialiser des données en XML

Contexte fonctionnel
La sérialisation de la liste de clients de l'éditeur doit se faire lorsque l'utilisateur clique le menu **Fichier** > **Enregistrer** (ou **Fichier** > **Enregistrer sous** si l'emplacement et le nom du fichier ne sont pas encore déterminés).



Les données de la grille sont alors enregistrées au format XML dans le fichier correspondant. On doit obtenir un fichier du type :




```
<?xml version="1.0" encoding="utf-8" ?>
- <ArrayOfClient xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  - <Client>
      <Id>1</Id>
      <Enterprise>Agilcom</Enterprise>
      <Nom>Christine DUBOIS</Nom>
      <Titre>co-gérante</Titre>
      <Adresse>10 route de Nanfray</Adresse>
      <Ville>Cran-Gevrier</Ville>
      <Region>Haute-Savoie</Region>
      <CodePostal>74960</CodePostal>
      <Pays>France</Pays>
      <Telephone>+33(0)450104970</Telephone>
      <Telecopie>+33(0)450104971</Telecopie>
    </Client>
  - <Client>
      <Id>2</Id>
      <Enterprise>Agilcom</Enterprise>
      <Nom>Bernard FEDOTOFf</Nom>
      <Titre>co-gérant</Titre>
      <Adresse>10 route de Nanfray</Adresse>
      <Ville>Cran-Gevrier</Ville>
      <Region>Haute-Savoie</Region>
      <CodePostal>7960</CodePostal>
      <Pays>France</Pays>
      <Telephone>+33(0)450104970</Telephone>
      <Telecopie>+33(0)450104971</Telecopie>
    </Client>
  </ArrayOfClient>
```

Déroulement de l'exercice :

1. Ouvrez le projet de démarrage de l'atelier :
   - Lancez Visual Studio à partir du menu **Démarrer** > **Tous les programmes** > **Microsoft Visual Basic 2008 Express Edition**.
   - Menu **Fichier** > **Ouvrir un projet**.
   - Récupérez le projet de démarrage de cet atelier dans le répertoire : **..\Atelier 7\Démarrage\Atelier 7.sln**.

Ce projet de démarrage vous est proposé car il n'est pas tout à fait identique au projet de solution issu de l'atelier précédent. Il est en effet dépourvu des éléments et fonctionnalités d'accès aux données en base (Access & SQL Server) devenues inutiles dans ce scénario applicatif.

Comment fait-on pour **sérialiser** un objet mémoire ?
Le Framework .Net nous fournit des **sérialisateurs** c'est-à-dire des classes qui réalisent la transformation demandée. Pourquoi fournit-il plusieurs sérialisateurs ? Tout simplement, parce qu'il est possible de sérialiser des objets sous différentes formes, dont celle qui nous intéresse, à savoir au format XML.

Quel est le résultat d'une sérialisation ?
Le processus de sérialisation que nous allons utiliser aboutit sur une chaîne de caractères représentant le document XML qu'il nous faudra ensuite écrire dans un fichier.

Et comment va-t-on écrire le résultat de la sérialisation dans un fichier ?
Exactement comme nous l'avons fait lors de l'atelier 4 pour l'écriture dans un fichier texte, via un objet **StreamWriter**, puisque nous avons vu qu'un document XML n'est rien d'autre qu'un document texte.

Attention ! Avant de vous lancer à sérialiser un objet, il faut vous assurer qu'il est *sérialisable*, c'est-à-dire que le Framework va savoir comment le transformer sous une forme persistante.
Les types standards du Framework .Net sont tous sérialisables (c'est pratique ☺). Evidemment pour un objet personnalisé, il est nécessaire d'indiquer explicitement qu'il est *sérialisable*. Cela concerne la classe mais également tous les types des données membres public de celle-ci (qui représente la forme statique d'un objet à un instant donné).

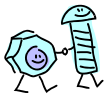
Pour tout savoir sur la sérialisation :
http://msdn2.microsoft.com/fr-fr/library/7ay27kt9.aspx

Pour tout savoir sur la sérialisation XML :
http://msdn2.microsoft.com/fr-fr/library/182eeyhh.aspx

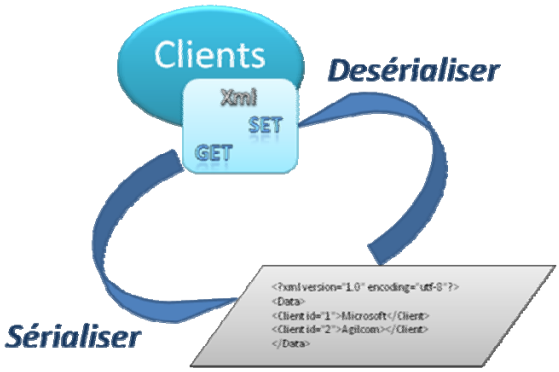
Il nous reste à déterminer où implémenter la sérialisation de la liste **Clients** dans le code ?
Un premier réflexe serait de coder le processus directement dans le gestionnaire d'évènement lié au clic de l'option de menu **Enregistrer**. Mais le processus de sérialisation d'un objet (de-même que le processus inverse) est très fortement lié à l'objet lui-même. Une autre solution consiste donc à l'implémenter directement au sein de la définition de l'objet lui-même, c'est-à-dire dans la classe **Clients**.

Comme le résultat de la sérialisation est une chaîne de caractères, nous allons ajouter à notre objet **Clients** une nouvelle propriété, nommée par exemple **Xml**, de type **String**. Elle contiendra la chaîne à transmettre aux objets **SreamReader** et **StreamWriter** qui gèrent la lecture/écriture dans

le fichier.



Pourquoi faire une propriété ?
Parce que les propriétés d'objet présentent l'avantage de fournir deux procédures **Get** et **Set** agissant respectivement sur la lecture et l'écriture de la propriété (cf. atelier 6 précédent). Ce sont donc les éléments dont nous avons besoin pour implémenter les processus de sérialisation/dé-sérialisation !



Vous êtes prêt ? Alors allons-y.

2. Commencez par indiquer au Framework .NET que vos objets sont *sérialisables* :
   - Dans l'**Explorateur de Solutions**, double-cliquez sur le fichier **Clients.vb** du projet **Coach.Types** afin d'afficher le code de la classe **Clients.**
   - Sur la déclaration de la classe **Clients**, indiquez qu'elle est sérialisable en rajoutant l'attribut **Serializable** comme suit :

Code VB

```
<Serializable()> _
Public Class Clients
    Inherits List(Of Client)
End Class
```

Comme la classe **Clients** est une liste d'objets de type **Client**, il faut également préciser au Framework .NET que la classe **Client** est elle-même *sérialisable*.

- Dans l'**Explorateur de Solutions**, double-cliquez sur le fichier **Client.vb** du projet **Coach.Types** afin d'afficher le code de la classe **Client.**
- Sur la déclaration de la classe **Client**, indiquez qu'elle est sérialisable en rajoutant l'attribut **Serializable** comme suit :

Code VB

```
…
<Serializable()> _
Public Class Client
```

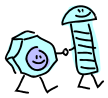
```
…
End Class
```

3. Ajoutez maintenant une propriété **Xml** au type **Clients** :

   - Basculez dans le code de la classe **Clients**.
   - Ajoutez une propriété nommée **Xml** de type **String** au type **Clients**.

Code VB

```
<Serializable()> _
Public Class Clients
   Inherits List(Of Client)

   Private _xml As String
   Public Property Xml() As String
      Get
         Return _xml
      End Get
      Set(ByVal value As String)
         _xml = value
      End Set
   End Property

End Class
```



Pour simplifier ici, on peut se passer de la variable privée _**xml**. En effet un membre privé est destiné à être utilisé dans le contexte de sa classe, or dans notre cas la classe **Clients** est vide (hormis la définition de la propriété **Xml** évidemment).

   - Supprimez la définition et l'utilisation du membre privé _**xml**. Retournez une chaîne de caractères vide dans la procédure **Get** en remplacement de la variable privée _**xml** :

Code VB

```
Private _xml As String
Public Property Xml() As String
      Get
         Return _xml
         Return String.Empty
      End Get
      Set(ByVal value As String)
         _xml = value
      End Set
End Property
```

Une bonne pratique de programmation consiste à prévoir un bloc de code **Try/Catch** dans chacune des deux procédures **Get** et **Set** pour gérer d'éventuelles exceptions au moment des processus de sérialisation/dé-

sérialisation.

- Ajoutez un bloc **Try/Catch** dans chacune des deux procédures :

Code VB

```
Public Property Xml() As String
      Get
         Try
            Return String.Empty
         Catch ex As Exception
            Return String.Empty
         End Try
      End Get
      Set(ByVal value As String)
         Try

         Catch ex As Exception

         End Try
      End Set
End Property
```

Rentrons maintenant au cœur du processus de sérialisation d'un objet mémoire :
Pour transformer un objet de type **Clients** en une chaîne de caractères au format XML, on va avoir besoin d'un « écrivain » qui va « écrire » le contenu de l'objet **Clients** dans une sorte de « bouquin en mémoire » au moyen d'un « stylo » (Evidemment c'est très imagé mais ça permet d'y voir plus clair ☺).

- Le « stylo » XML, c'est une classe **XmlSerializer** qui va nous le fournir.
- Le « bouquin en mémoire » est un objet de type **MemoryStream**.
- L' « écrivain » qui tient le stylo est un objet de type **XmlTextWriter**.

La procédure **Get** doit renvoyer le contenu du « bouquin en mémoire ». En quelque sorte, il nous faut donc un « lecteur » pour terminer le processus.
Vous allez rire mais dans .Net, un « écrivain » ne sait pas lire…
Donc en dernier lieu, on va demander à un « lecteur » de lire le livre en bloc. Ce lecteur, c'est l'objet de type **StreamReader** que vous avez déjà rencontré à l'atelier 4 pour lire un fichier Texte-délimité.

- Au début du fichier **Clients.vb**, ajoutez les trois espaces de nommage nécessaires pour implémenter le processus :

Code VB

```
Imports System.IO
Imports System.Xml
Imports System.Xml.Serialization
```

```
Imports System.Text

<Serializable()> _
Public Class Clients
…
End Class
```

- Implémentez le processus complet de sérialisation dans la procédure **Get** comme suit :

Code VB

```
  Get
      Try
        'Définir le bouquin [l'espace mémoire] où nous allons écrire
        Dim xmlMemoryStream As MemoryStream = _
              New MemoryStream(1024)
        ' Definir "l'écrivain" [de type XML] pour écrire à notre place dans
        ' le bouquin [l'espace mémoire]
        Dim xmlTextWriter As XmlTextWriter = _
              New XmlTextWriter(xmlMemoryStream, Encoding.UTF8)
        ' Définir la façon dont on va écrire[sérialiser] la
        ' structure de l'histoire c'est à dire le type de l'objet en cours
        Dim xmlSerialiser As XmlSerializer = _
              New XmlSerializer(Me.GetType())
        ' et hop, on écrit le contenu de l'histoire c'est-à-dire l'instance de
        ' l'objet en cours (mot clé Me)
        xmlSerialiser.Serialize(xmlTextWriter, Me)
        ' Stop, on  arrête d'écrire maintenant !
        xmlTextWriter.Flush()
        ' On se remet au début du bouquin
        xmlMemoryStream.Position = 0
        ' On definit un lecteur qui aura pour mission de venir
        ' lire le contenu du bouquin
        Dim xmlStreamReader As StreamReader = _
              New StreamReader(xmlMemoryStream)
        ' Le lecteur lit le bouquin et renvoie le contenu
        ' sous la forme d'une chaîne de caractères
        Return xmlStreamReader.ReadToEnd()
      Catch ex As Exception
        Return String.Empty
      End Try
  End Get
```

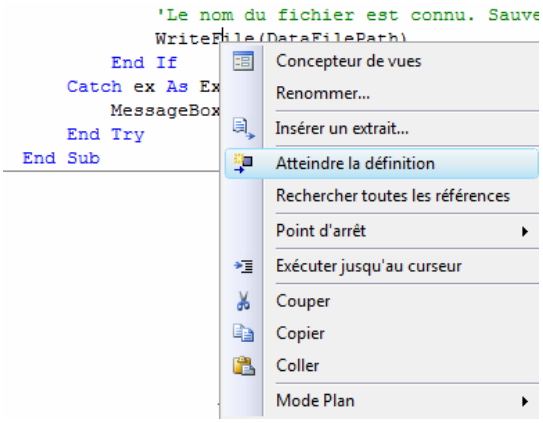
- Enregistrez vos changements en cliquant sur dans la barre d'outils de Visual Studio.
- Sélectionnez le menu **Générer** > **Générer la solution** pour vérifier qu'il n'y a pas d'erreur à la compilation.

4. Implémentez maintenant la sauvegarde des données de la grille dans un fichier d'extension *.xml :
   - Dans l'**Explorateur de Solutions**, double-cliquez sur le fichier **Main.vb** du projet **Coach.Editeur** afin d'afficher le code de la classe **Main**.
   - Dans la région de code « **Traitement des options de menu** », retrouvez le gestionnaire d'évènement **EnregistrerToolStripMenuItem_Click** :

```
92  #Region "Traitement des options de menu"
93      Private Sub Main_SizeChanged ...
98      Private Sub OptionsToolStripMenuItem_Click ...
109     Private Sub FormClose ...
115     Private Sub MaximiserLaFenêtreToolStripMenuItem_Click ...
118     Private Sub RestaurerLaFenêtreToolStripMenuItem_Click ...
121     Private Sub MinimiserLaFenêtreToolStripMenuItem_Click ...
124     Private Sub OuvrirToolStripMenuItem_Click ...
139     Private Sub EnregistrerToolStripMenuItem_Click(ByVal sender As System.Object, ByVal e As System.EventArg
140         Try
141             If String.IsNullOrEmpty(DataFilePath) Then
142                 'Il s'agit d'un nouveau fichier. Déclencher la pro      e Enregistrer Sous
143                 EnregistrersousToolStripMenuItem_Click(sender, e)
144             Else
145                 'Le nom du fichier est cor      Sauvegarder les données
146                 WriteFile(DataFilePath)
147             End If
148         Catch ex As Exception
149             MessageBox.Show("Une erreur est survenue au moment de la mise à jour", "Erreur de mise-à-jour")
150         End Try
151     End Sub
152     Private Sub EnregistrerSousToolStripMenuItem_Click ...
164     Private Sub NouveauToolStripMenuItem_Click ...
167 #End Region
```

Pour rappel la sauvegarde des données en elle-même se fait via la méthode **WriteFile** ② à laquelle on transmet en paramètre le nom du fichier de sauvegarde. Si les données n'ont encore jamais été enregistrées ①, la procédure invoque le gestionnaire d'évènement associé au clic sur le menu **Fichier** > **Enregistrer sous** de façon à ce que l'utilisateur puisse déterminer le nom et l'emplacement du fichier de sauvegarde.

- Faites un clic droit sur le mot **WriteFile** > **Atteindre la définition** pour basculer dans la définition de la procédure :

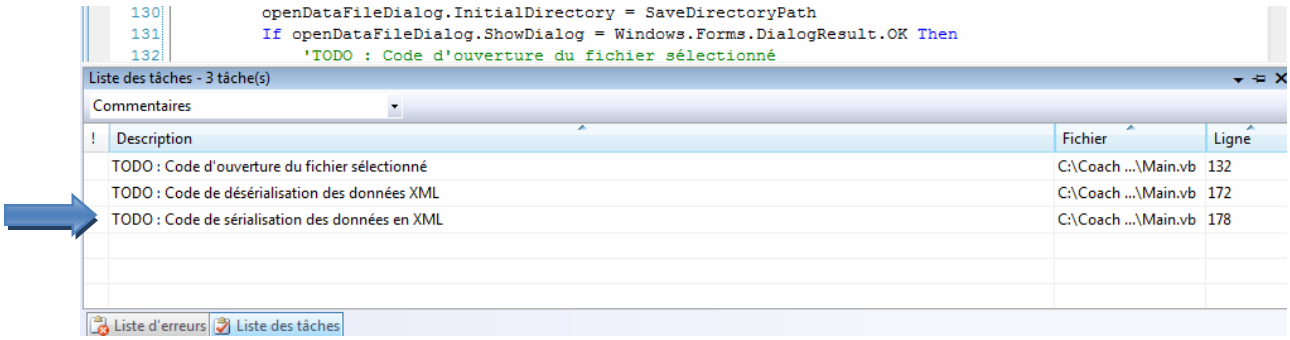


La procédure se situe dans la région de code **Traitement des fichiers de données** de la classe **Main** :

```
#Region "Traitement des fichiers de données"
    Function ReadFile(ByVal FileName As String) As Clients
        'Définition des variables locales à la fonction
        Dim result As Clients = Nothing
        'TODO : Code de désérialisation des données XML
        'Renvoi de la valeur de retour
        Return result
    End Function
    Sub WriteFile(ByVal FileName As String)
        Using sw = New System.IO.StreamWriter(FileName, False)
            'TODO : Code de sérialisation des données en XML
        End Using
    End Sub
#End Region
```

Notez que le code que nous devons compléter est situé à l'emplacement du commentaire **'TODO : Code de sérialisation des données en XML**.

Pour rappel, Visual Studio génère automatiquement une tâche pour chaque commentaire marqué du jeton **TODO** dans la liste des tâches de l'utilisateur (cf. §3.2 point 2 de l'Atelier 1). La fenêtre **Liste des tâches** est généralement affichée au bas de la page et accessible via la « poignée » de même nom. Si elle n'est pas disponible, faites la apparaître via le menu **Affichage** > **Autres fenêtres** > **Liste des tâches** :
Un double clic sur le commentaire vous renvoie automatiquement sur la ligne correspondante.



- Ajoutez la ligne de code suivante :

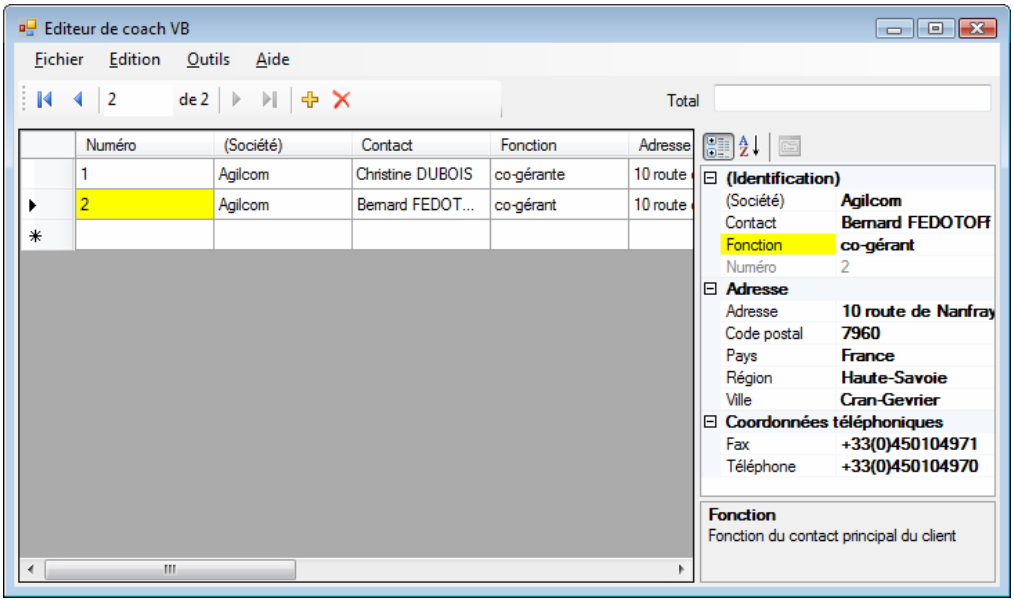
Code VB

```
Sub WriteFile(ByVal FileName As String)
    Using sw = New System.IO.StreamWriter(FileName, False)
        'TODO : Code de sérialisation des données en XML
        sw.Write(CType(MainBindingSource.DataSource, Clients).Xml)
    End Using
End Sub
```

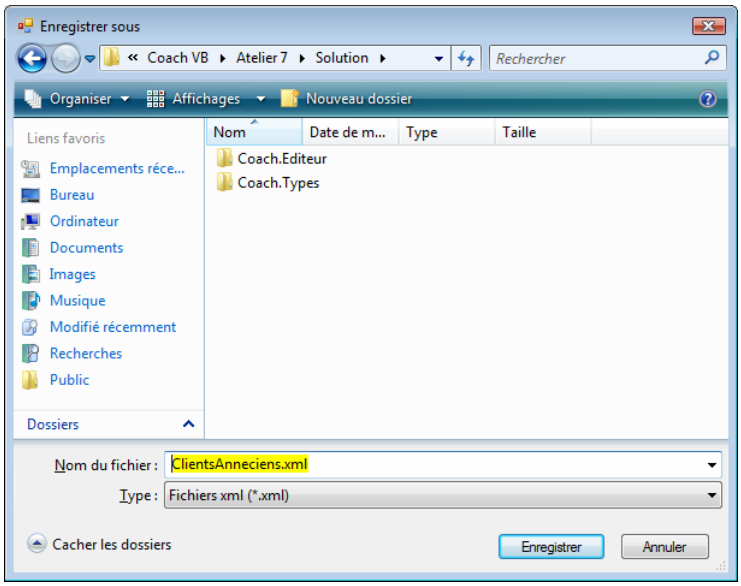
Que signifie cette ligne de code ?
Pour rappel, la liste de clients affichée dans la grille (que nous cherchons à sauvegarder sur disque) est la source de données gérée par l'objet **MainBindingSource**. La fonction **CType** convertit l'objet en **Clients**.
Le fait d'accéder à la propriété **Xml** de l'objet résultant **Clients** déclenche l'exécution de la procédure **Get** correspondante, dans laquelle nous avons implémenté le processus de sérialisation de l'objet en XML. La

chaîne de caractères résultante est alors écrite dans le fichier à l'aide de la méthode **Write** de l'objet **sw** de type **StreamWriter** et le tour est joué ☺ !

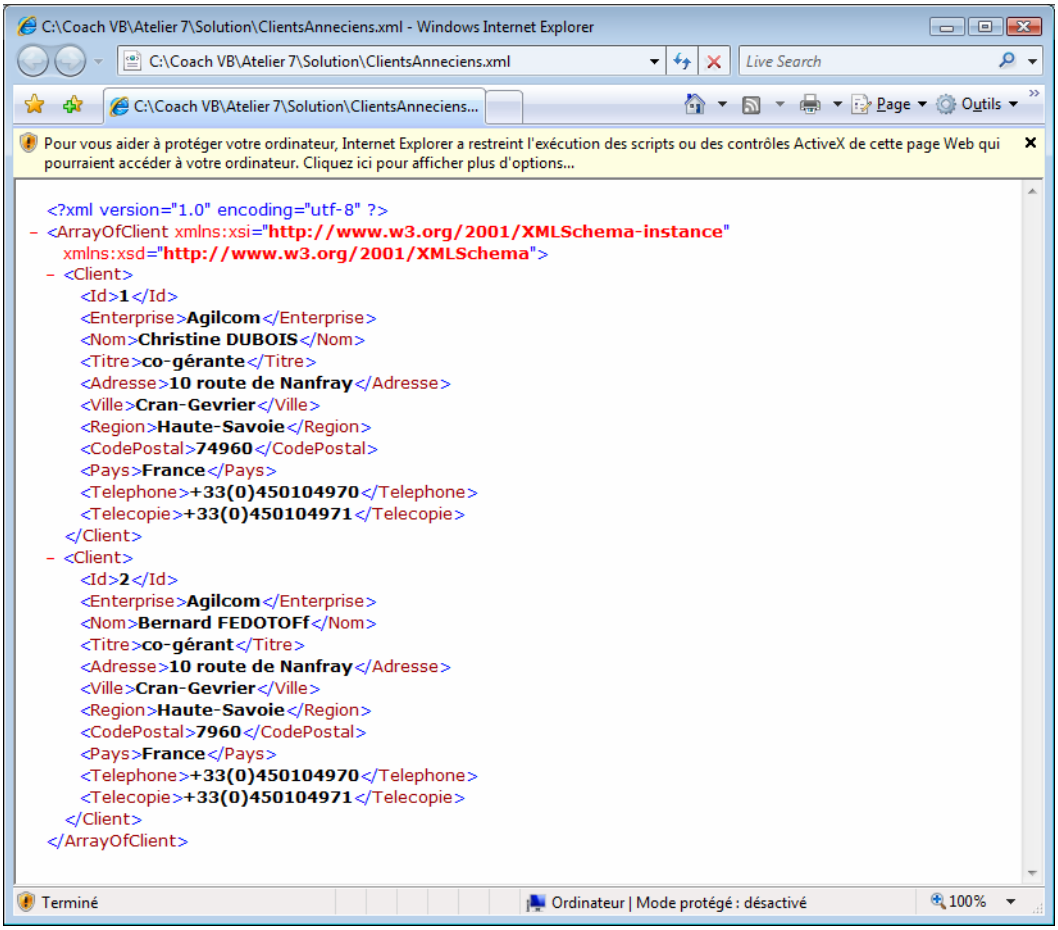
5. Et si on testait tout ça ?
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Nouveau**. Vous devez obtenir une liste de clients vide.
   - Saisissez de nouveaux clients en utilisant le bouton :



   - Sélectionnez le menu **Fichier** > **Enregistrer sous**.
   - Dans la boîte **Enregistrer sous**, choisissez un emplacement sur le disque et un nom de fichier de sauvegarde :



- Cliquez **Enregistrer**.
- Basculez sur le disque à l'emplacement du fichier.
- Ouvrez le fichier en double cliquant sur le fichier. En principe un fichier d'extension *.xml s'ouvre par défaut dans **Internet Explorer** :



Vous constatez que par défaut toutes les propriétés d'un objet **Client** deviennent un élément XML, comportant une balise de début et une balise de fin. Ce format peut s'avérer un peu « verbeux » et contribue à alourdir la taille du document XML. Dans le cas d'un échange avec une application tierce par exemple, cela peut s'avérer gênant.

Comme les propriétés de l'objet **Client** sont de type simple (**String** ou **Integer**), il n'y a qu'une seule valeur à sauvegarder pour chacune d'elle. Dans ce cas, le sérialisateur XML peut la sérialiser sous forme d'un attribut d'élément. Cela prend un peu moins de place (on économise la taille des balises de fermeture), et c'est plus exploitable.

Code XML : un objet Client sérialisé sous forme d'éléments

```
<Client>
  <Id>1</Id>
  <Entreprise>Microsoft France</Entreprise>
  <Nom>Coach C#</Nom>
  <Titre>Coach</Titre>
```

```
<Adresse>18 avenue de Quebec</Adresse>
<Ville>Villebon</Ville>
<Region>IDF</Region>
<CodePostal>91140</CodePostal>
<Pays>France</Pays>
<Telephone>0 825 827 829</Telephone>
<Telecopie>0 164 461 911</Telecopie>
</Client>
```

Code XML : le même objet Client sérialisé sous forme d'attributs

```
<Client Id="1" Entreprise="Microsoft France" Nom="Coach C#" Titre="Coach"
Adresse="18 avenue de Quebec" Ville="Villebon" Region="IDF"
CodePostal="91140" Pays="France" Telephone="0 825 827 829" Telecopie="0 164
461 911" />
```

Pour indiquer les propriétés à sérialiser sous forme d'attribut, il suffit de décorer la déclaration de la propriété avec l'attribut **XmlAttribute()**, qui se trouve dans l'espace de nommage **System.Xml.Serialization**. Dans le constructeur de cet attribut, il serait possible d'indiquer le nom de l'attribut XML qui serait utilisé pour sérialiser l'information.

Pour tout savoir sur **XmlAttribute** :
http://msdn.microsoft.com/fr-fr/library/system.xml.serialization.xmlattributeattribute.aspx

Pour en savoir un peu plus sur comment générer un XML qui corresponde exactement à ce que vous attendez (en anglais) :
http://msdn2.microsoft.com/fr-fr/library/ms950721.aspx

6. Implémentez la sérialisation par attributs :
   - Dans l'**Explorateur de solutions**, double cliquez sur le fichier **Client.vb** du projet **Coach.Types** pour ouvrir le code de la classe **Client**.
   - Au début du fichier, ajoutez l'espace de nommage **System.Xml.Serialization** :

Code VB

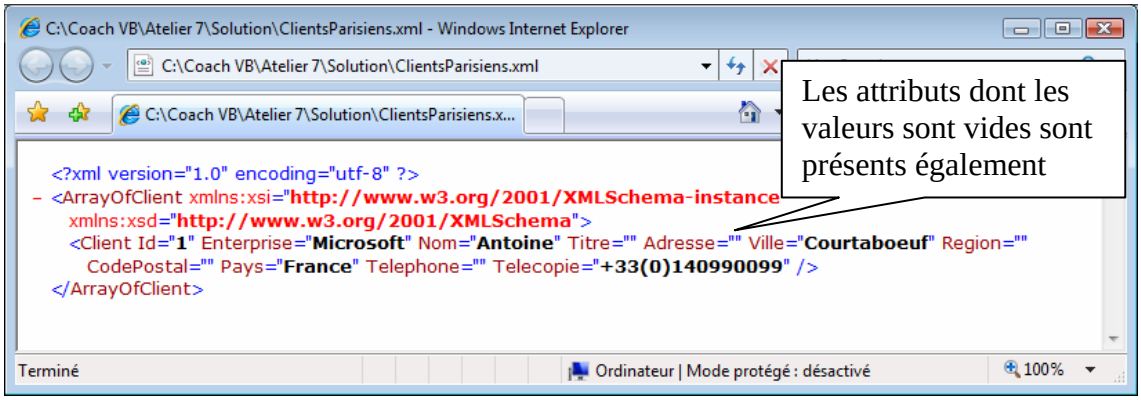
```
Imports System.ComponentModel
Imports System.ComponentModel.Design
Imports System.Drawing.Design
Imports System.Xml.Serialization

<Serializable()> _
Public Class Client
…
```

   - Sur chacune des propriétés de la classe **Client**, ajoutez l'attribut **XmlAttribute** comme suit :

Code VB

```
<Description("Identification du client"), Category("(Identification)"), _
    DisplayName("Numéro"), [ReadOnly](True), XmlAttribute()> _
    Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)
            _id = value
        End Set
    End Property
```

- Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
- Exécutez l'application en cliquant sur (ou touche F5).
- Cliquez le menu **Fichier** > **Nouveau**. Vous devez obtenir une liste de clients vide.
- Saisissez de nouveaux clients en utilisant le bouton .
- Sélectionnez le menu **Fichier** > **Enregistrer sous**.
- Dans la boîte **Enregistrer sous**, choisissez un emplacement sur le disque et un nom de fichier de sauvegarde.
- Cliquez **Enregistrer**.
- Basculez sur le disque à l'emplacement du fichier.
- Ouvrez le fichier en double cliquant sur le fichier :

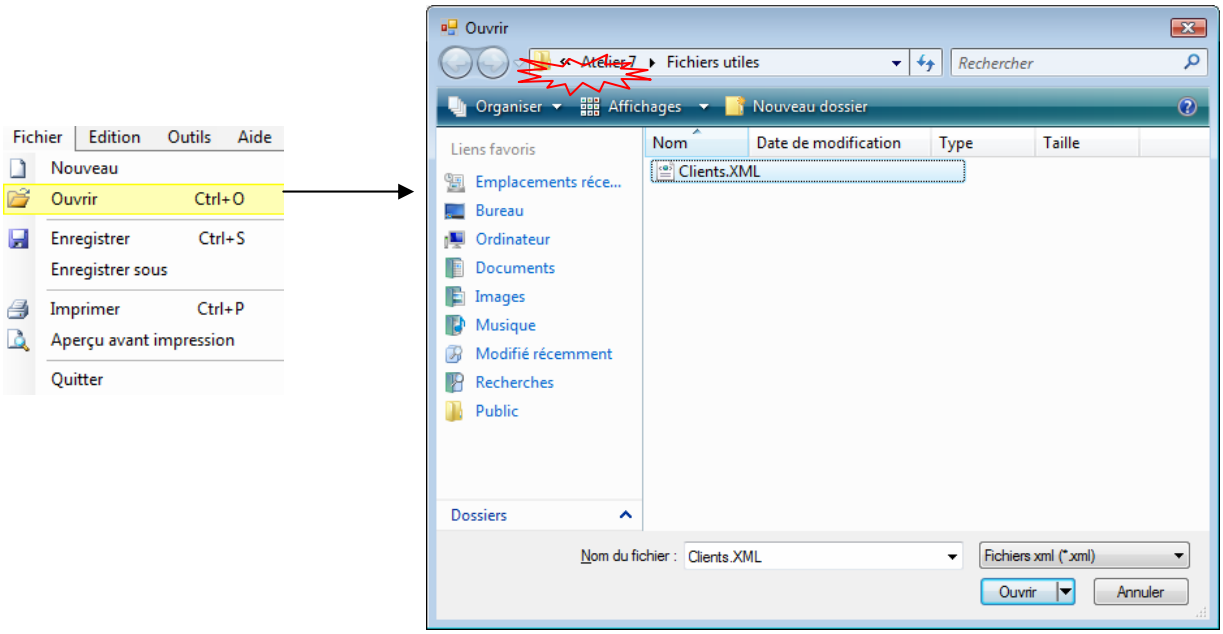


Bravo ! Il ne reste plus qu'à implémenter le processus inverse de dé-sérialisation dans la procédure **Set** de la propriété **Xml**.

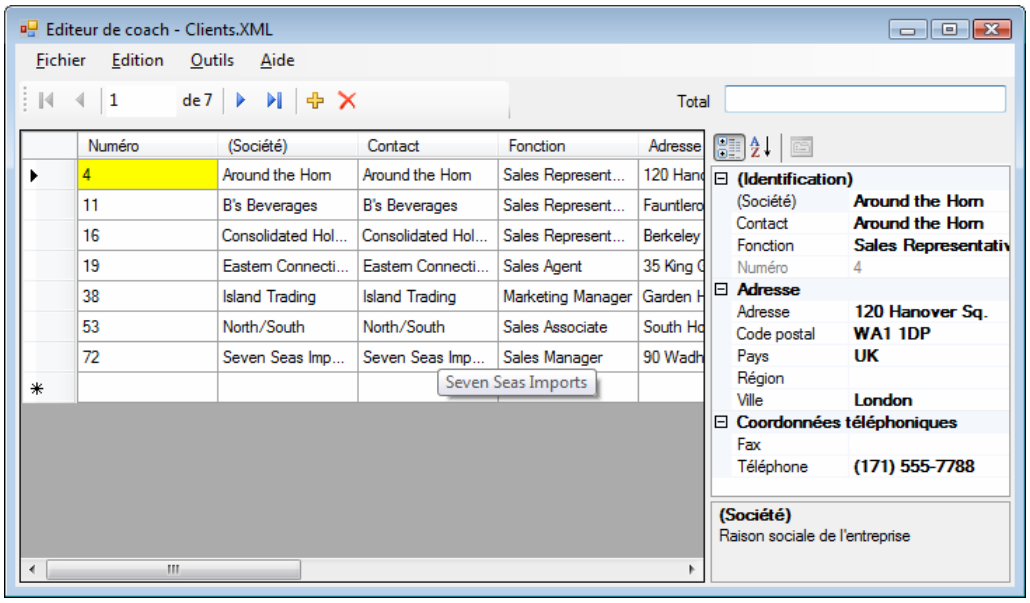
2.2 Dé sérialiser des données XML

Contexte fonctionnel

La dé-sérialisation de la liste de clients de l'éditeur doit se faire lorsque l'utilisateur clique le menu **Fichier** > **Ouvrir** et sélectionne un fichier au format XML sur le disque.



La grille de l'éditeur se charge alors avec les données lues à partir du fichier XML.



En quoi consiste le processus de dé-sérialisation d'une chaîne de caractères XML en un objet mémoire ?
La bonne nouvelle est que le procédé est beaucoup plus simple que la sérialisation :

- La chaîne de caractères à dé-sérialiser est transmise à l'accesseur **Set** de la propriété **Xml** via le paramètre **value**.
- A partir de cette chaîne, il s'agit de générer un bel objet **Clients** tout neuf et bien propre. Le même objet de type **XmlSerializer** va faire office de « stylo magique » pour réaliser l'opération.
- Le lecteur des données XML est un objet de type **XmlReader**.

Déroulement de l'exercice :

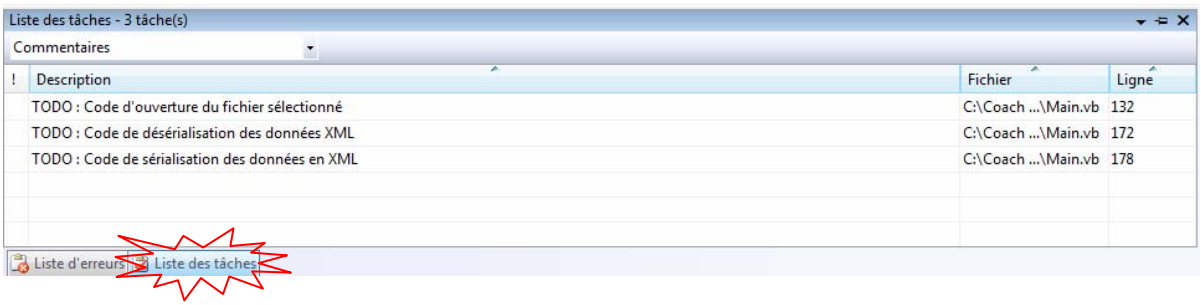
1. Ajoutez le code de dé-sérialisation à la procédure Set de la propriété Xml de la classe Clients :
   - Dans le code de la classe **Clients**, rajoutez les lignes suivantes à la procédure **Set** de la propriété **Xml** :

Code VB

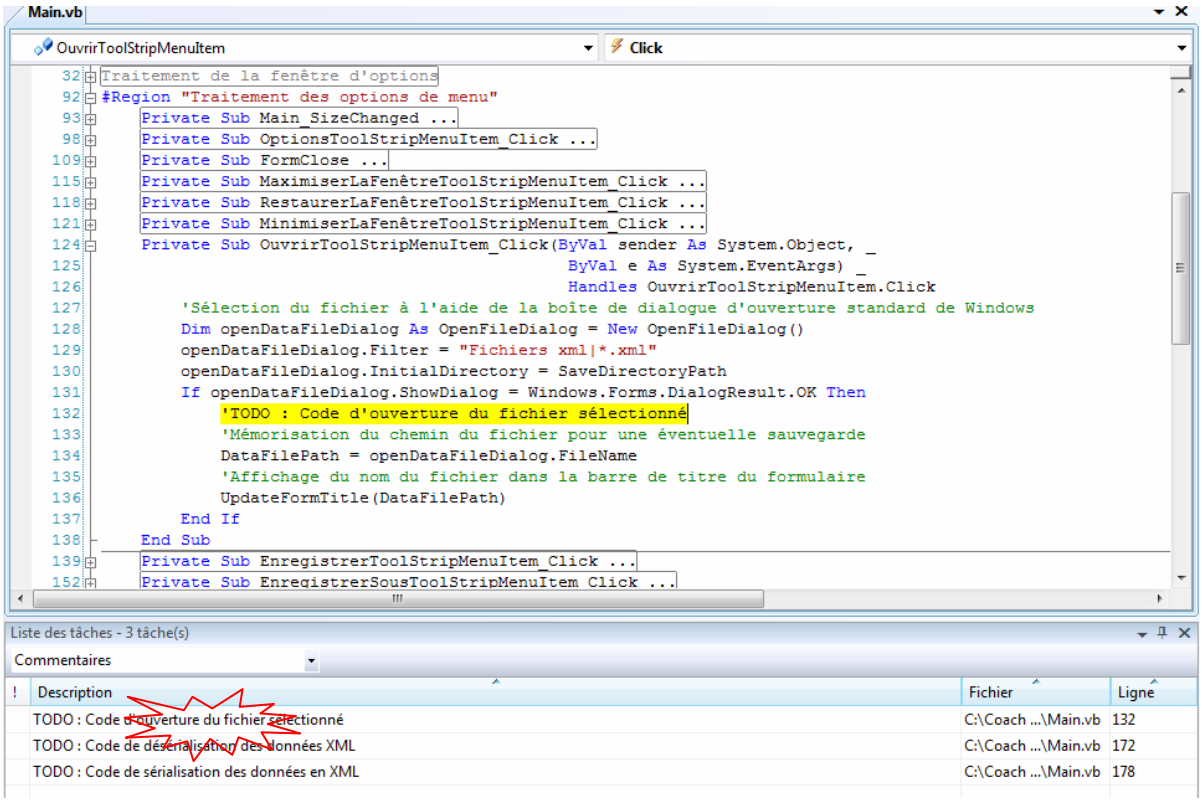
```
Set(ByVal value As String)
        Try
         ' Définir un lecteur de données XML basé sur la chaîne transmise
         ' dans le paramètre value de la procédure Set
         Using xmlR As XmlReader = _
                        XmlReader.Create(New StringReader(value))
            ' Vider la liste de clients actuelle (Me) de tout son contenu
            Me.Clear()
            ' Définir la façon dont on vas recréer[désérialiser] le contenu de
            ' l'histoire [l'objet précédement sauvegardé]
            Dim xmlSerialiser As XmlSerializer = _
                        New XmlSerializer(Me.GetType())
            ' et hop, on lit le contenu dans un objet tout neuf, avec un petit
            ' cast au passage car la méthode Deserialize renvoie un objet
            Dim clients As Clients = _
                        CType(xmlSerialiser.Deserialize(xmlR), Clients)
            ' Ajouter les clients lus à la liste de clients actuelle (Me)
            Me.AddRange(clients)
            ' Environment.IdCounter = Me.MaximumIdValue
          End Using
        Catch ex As Exception
        End Try
    End Set
```

   - Enregistrez vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Sélectionnez le menu **Générer** > **Générer la solution** pour vérifier qu'il n'y a pas d'erreur à la compilation.

2. Implémentez maintenant le chargement des données dans la grille à partir du fichier XML :
   - Affichez la fenêtre **liste des tâches** de Visual Studio.



- Repérez le commentaire **TODO : Code d'ouverture du fichier sélectionné** et double cliquez dessus pour basculer sur la ligne de code correspondante :



Pour rappel l'ouverture du fichier sélectionné par l'utilisateur se fait via la méthode **ReadFile** à laquelle on transmet en paramètre le nom du fichier XML. Dans l'Atelier 4, cette méthode retournait un objet de type **DataTable**. Dans notre cas, la méthode renverra un objet de type **Clients**.

- Ajoutez la ligne de code suivante d'appel de la méthode **ReadFile** :

Code VB

```
Private Sub OuvrirToolStripMenuItem_Click(…) Handles …
    'Sélection du fichier à l'aide de la boîte de dialogue …
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers xml|*.xml"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
        'TODO : Code d'ouverture du fichier sélectionné
        Dim listeClients As Clients = ReadFile(openDataFileDialog.FileName)
        'Mémorisation du chemin du fichier pour une éventuelle sauvegarde
        DataFilePath = openDataFileDialog.FileName
        'Affichage du nom du fichier dans la barre de titre du formulaire
        UpdateFormTitle(DataFilePath)
    End If
End Sub
```

- Faites un clic droit sur le mot **ReadFile** > **Atteindre la définition** pour basculer dans la définition de la procédure :

La procédure se situe dans la région de code **Traitement des fichiers de données** de la classe **Main** :

```
#Region "Traitement des fichiers de données"
    Function ReadFile(ByVal FileName As String) As Clients
        'Définition des variables locales à la fonction
        Dim result As Clients = Nothing
        Using sr As StreamReader = New StreamReader(FileName)
            'TODO : Code de désérialisation des données XML
        End Using
        'Renvoi de la valeur de retour
        Return result
    End Function
    Sub WriteFile(ByVal FileName As String)
        Using sw = New System.IO.StreamWriter(FileName, False)
            'TODO : Code de sérialisation des données en XML
            sw.Write(CType(MainBindingSource.DataSource, Clients).Xml)
        End Using
    End Sub
#End Region
```

Notez que le code que nous devons compléter est situé à l'emplacement du commentaire **'TODO : Code de dé-sérialisation des données XML**.

- Ajoutez les deux lignes de code suivantes :

Code VB

```
Function ReadFile(ByVal FileName As String) As Clients
    'Définition des variables locales à la fonction
    Dim result As Clients = Nothing
    Using sr As StreamReader = New StreamReader(FileName)
        'TODO : Code de désérialisation des données XML
        result = New Clients()
        result.Xml = sr.ReadToEnd()
    End Using
    'Renvoi de la valeur de retour
    Return result
End Function
```

Que signifie ce code ?
La méthode **ReadToEnd** de l'objet de type **StreamReader** lit l'intégralité du flux de données du fichier. Le résultat est affecté à la propriété **Xml** de l'objet **Clients**, ce qui déclenche l'exécution de la procédure **Set** correspondante, dans laquelle nous avons implémenté le processus de dé-sérialisation des données XML. L'objet **result** de type **Clients** est alors chargé avec la liste des clients et le tour est joué ☺ !

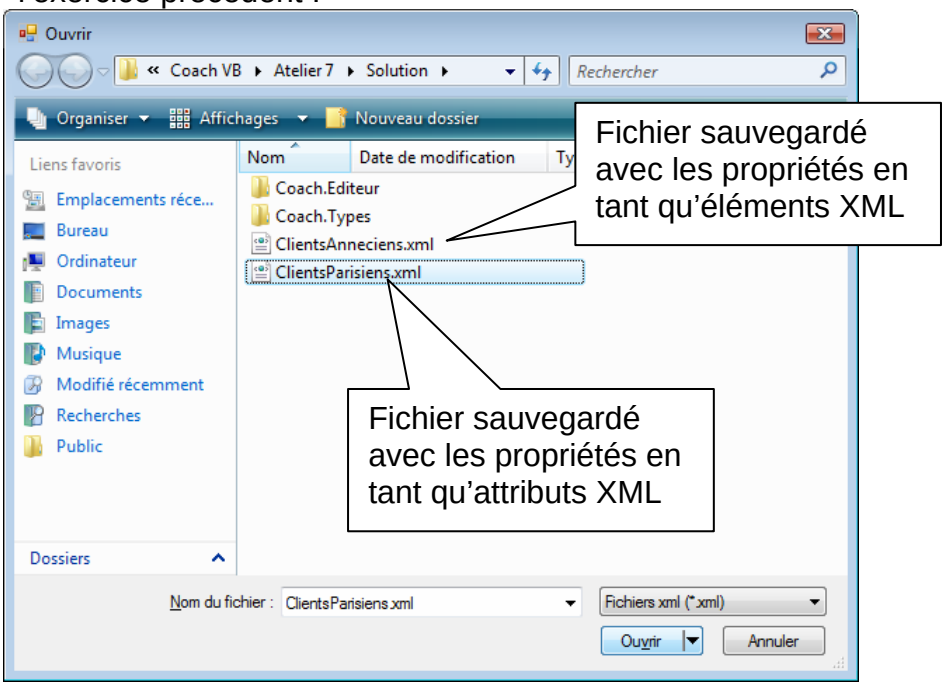
C'est presque terminé ! Pour que la liste des clients soit chargée dans la grille d'affichage de l'éditeur, il ne faut pas oublier d'effectuer la liaison de données avec le contrôle d'affichage.

- Rebasculez dans le gestionnaire d'évènement **OuvrirToolStripMenuItem_Click** de la classe **Main**.
- Ajoutez la liaison de la liste **Clients** avec le gestionnaire de liaison en utilisant la méthode **Bind** (que nous avions construite au point 4 du § 2.4 de l'atelier 6 précédent) :

Code VB

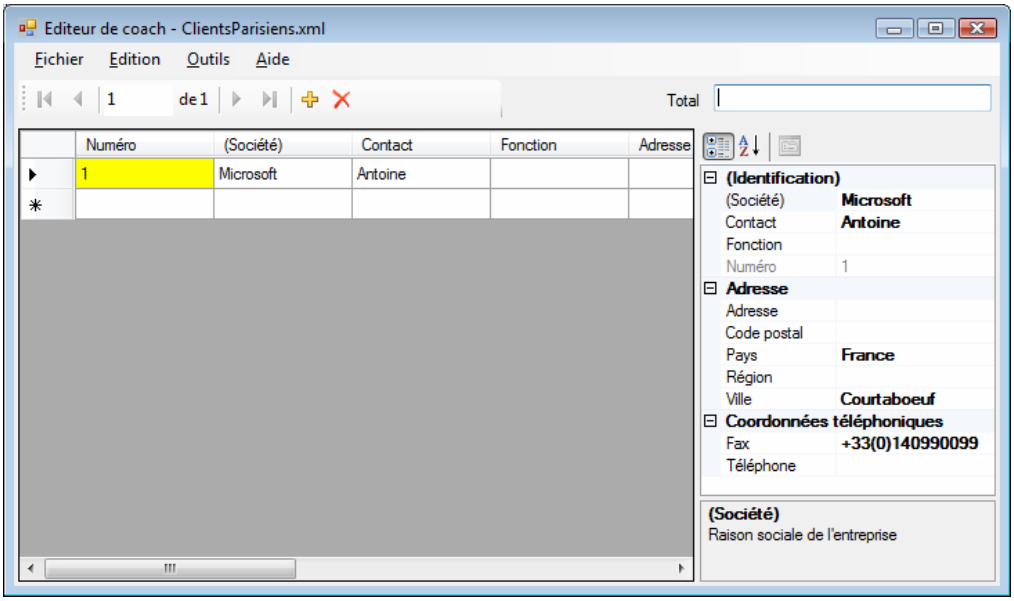
```
Private Sub OuvrirToolStripMenuItem_Click(…) Handles …
    'Sélection du fichier à l'aide de la boîte de dialogue …
    Dim openDataFileDialog As OpenFileDialog = New OpenFileDialog()
    openDataFileDialog.Filter = "Fichiers xml|*.xml"
    openDataFileDialog.InitialDirectory = SaveDirectoryPath
    If openDataFileDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
      'TODO : Code d'ouverture du fichier sélectionné
      Dim listeClients As Clients = ReadFile(openDataFileDialog.FileName)
      'Configuration du gestionnaire de liaison sur la source de données
      Bind(listeClients)
      'Mémorisation du chemin du fichier pour une éventuelle sauvegarde
      DataFilePath = openDataFileDialog.FileName
      'Affichage du nom du fichier dans la barre de titre du formulaire
      UpdateFormTitle(DataFilePath)
    End If
  End Sub
```

3. Il ne reste plus qu'à tester le fonctionnement du processus d'ouverture :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir**.
   - Sélectionnez le fichier d'extension *.xml que vous avez sauvegardé à l'exercice précédent :



Attention au fichier que vous sélectionnez car vous devez disposer de deux formats de fichier, l'un basé sur des éléments XML et l'autre sur des attributs. Comme votre classe **Client** a ses propriétés toujours décorées avec l'attribut **XmlAttribute**, la grille ne se chargera pas correctement si vous ouvrez le fichier sauvegardé avec des éléments uniquement…

- Cliquez **Ouvrir** pour charger la grille avec les données correspondantes:



Et voilà ! La sérialisation n'a plus de secret pour vous !

Si vous avez du mal avec les procédures **Get/Set** de l'objet **Clients**, n'hésitez pas à dérouler les tests en mode débogage ☺.

- Positionnez un point d'arrêt dans les procédures **Get** et **Set** de la classe **Clients**.
- Exécutez l'application en cliquant sur (ou touche F5).
- Cliquez le menu **Fichier** > **Nouveau**. Vous devez obtenir une liste de clients vide.
- Saisissez de nouveaux clients en utilisant le bouton .
- Sélectionnez le menu **Fichier** > **Enregistrer sous**.
- Dans la boîte **Enregistrer sous**, choisissez un emplacement sur le disque et un nom de fichier de sauvegarde.
- Cliquez **Enregistrer**.
- Vous devez basculez dans le débogueur de Visual Studio qui vous arrête dans la procédure **Get**.

```
Clients.vb | Main.vb
Clients                                          Xml
 1  Imports System.IO
 2  Imports System.Xml
 3  Imports System.Xml.Serialization
 4  Imports System.Text
 5
 6  <Serializable()> _
 7  Public Class Clients
 8      Inherits List(Of Client)
 9
10      Public Property Xml() As String
11          Get
12              Try
13                  'Définir le bouquin [l'espace mémoire] où nous allons écrire
14                  Dim xmlMemoryStream As MemoryStream = New MemoryStream(1024)
15                  ' Definir "l'écrivain public" [de type XML] pour écrire à notre place dans
16                  ' le bouquin [l'espace mémoire]
```

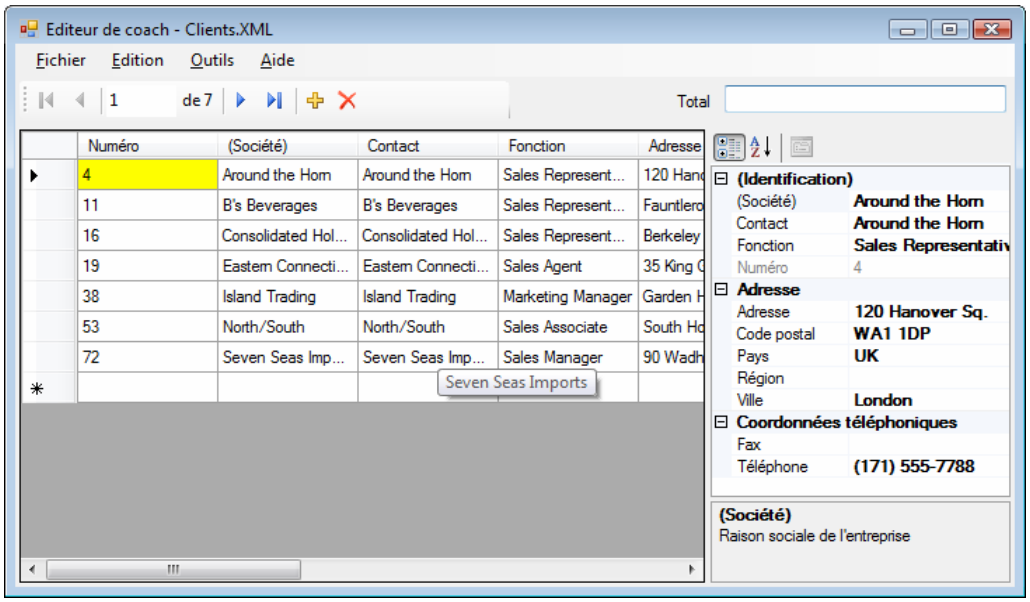
- Continuez l'exécution pas à pas en cliquant pour suivre l'avancement du processus.
- Une fois que la sérialisation est réalisée, ré ouvrez le fichier en cliquant le menu **Fichier** > **Ouvrir**.
- Sélectionnez le fichier d'extension *.xml que vous venez de sauvegarder.
- Cliquez **Ouvrir** pour charger les données dans la grille.
- Vous devez basculez dans le débogueur de Visual Studio qui vous arrête dans la procédure **Set**.

```
Clients.vb | Main.vb
Clients                                          Xml
23                  ' He gazier, on  arrête d'écrire maintenant !
24                  xmlTextWriter.Flush()
25                  ' Maintenant on se remet au début du bouquin
26                  xmlMemoryStream.Position = 0
27                  ' on demande à un lecteur de venir voir
28                  Dim xmlStreamReader As StreamReader = New StreamReader(xmlMemoryStream)
29                  ' Lecteur lambda, peut tu nous lire notre livre ? (c'est magique, non ??
30                  Return xmlStreamReader.ReadToEnd()
31              Catch ex As Exception
32                  Return String.Empty
33              End Try
34          End Get
35          Set(ByVal value As String)
36              Try
37                  'Définir un lecteur de données XML basé sur la chaîne transmise
38                  ' dans le paramètre value de la procédure Set
39                  Using xmlR As XmlReader = XmlReader.Create(New StringReader(value))
40                      ' retirer tous les clients en cours
41                      Me.Clear()
```

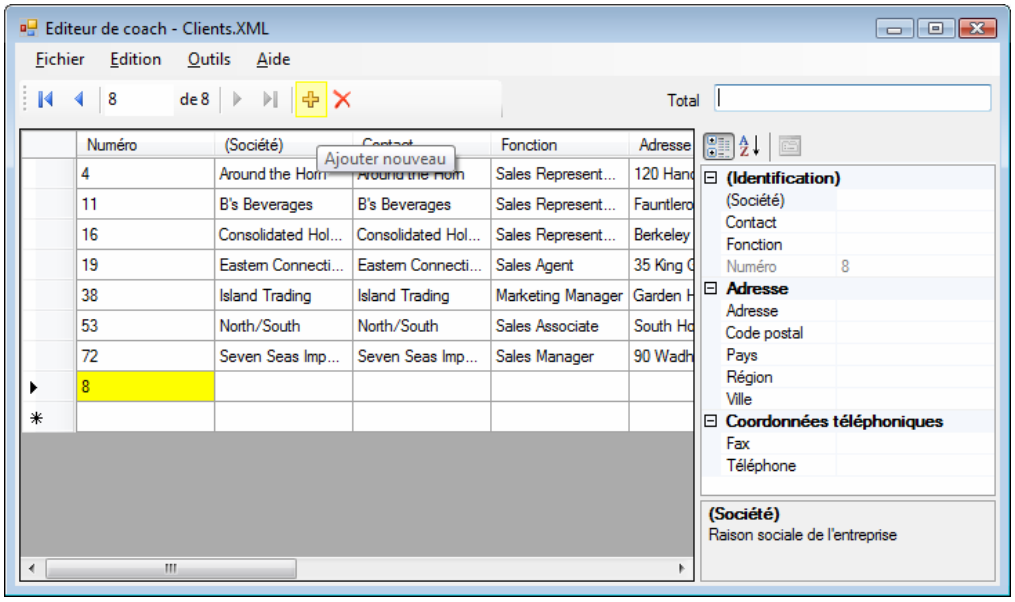
- Déroulez pas à pas le processus de dé-sérialisation.

C'est puissant ces propriétés d'objet non ?
Il reste juste un petit bug de cosmétique dans notre application. L'avez-vous remarqué ? Oui ? Non ?

- Ouvrez dans l'éditeur du coach le fichier **Clients.xml** fourni dans le répertoire **Atelier 7\Fichiers utiles** :



- Saisissez un nouveau client en utilisant le bouton .



Le nouveau client se voit attribué un **Id** de valeur **8**. C'est normal puisqu'il est le 8ème élément. Sauf que dans le cas présent, il risque donc d'y avoir redondance des Ids à un moment ou à un autre avec les Id des clients existants, qui eux vont de **4** à **72**…
Ce phénomène se produit lorsque vous avez des **Id** de clients plus grands que le nombre de clients que vous avez dans la liste.
C'est pour cela qu'il faut penser, après la lecture du fichier, à configurer l'**Id** courant de la classe statique **Environnement** avec la plus grande valeur (plus un) **Id** des clients lus.

4. Implémentez la vérification des Ids de Client à la lecture :
   - Basculez dans le code de la classe **Clients**.

- Ajoutez une propriété privée nommée **MaximumIdValue**, accessible **en lecture seule** et de type entier (**Integer**) :

Code VB

```
…
<Serializable()> _
Public Class Clients
   Inherits List(Of Client)

   Public Property Xml() As String
      …
   End Property

   Private _maximumIdValue As Integer = 0
   Public ReadOnly Property MaximumIdValue() As Integer
      Get
         Return _maximumIdValue
      End Get
   End Property

End Class
```

Initialisez le membre privé à 0

N'oubliez pas le mot clé **ReadOnly** pour indiquer que la propriété est en lecture seule

- A l'aide d'une boucle **Foreach**, balayez l'ensemble des objets **Client** de la liste de l'instance courante **Clients** (**Me**) afin de ne mémoriser que la valeur la plus grande de l'**Id** que vous allez rencontrer. C'est cette valeur que la procédure **Get** doit retourner :

Code VB

```
…
<Serializable()> _
Public Class Clients
   Inherits List(Of Client)

   Public Property Xml() As String
      …
   End Property

   Private _maximumIdValue As Integer = 0
   Public ReadOnly Property MaximumIdValue() As Integer
      Get
         For Each cli As Client In Me
            If (cli.Id > _maximumIdValue) Then
               _maximumIdValue = cli.Id
            End If
         Next
         Return _maximumIdValue
      End Get
   End Property
```

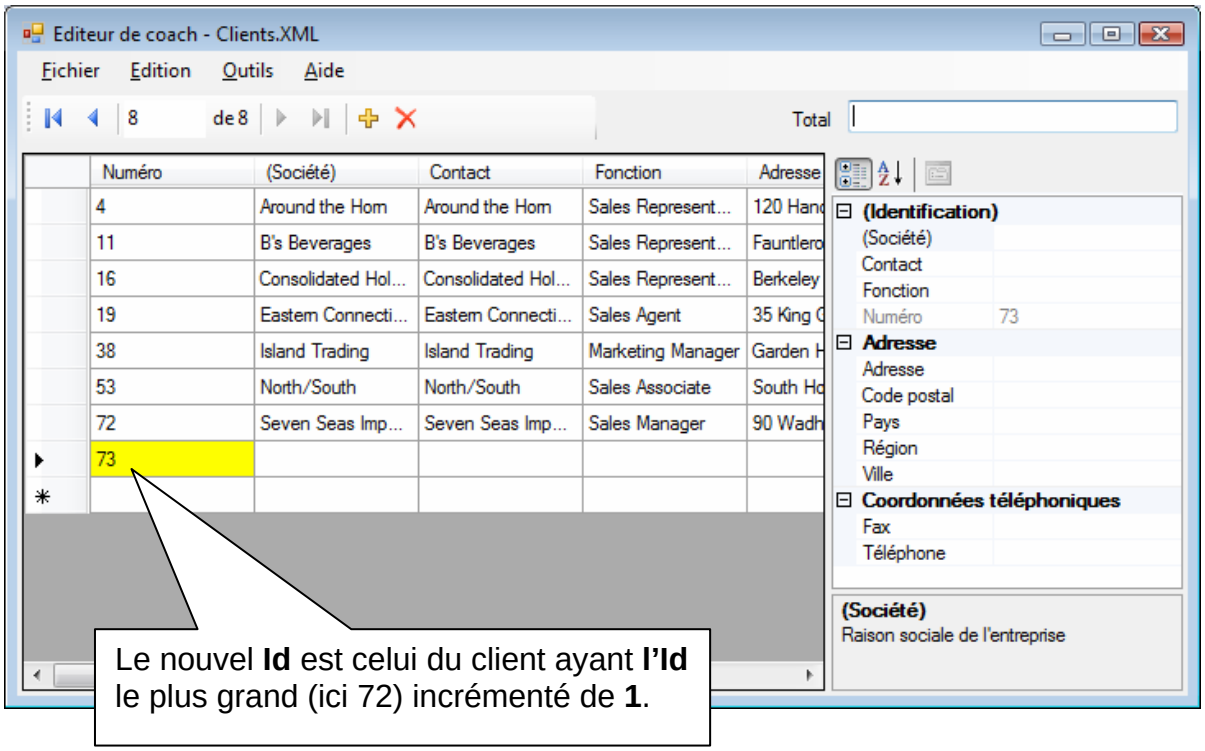
```
End Class
```

- Toujours dans le code de la classe **Clients**, dans l'accesseur **Set** de la propriété **Xml**, configurez la valeur courante du compteur d'Id, donné par la propriété **IdCounter** de la classe **Environnement**, avec la valeur maximum parmi les clients lus :

Code VB

```
…
<Serializable()> _
Public Class Clients
    Inherits List(Of Client)

    Public Property Xml() As String
        …
        Set(ByVal value As String)
            Try
                'Définir un lecteur de données XML basé sur la chaîne transmise
                ' dans le paramètre value de la procédure Set
                Using xmlR As XmlReader = _
                        XmlReader.Create(New StringReader(value))
                    …
                    ' Reste maintenant à remplacer la liste des clients en cours
                    Me.AddRange(clients)
                    Environment.IdCounter = Me.MaximumIdValue
                End Using
            Catch ex As Exception

            End Try
        End Set
    End Property

    Private _maximumIdValue As Integer = 0
    …
    End Property

End Class
```

5. Il ne reste plus qu'à tester :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir**.
   - Sélectionnez le fichier **Clients.xml** fourni dans le répertoire **…Atelier 7\Fichiers utiles**.
   - Saisissez un nouveau client en utilisant le bouton .



Cette fois, ça fonctionne bien ! Bravo !

3 Manipuler des documents XML

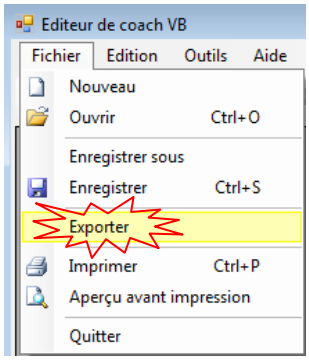
A l'instar du langage C #, Visual Basic 9 comporte de très belles nouveautés en matière de manipulation de contenu XML. Il comporte même des spécificités super sympa par rapport au langage C#. Pour une fois, on ne va pas se priver ☺…

Objectif

L'objectif de cet exercice est d'introduire les nouvelles techniques liées au langage Visual Basic introduites par la version 3.5 du Framework .NET pour manipuler des documents XML. Nous explorerons notamment la nouvelle interface de programmation XML en mémoire nommée **LINQ to XML**.

Contexte fonctionnel

Du point de vue fonctionnel, nous allons simplement rajouter une nouvelle option de menu au menu **Fichier** de notre éditeur, nommée **Export**. Comme son nom l'indique, cette option devra permettre à l'utilisateur d'exporter tout ou une partie des données clientes de la grille…évidemment au format XML ☺, dans l'optique de le transférer par exemple à une application tierce.



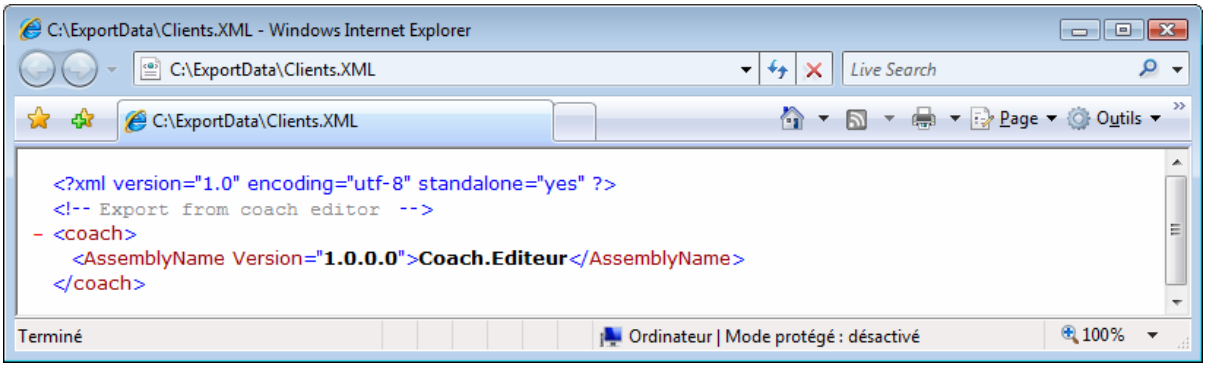
3.1 Créer un document XML

Dans cet exercice, vous allez apprendre à :

- Créer un document XML vierge,
- Utiliser les classes XDocument, XElement et XComment.

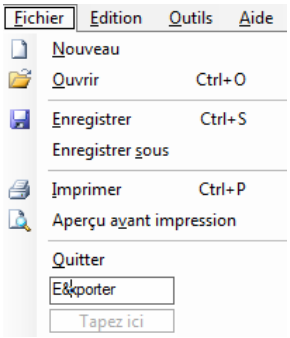
Contexte fonctionnel

Dans ce premier exercice, nous allons construire un document XML de toute pièce.
Il contiendra :
- une ligne de déclaration,
- une ligne de commentaire,
- un élément nommé <coach>
- et un sous élément nommé <AssemblyName> donnant le nom de l'assemblage .NET de notre application ainsi que sa version :

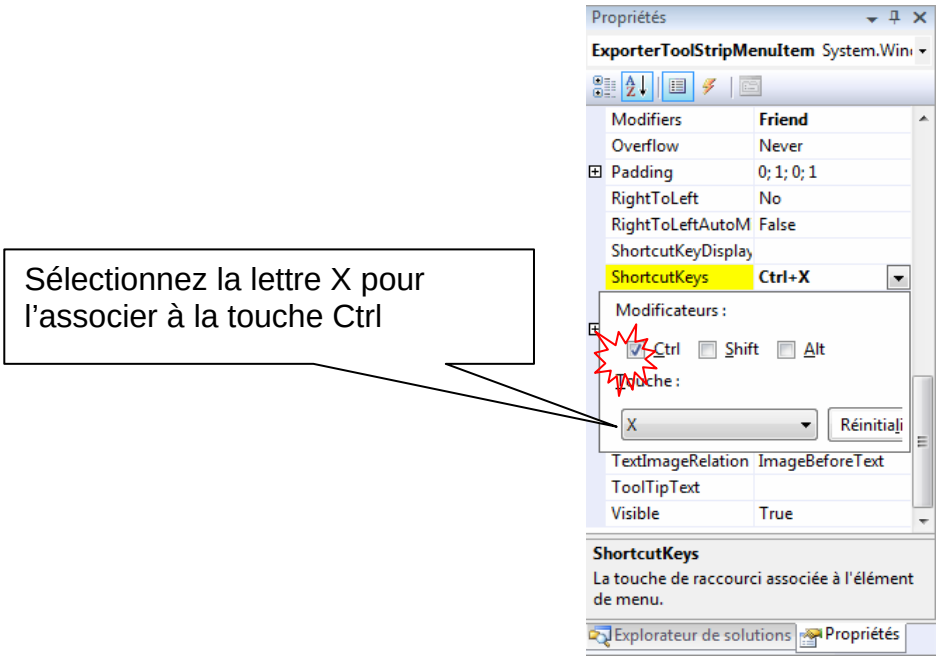


Déroulement de l'exercice :

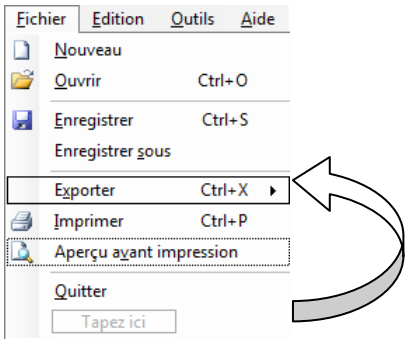
1. Rajoutez une option de menu **Fichier** > **Export** au menu de l'application :
   - Dans l'**Explorateur de solutions**, double cliquez sur le fichier **Main.vb** pour afficher le formulaire dans le **Concepteur de vue**.
   - Cliquez le menu **mainMenuStrip** puis l'option **Fichier** pour faire apparaître ses sous options.
   - Remplacez la zone **Taper ici** par **E&xporter**.



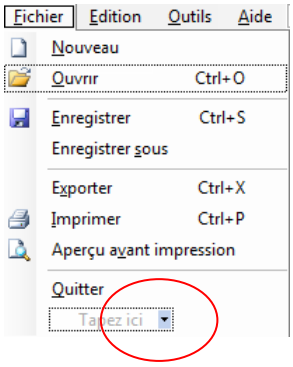
   - Affichez la fenêtre de propriétés de l'élément **ExporterToolStripMenuItem** ainsi créé en cliquant **F4**.
   - Vérifiez que la propriété **ShowShortcutKeys** est configurée à la valeur **True**.
   - Configurez la propriété **ShortcutKeys** à **Ctrl+X** en vous aidant de l'éditeur de propriétés associé qui vous montre les combinaisons de touches disponibles :



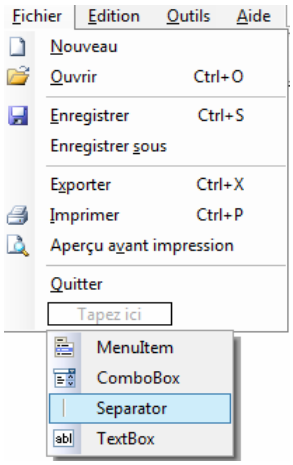
- Faites un glisser déplacer avec la souris de l'option **Exporter** pour la déplacer juste au dessus de l'option **Imprimer** du menu **Fichier**.



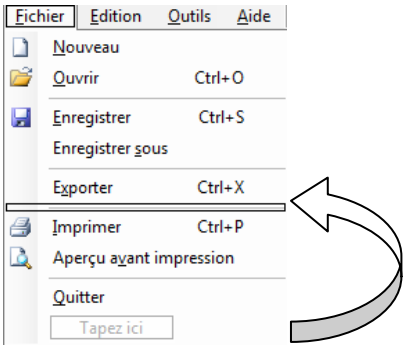
- Amenez la souris sur la zone **Tapez ici** au bas de la liste d'options du menu **Fichier** pour faire apparaître la poignée de la liste déroulante.



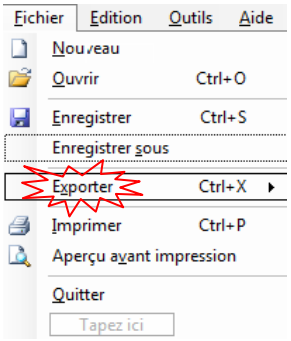
- Sélectionnez dans la liste **Separator** pour inclure un nouveau séparateur de menu :



- Faites un glisser déplacer avec la souris du séparateur pour le déplacer juste au dessus de l'option **Imprimer** du menu **Fichier**.



2. Créer un document XML vierge associé au clic du menu Fichier > Exporter :
   - A partir du **Concepteur de formulaire**, double cliquez sur l'élément **ExporterToolStripMenuItem** pour générer un gestionnaire d'évènement associé à l'évènement **Click**.



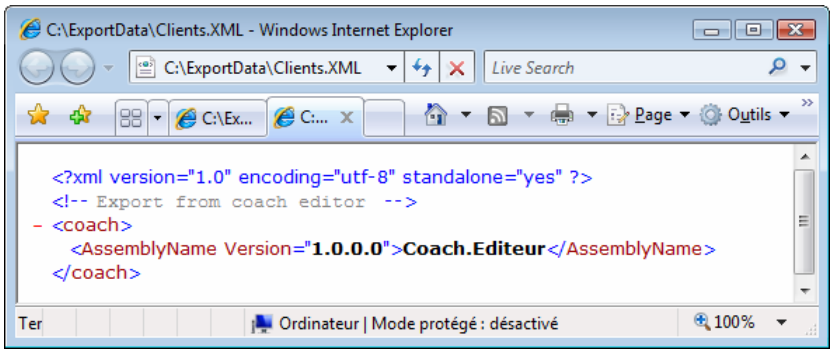
   - Visual studio bascule sur le fichier de code **Main.vb** sur la nouvelle procédure créée :

```
Private Sub ExporterToolStripMenuItem_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                    Handles ExporterToolStripMenuItem.Click

End Sub
```

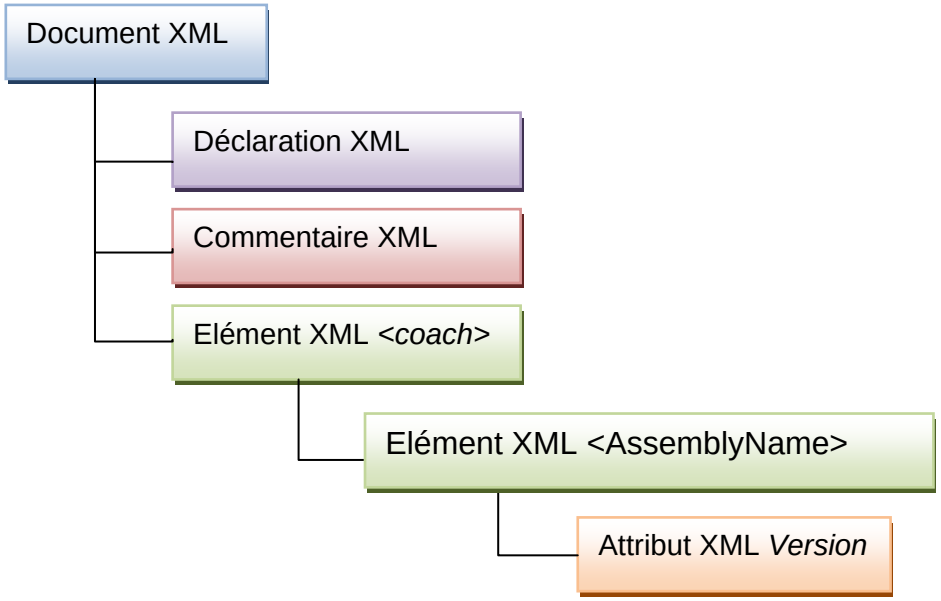
Le Framework .NET 3.5 apporte de toutes nouvelles classes pour générer des documents XML de manière très intuitive et très logique. Elles sont fournies par l'espace de noms **System.Xml.Linq** et calquent parfaitement la structure d'un document XML type.

C'est quoi la structure d'un document XML ?
Un document XML est constitué d'une arborescence de nœuds de différents types, imbriqués les uns dans les autres. Pour créer un document XML, il s'agit donc de reconstituer l'arborescence d'éléments voulue.

Voilà le document que l'on veut obtenir :



Cela revient à construire l'arborescence suivante :



Comment s'y prendre ?
Pour chaque type de nœud, le Framework.NET a prévu une classe qu'il suffit d'instancier comme n'importe quel objet mémoire pour créer un élément de type correspondant. L'arborescence, elle, se construit d'elle-même en utiliser les constructeurs de chacune des classes qui accepte les sous éléments en tant que paramètre.
En conclusion, la construction d'un tel document par code se fait en une

et une seule ligne. On y va ?

- Cette ligne, la voilà :

Code VB

```
Dim xmlDoc = New XDocument( _
        New XDeclaration("1.0", "utf-8", "yes"), _
        New XComment("Export from coach editor"), _
        New XElement("coach", _
                New XElement("AssemblyName", _
                        Assembly.GetExecutingAssembly().GetName().Name, _
                    New XAttribute("Version", _

Assembly.GetExecutingAssembly().GetName().Version))))
```

Il n'y a qu'une ligne de code mais je vous l'accorde, elle prend de la place !!!
Une bonne pratique est d'utiliser le caractère _ en fin de ligne pour réorganiser la ligne de façon à ce qu'elle reflète le plus possible l'arborescence de votre document.

Prenons le temps d'y regarder de plus près ?

- Nous cherchons à construire un document XML en mémoire, donc nous avons instancié un objet nommé **xmlDoc** de type **XDocument**. Le constructeur de la classe accepte en paramètre les éléments qui vont constituer le document.
- Le premier élément du document étant une directive de déclaration XML, le premier paramètre est donc l'instanciation d'un élément **XDeclaration**. Il attend trois paramètres qui sont tout simplement les trois attributs de l'élément :

Syntaxe du constructeur de l'élément XDeclaration extraite de MSDN :

Syntaxe

Visual Basic (Déclaration)

```
Public Sub New ( _
    version As String, _
    encoding As String, _
    standalone As String _
)
```

- Dans le même esprit, nous instancions ensuite un élément **XComment** dont le constructeur attend la chaîne de commentaire en paramètre.
- L'élément suivant est de type **XElement** et contient lui-même un autre **XElement** imbriqué qui contient à son tour un **XAttribute.** Les constructeurs de ces deux classes sont un petit peu plus complexes dans la mesure où ils permettent d'imbriquer des sous-éléments en même temps que d'initialiser la valeur de l'élément en cours.

- Notez pour terminer, que les informations de version et de nom d'assemblage de l'élément XML **<AssemblyName>** sont récupérées par le biais de l'objet **Assembly** qui fait partie de l'espace de nom **System.Reflection** du Framework .NET.

Retrouvez toutes les classes de l'espace de noms **System.Xml.Linq** ici :
http://msdn.microsoft.com/fr-fr/library/system.xml.linq.aspx
Et celles de l'espace **System.Reflection** ici :
http://msdn.microsoft.com/fr-fr/library/system.reflection.aspx

Pour voir le document XML que nous venons de construire en mémoire, il nous faudrait pouvoir le rendre persistant d'une manière ou d'une autre… Ce processus, vous le connaissez, c'est la **sérialisation** !
Rien de plus simple avec **LINQ to XML** qui a tout prévu puisqu'un élément de type **XDocument** possède une méthode **Save** qui sérialise le document dans le réceptacle voulu, par exemple dans un fichier.

3. Sérialisez le document XML mémoire dans un fichier texte nommé **Export des clients.xml** :
   - Rajoutez à la classe la définition de deux constantes **EXPORTFOLDER** et **OUTPUTFILE** correspondant respectivement au chemin du dossier de sauvegarde et au nom du fichier de destination :

Code VB

```
…
Public Class Main

   Const EXPORTFOLDER As String = "c:\ExportData\"
   Const OUTPUTFILE As String = _
                              EXPORTFOLDER & "Export des clients.XML"

   Public Sub New()
…
```

Nous vous proposons ici d'exporter le document XML dans un dossier **ExportData** situé sur **C**. Vous pouvez bien sûr effectuer l'export à n'importe quel emplacement de votre choix.
Prévoyez simplement de créer dès maintenant le dossier à partir de l'**Explorateur de Windows** si vous ne voulez pas récupérer une exception au moment de l'exécution…

- Revenez sur le gestionnaire d'évènement **ExporterToolStripMenuItem_Click**.
- Ajoutez à la suite le code de sérialisation du document **xmlDoc** à l'aide de la méthode **Save** :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                    Handles ExporterToolStripMenuItem.Click
```

```
    Dim xmlDoc = New XDocument(…)

    xmlDoc.Save(OUTPUTFILE)
End Sub
```

Pour éviter au moment des tests de basculer de Visual Studio à l'Explorateur Windows pour visualiser le fichier, une bonne pratique est d'utiliser la classe **Process** de l'espace de noms **System.Diagnostics**. Elle comprend une méthode **Start** qui démarre automatiquement le processus par défaut associé à l'extension du fichier passé en paramètre.

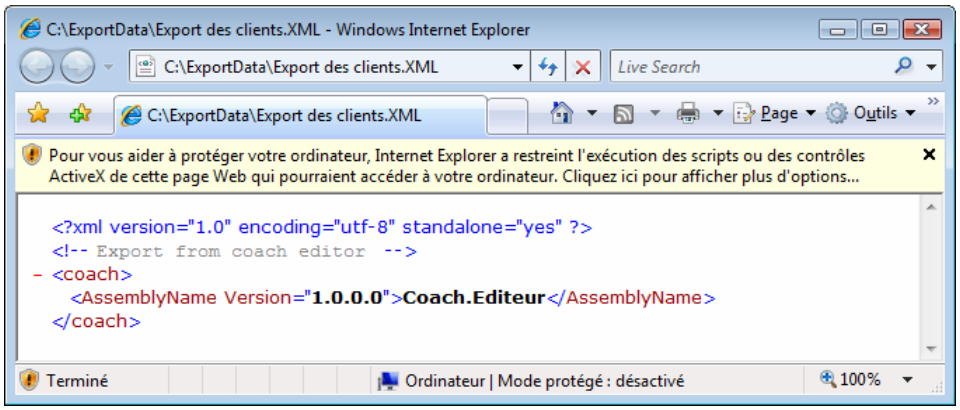
- Ajoutez à la suite le code de démarrage d'un processus de visualisation du fichier :

Code VB

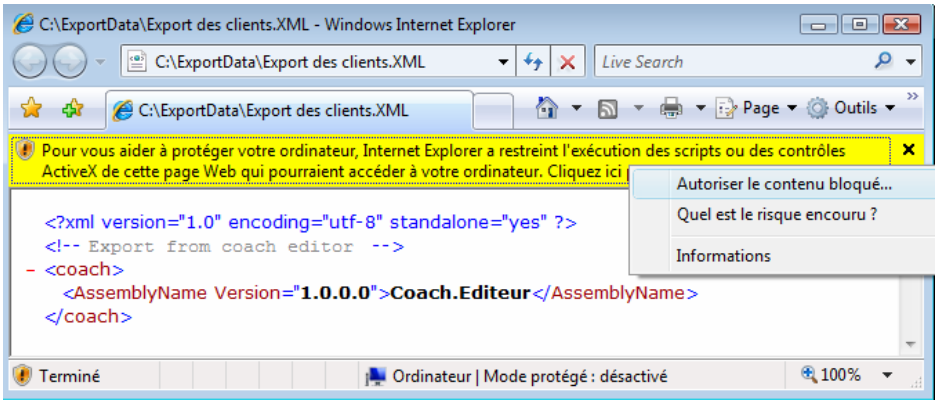
```
Private Sub ExporterToolStripMenuItem_Click(…) _
                Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc = New XDocument(…)

    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

Pour en savoir plus sur la méthode **Save** de la classe **XDocument** :
http://msdn.microsoft.com/fr-fr/library/system.xml.linq.xdocument.save.aspx
Pour en savoir plus sur la méthode **Start** de la classe **Process** :
http://msdn.microsoft.com/fr-fr/library/53ezey2s.aspx

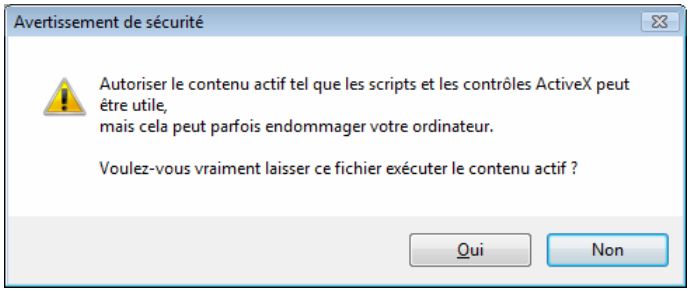
4. Testez la création du document XML :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Exporter**. Une fenêtre Internet Explorer s'ouvre avec le document résultant :



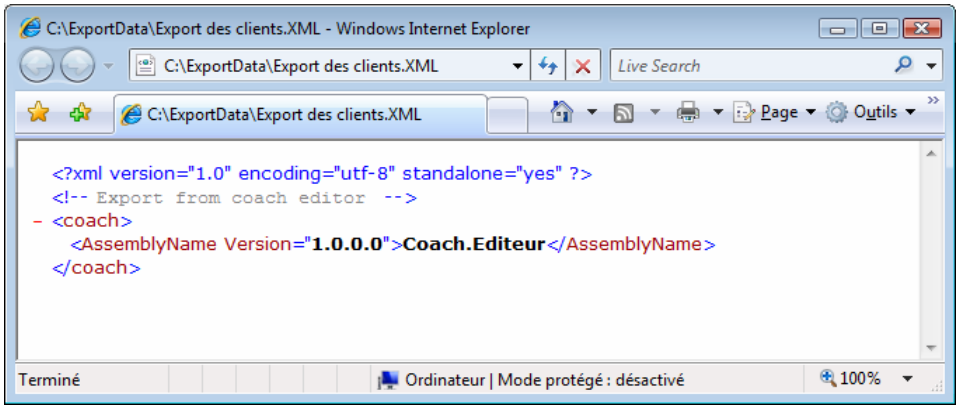
- Cliquez sur la barre qui s'affiche en haut de page (si elle apparaît) puis **Autoriser le contenu bloqué** pour activer la lecture dynamique du document XML dans Internet Explorer :



- Répondez **Oui** à la question :

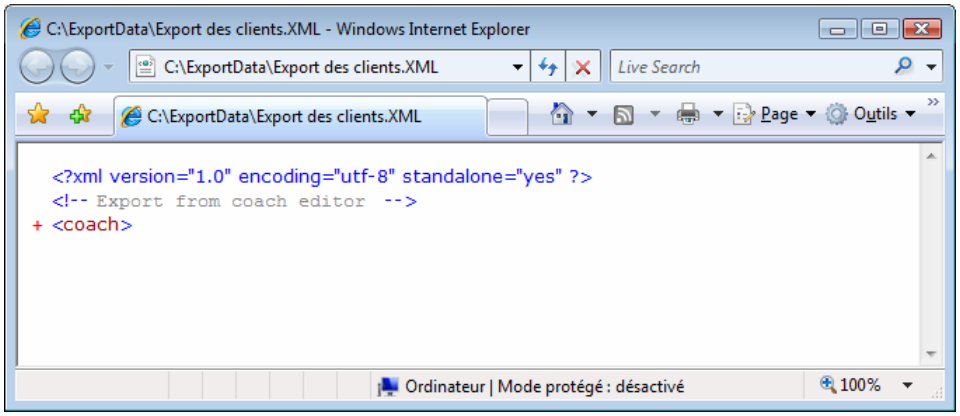


- Le document final s'affiche comme suit :



Notez :

- le nom et la version de l'assemblage retournés par les méthodes de la classe **Assembly** de l'espace de noms **System.Reflection**.
- le commentaire et la directive de déclaration sont matérialisés par une couleur spécifique par Internet Explorer qui reconnait ces éléments XML un peu particuliers.
- Le sigle ⁻ permet de réduire l'élément XML correspondant et de cacher ses enfants :



Il suffit de cliquer sur le sigle + pour réafficher l'élément XML complet.

3.2 Utiliser les littéraux XML

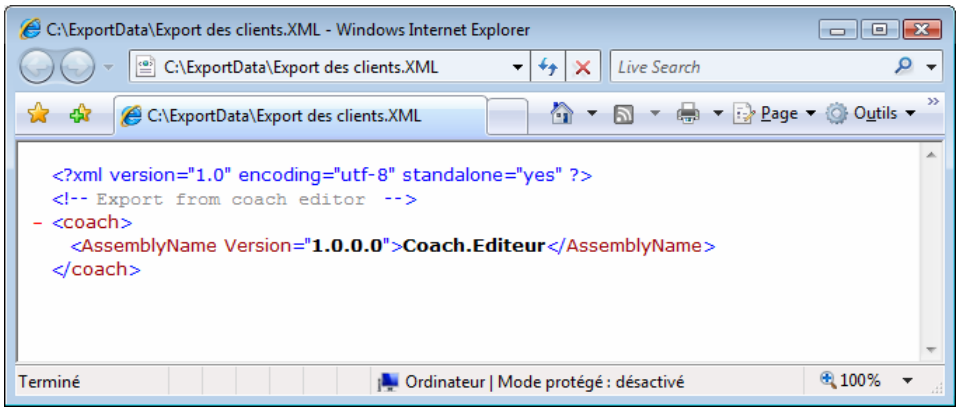
Dans cet exercice, vous allez apprendre à :

- Utiliser les littéraux XML du langage Visual Basic pour créer un document XML,
- Utiliser les expressions incorporées de Visual Basic.

Une fois n'est pas coutume, ce que nous allons voir maintenant est une spécificité du langage VB.NET que vous ne trouverez pas dans le langage C #. Il s'agit des **littéraux XML** qui permettent d'incorporer directement du XML en tant que type de données de premier ordre dans le code Visual Basic.

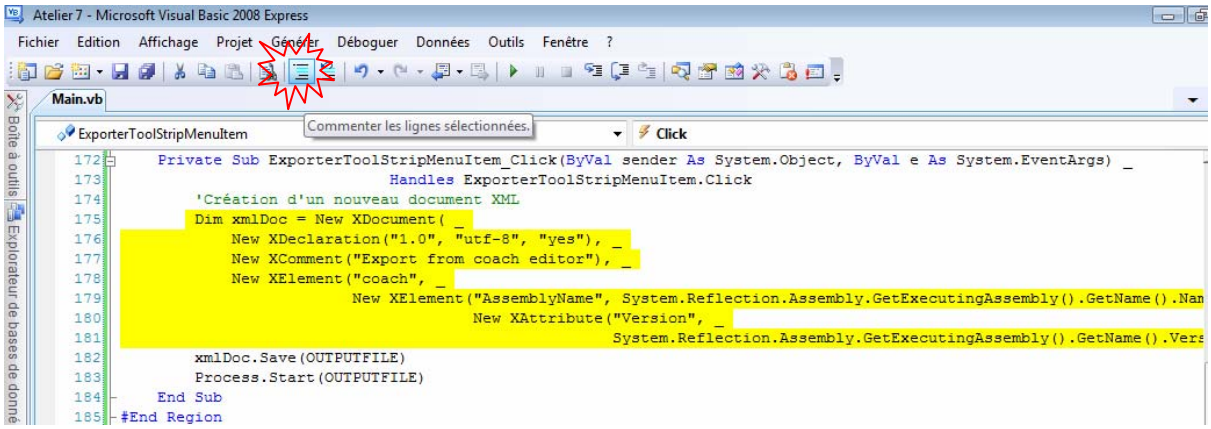
Contexte fonctionnel

Le contexte fonctionnel de cet exercice reste inchangé. La fonction d'export de l'éditeur génère toujours le même document :



1. Utilisez les littéraux XML de Visual Basic pour construire le même document XML que dans l'exercice précédent :
   - Revenez sur le gestionnaire d'évènement **ExporterToolStripMenuItem_Click**.

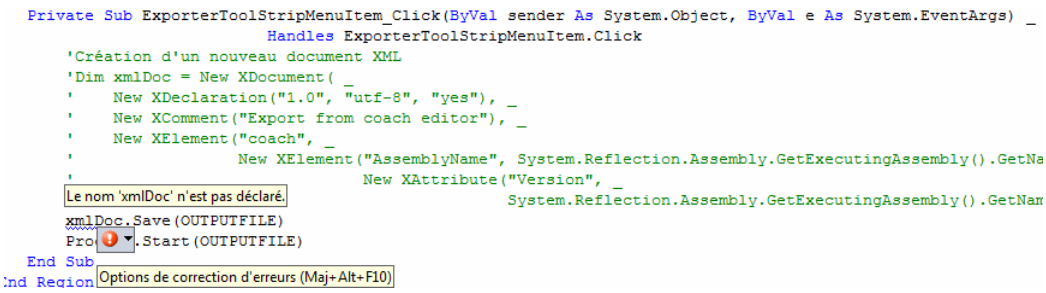
- Sélectionnez les lignes suivantes, puis cliquez sur l'icône dans la barre de menu **Standard** de **Visual Studio**. Cette action aura pour effet de commenter automatiquement les lignes sélectionnées :



- Vous obtenez :

```
Private Sub ExporterToolStripMenuItem_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                    Handles ExporterToolStripMenuItem.Click
    'Création d'un nouveau document XML
    'Dim xmlDoc = New XDocument( _
    '    New XDeclaration("1.0", "utf-8", "yes"), _
    '    New XComment("Export from coach editor"), _
    '    New XElement("coach", _
    '                New XElement("AssemblyName", System.Reflection.Assembly.GetExecutingAssembly().GetName().N
    '                             New XAttribute("Version", _
    '                                            System.Reflection.Assembly.GetExecutingAssembly().GetName().Ve
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

Notez qu'évidemment Visual Studio ne trouve plus de déclaration pour la variable **xmlDoc** et surligne donc le mot pour attirer notre attention. Si vous positionnez la souris sur le tiret rouge en fin de mot, une balise active s'affiche pour indiquer le problème potentiel :



- Ajoutez à la suite des lignes commentées une nouvelle déclaration de l'élément **xmlDoc** en utilisant les littéraux XML. Commencez à taper la ligne suivante :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                    Handles ExporterToolStripMenuItem.Click
    'Dim xmlDoc = New XDocument(…)
```

```
    Dim xmlDoc As XDocument = <?

    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

A peine avez-vous tapé le **?** que Visual Studio vous referme automatiquement la balise correspondante.

```
Dim xmlDoc As XDocument = <??>
```

- Continuez à taper la suite du document comme suit :

```
Dim xmlDoc As XDocument = <?xml version="1.0" encoding="utf-8" standalone="yes"?>
                          <!--Export from coach editor-->
                          <Coach></Coach>
```

Est-ce que vous remarquez qu'il n'y a pas besoin d'utiliser le caractère _ de fin de ligne pour indenter correctement le document XML sur plusieurs lignes ? C'est pratique, non ?

Si vous tapez la touche **Entrée** de votre clavier entre la balise de début et de fin de l'élément **coach**, l'éditeur de Visual Studio comprend que vous allez maintenant saisir un sous élément de l'élément XML en cours et effectue donc automatiquement l'indentation du code appropriée pour une meilleure lisibilité :

```
Dim xmlDoc As XDocument = <?xml version="1.0" encoding="utf-8" standalone="yes"?>
                          <!--Export from coach editor-->
                          <Coach>
                              |
                          </Coach>
```

Indentation automatique pour insérer un élément enfant

- Terminez la saisie du document :

```
Dim xmlDoc As XDocument = <?xml version="1.0" encoding="utf-8" standalone="yes"?>
                          <!--Export from coach editor-->
                          <Coach>
                              <AssemblyName>
                              </AssemblyName>
                          </Coach>
```

Ce n'est pas tout à fait terminer puisqu'il manque l'attribut **Version** de l'élément XML **AssemblyName** ainsi que le nom de l'assemblage défini en tant que valeur de l'élément de façon à obtenir :

```
<AssemblyName Version="1.0.0.0">Coach.Editeur</AssemblyName>
```

Tout le problème, c'est que ce sont des informations que nous voulons évaluées au moment de l'exécution !

Comment faire pour ajouter ces informations dynamiquement ?
Visual Basic autorise les littéraux XML à contenir des expressions évaluées au moment de l'exécution appelées **expressions incorporées**. La syntaxe pour une telle expression est **<%=*expression*%>**.

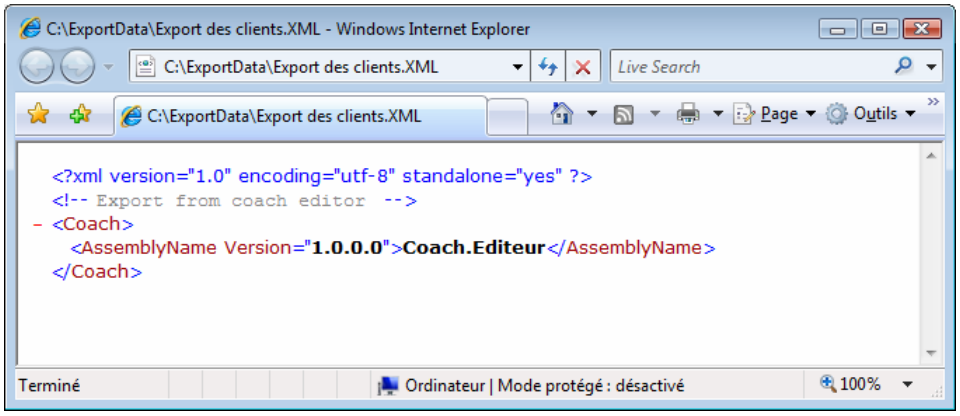
Pour tout savoir sur les **littéraux XML** du langage Visual Basic :
http://msdn.microsoft.com/fr-fr/library/bb384629.aspx
Pour en savoir plus sur les **expressions incorporées** en XML :
http://msdn.microsoft.com/fr-fr/library/bb384964.aspx

- Terminez la saisie du document en ajoutant deux expressions incorporées qui utilisent la classe **Assembly** comme dans l'exercice précédent :

```
Dim xmlDoc As XDocument = <?xml version="1.0" encoding="utf-8" standalone="yes"?>
                          <!--Export from coach editor-->
                          <Coach>
                              <AssemblyName
                                  Version=<%= System.Reflection.Assembly.GetExecutingAssembly().GetName().Version %>>
                                  <%= System.Reflection.Assembly.GetExecutingAssembly().GetName().Name %>
                              </AssemblyName>
                          </Coach>
```

N'oubliez pas la fermeture de la balise ouvrante de l'élément **AssemblyName** juste après la définition de l'attribut **Version**.
<AssemblyName Version=" "**>**…</AssemblyName>

2. Testez la création du document XML :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Exporter**. Vous devez obtenir le même document qu'à l'exercice précédent :



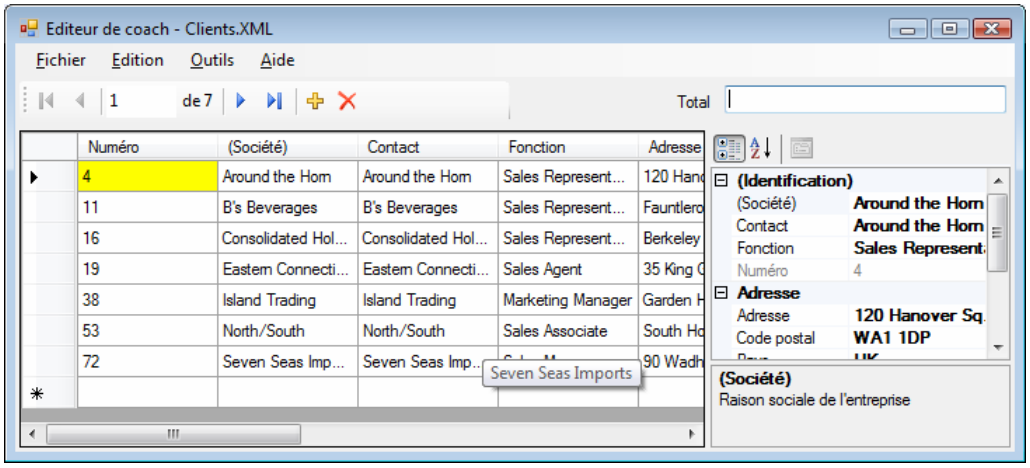
3.3 Manipuler les données

Dans cet exercice, vous allez apprendre à :

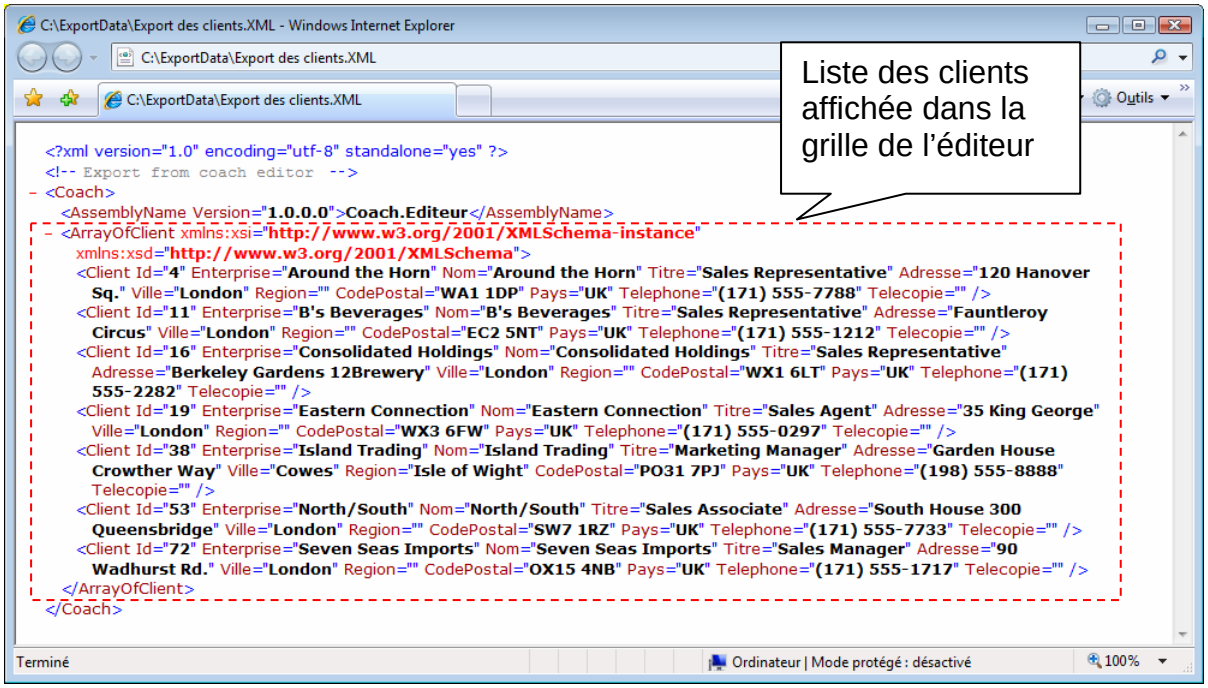
- Ajouter et supprimer des éléments XML dans une arborescence XML,
- Retrouver un élément XML dans une arborescence par son nom.

Contexte fonctionnel

Nous allons maintenant enrichir le document d'export avec des données de la liste des clients de notre éditeur.
Si la liste suivante de clients est affichée dans l'éditeur…



…alors le document d'export ressemblera à ceci :



Déroulement de l'exercice :

1. Récupérez la liste des clients affichés dans la grille de données de l'éditeur :

Pour rappel, la liste de clients affichée dans la grille dont nous avons besoin est la source de données gérée par l'objet **MainBindingSource**. En utilisant la fonction **CType**, on récupère un objet de type **Clients**.

- Toujours dans le gestionnaire d'évènement **ExporterToolStripMenuItem_Click**, ajoutez la déclaration d'une variable **clientsDataSource** et initialisez-la avec la liste des clients.

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>

    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)

    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

Pour ajouter la liste de clients à notre document XML, il faut transformer l'objet **Clients** en un objet de type **XElement**. Comment faire ?
Grâce au premier exercice de cet atelier, nous disposons d'une propriété nommée **Xml** sur notre objet **Clients**, qui renvoie une chaîne de caractères représentant l'objet sérialisé en XML. C'est exactement ce qu'il nous faut ! A partir d'une chaîne de caractères XML, il suffit d'utiliser la méthode **Parse** de la classe **XElement** pour charger un objet **XElement**.

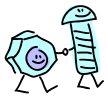
- Récupérez la chaîne XML correspondant à la sérialisation de la liste **clientsDataSource** puis utilisez la méthode **Parse** de la classe **XElement** pour instancier un nouvel objet de type **XElement** à partir de cette chaîne :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>

    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    Dim clientsXML = XElement.Parse(clientsDataSource.Xml)

    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

msdn

Pour en savoir plus sur la méthode **Parse** de la classe **XElement** :
http://msdn.microsoft.com/fr-fr/library/bb468714.aspx



Pour rappel, **Parse** est une *méthode partagée* de la classe **XElement** c'est-à-dire qu'il n'est pas nécessaire de définir une instance de la classe **XElement** pour l'utiliser.

Notez qu'il serait peut-être judicieux de prévoir le cas où aucun client n'est affiché dans la liste de l'éditeur, pour éviter de manipuler un document vide.

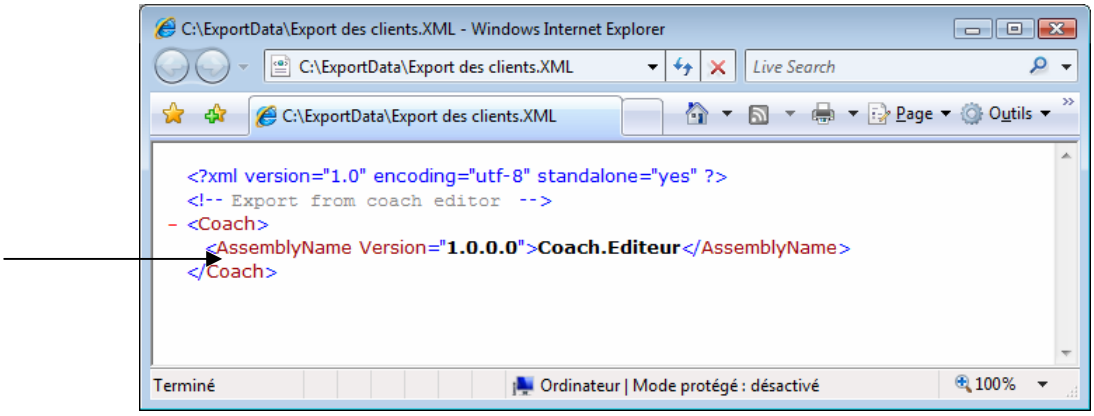
- Ajoutez un test pour vérifier que la grille contient bien des données avant de poursuivre le traitement :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>

    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
       Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
    Else
       'Aucun client affiché dans l'éditeur
    End If

    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

2. Ajoutez la liste des clients au format XML dans le nœud <coach> du document XML d'export :

Il s'agit maintenant d'insérer les nœuds de notre liste de clients XML à l'intérieur de la hiérarchie de nœuds de notre document d'export. Nous vous proposons de l'insérer par exemple en tant que sous élément de l'élément **<coach>** au même niveau que l'élément **<AssemblyName>** :



La classe **XElement** possède des méthodes **Add** et **Remove** qui permettent respectivement d'ajouter et de supprimer des éléments en tant

qu'enfants d'un élément XML. Pour obtenir l'élément **<coach>** dans lequel nous voulons accrocher la liste des clients, LINQ to XML fournit une méthode **Element** qui attend en paramètre le nom de l'élément à trouver.

- Ajoutez la ligne suivante pour extraire l'élément **<coach>** du document **xmlDoc** à l'aide de la méthode **Element** :

Code VB

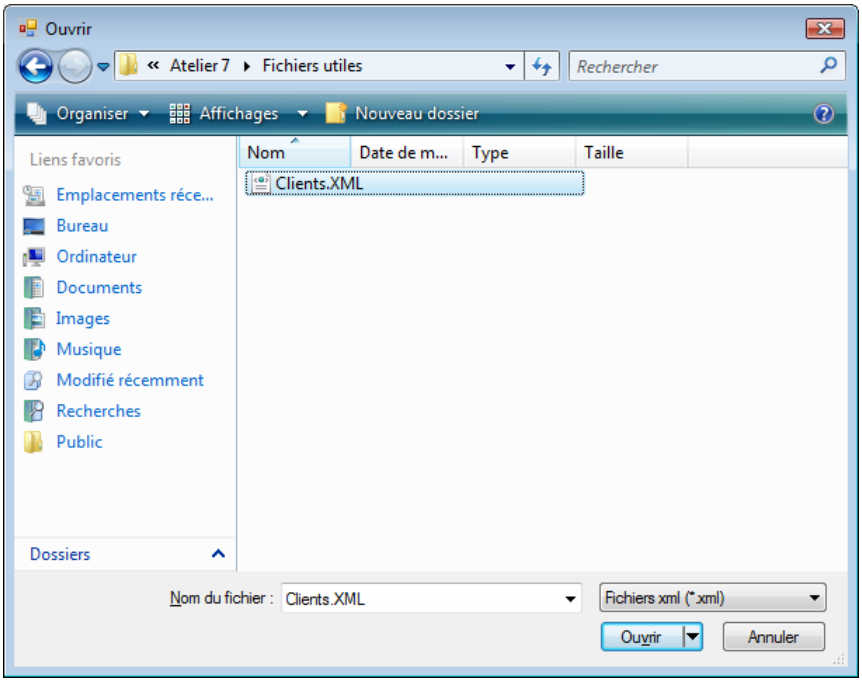
```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
       Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
       Dim coachElement = xmlDoc.Element("Coach")
    Else
       'Aucun client affiché dans l'éditeur
    End If
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

- Utilisez la méthode **Add** pour ajouter la liste des clients **clientsXML** en tant qu'enfant de l'élément **<coach>** :

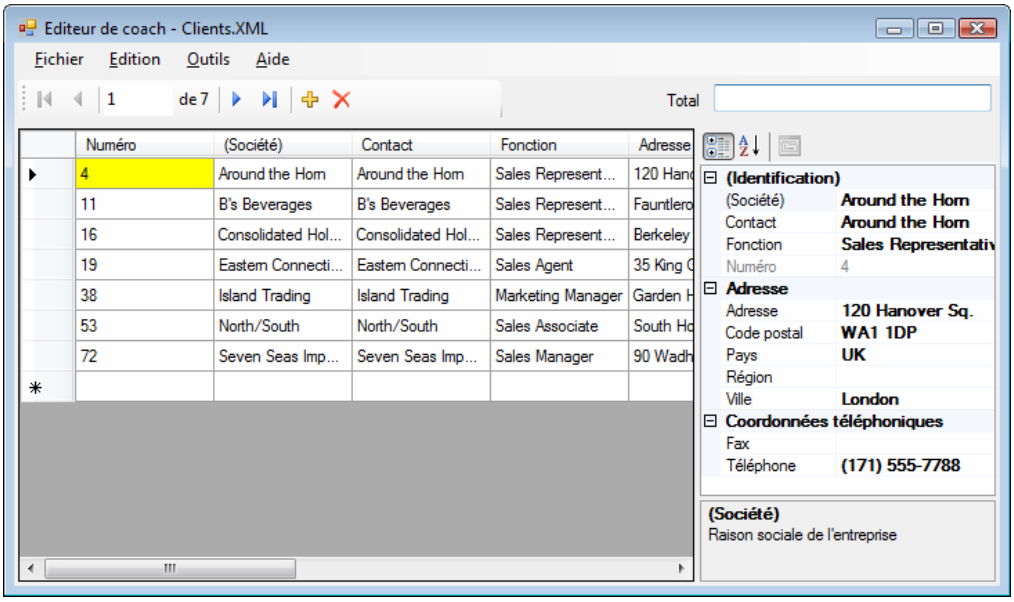
Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
       Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
       Dim coachElement = xmlDoc.Element("Coach")
       coachElement.Add(clientsXML)
    Else
       'Aucun client affiché dans l'éditeur
    End If
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

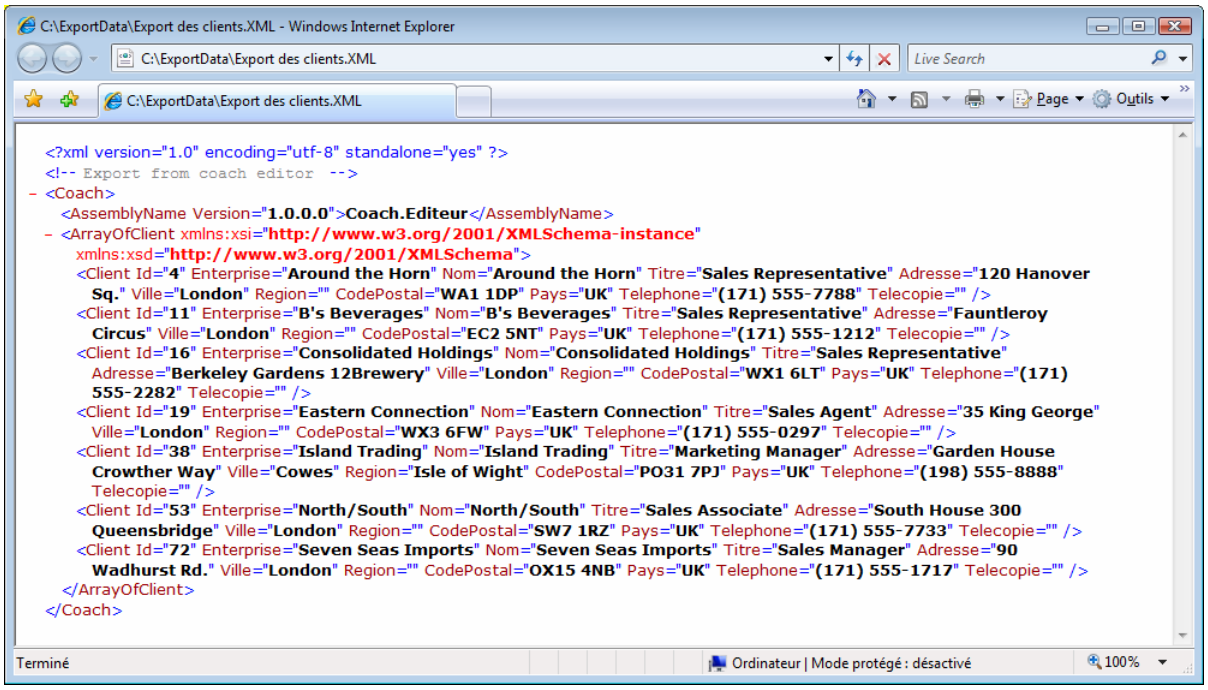
3. Testez le contenu du document exporté :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir**.
   - Sélectionnez le fichier d'exemple **Clients.XML** fourni avec le code de cet atelier dans le répertoire **\Atelier 7\Fichiers utiles** :



- Cliquez **Ouvrir** pour afficher la liste de clients dans l'éditeur :



- Cliquez le menu **Fichier** > **Exporter**. Une fenêtre Internet Explorer s'ouvre avec le document résultant.
- Autoriser le contenu bloqué si cela vous est demandé. Vous obtenez :



3.4 Utiliser les propriétés d'axe XML

Dans cet exercice, vous allez apprendre à :

- Utiliser les propriétés d'axe XML du langage Visual Basic.

Contexte fonctionnel

Le contexte fonctionnel est rigoureusement le même que pour l'exercice précédent.

Une fois de plus, Visual Basic nous propose une petite spécificité de langage non négligeable pour nous aider à accéder encore plus facilement que ce que nous venons de voir à un élément XML.
Cette fonctionnalité s'appelle les *propriétés d'axe XML*.

Par exemple, pour accéder aux éléments enfants d'un élément XML, il suffit d'utiliser un point suivi du nom de l'élément enfant recherché entre crochets.

- Ainsi pour rechercher tous les éléments **<Coach>** du document **xmlDoc**, Visual Basic propose la syntaxe suivante :
  xmlDoc.<Coach> (au lieu de xmlDoc.Element("Coach"))
- Pour rechercher tous les éléments **<Coach>** n'importe où dans la hiérarchie du document, on a l'écriture :
  xmlDoc…<Coach>
- Ou encore pour retrouver l'attribut **Titre** d'un élément **<Client>** on aurait :
  xmlClient.@Titre

Pour tout savoir sur les propriétés d'axe XML du langage Visual Basic:
http://msdn.microsoft.com/fr-fr/library/bb384769.aspx

Déroulement de l'exercice :

1. Remplacez le code d'accès à l'élément <Coach> vu dans l'exercice précédent par un code utilisant les propriétés d'axe XML :
   - Dans le gestionnaire **ExporterToolStripMenuItem_Click**, supprimez la ligne suivante :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
        Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
        ~~Dim coachElement = xmlDoc.Element("Coach")~~
        coachElement.Add(clientsXML)
    Else
        'Aucun client affiché dans l'éditeur
    End If
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

   - Remplacez-la par la ligne suivante :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
        Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
        Dim coachElement = xmlDoc.<Coach>
        coachElement.Add(clientsXML)
    Else
        'Aucun client affiché dans l'éditeur
    End If
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

Attention ! Quelle est le résultat renvoyé par la propriété d'axe enfant XML que nous avons utilisée ?
Il s'agit en fait d'une collection d'objets **XElement** et non d'un élément unique…D'ailleurs Visual Studio râle un peu sur la ligne de code réalisant

l'ajout de la liste de clients à notre élément **coachElement** en surlignant l'instruction en bleu. Si vous positionnez la souris sur la ligne bleu, vous pouvez lire un commentaire vous expliquant que la méthode **Add** ne s'applique pas sur une collection d'éléments **XElement** :

```
Dim coachElement = xmlDoc.<Coach>
coachElement.Add(clientsXML)
'Add' n'est pas un membre de 'System.Collections.Generic.IEnumerable(Of System.Xml.Linq.XElement)'.
```

Or nous savons qu'il n'y a qu'un seul élément **<Coach>** dans notre document **xmlDoc**. Donc nous pouvons sans hésiter ne récupérer que le premier (et unique) élément de la collection.

- Utilisez la méthode **First** pour ne renvoyer que le premier élément de la collection :

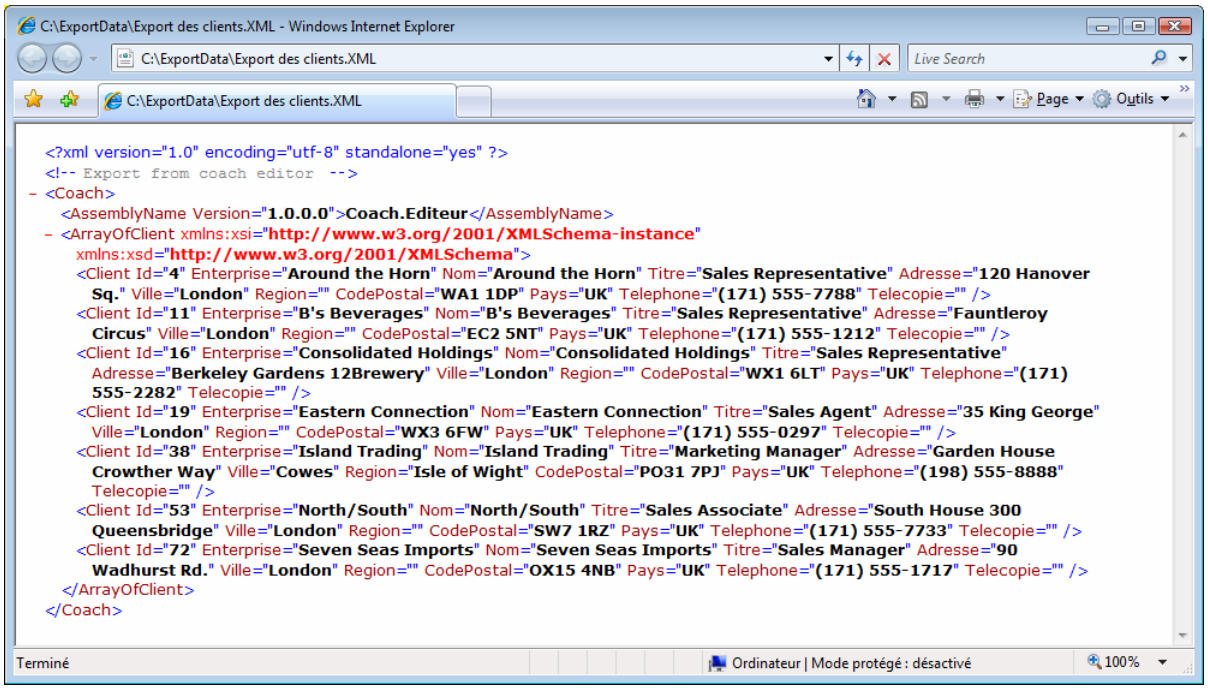
Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
        Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
        Dim coachElement = xmlDoc.<Coach>.First
        coachElement.Add(clientsXML)
    Else
        'Aucun client affiché dans l'éditeur
    End If
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

Visual Studio ne râle plus ☺ :

```
Dim coachElement = xmlDoc.<Coach>.First
coachElement.Add(clientsXML)
```

2. Vérifiez que la réécriture du code n'a pas modifié le contenu du document à l'export :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir**.
   - Sélectionnez le fichier d'exemple **Clients.XML** fourni avec le code de cet atelier dans le répertoire **\Atelier 7\Fichiers utiles**.
   - Cliquez **Ouvrir** pour afficher la liste de clients dans l'éditeur.
   - Cliquez le menu **Fichier** > **Exporter**. Vous devez obtenir toujours le même résultat :



3.5 Rechercher des données

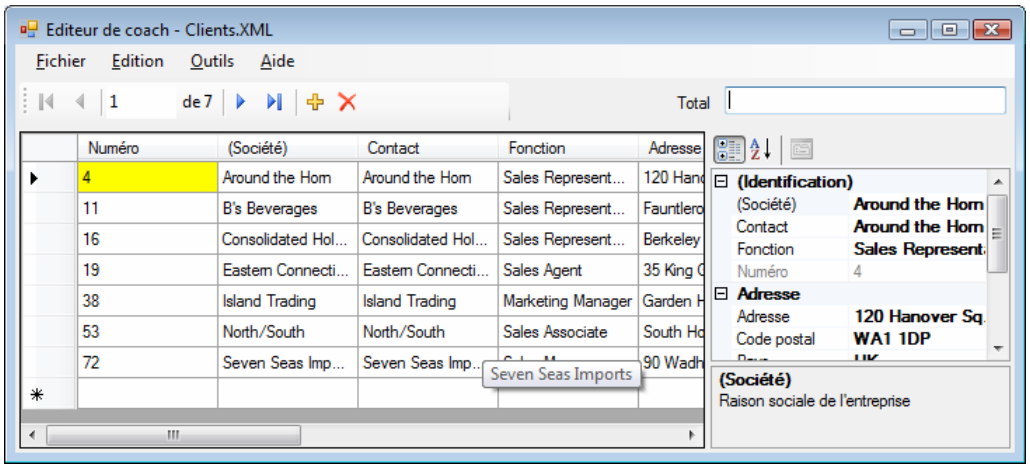
Dans cet exercice, vous allez apprendre à :

- Ecrire et exécuter une requête LINQ.

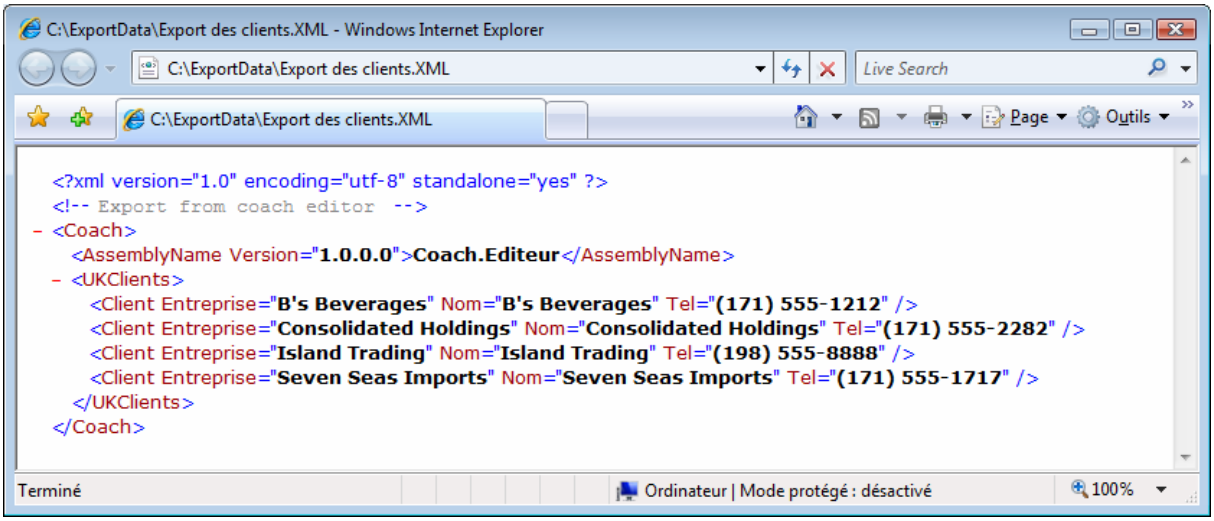
Contexte fonctionnel

Plutôt que d'exporter l'intégralité de la liste des clients, nous n'allons exporter qu'une partie de cette liste. Par exemple, nous n'allons récupérer que les informations **Entreprise**, **Nom** et **Téléphone** pour tous les clients d'un pays donné.

Si la liste suivante de clients est affichée dans l'éditeur…



…alors le document d'export ressemblera à ceci :



Déroulement de l'exercice :

1. Créez une requête LINQ pour extraire tous les éléments <Client> de l'objet XElement clientsXML contenant notre liste de clients :

Comment faire pour retrouver un élément donné dans un document XML ?
Une première solution consiste à utiliser l'espace de noms **System.Xml.XPath** qui propose des méthodes pour naviguer dans un document XML à l'aide du langage **XPath**. Mais cette méthode vous obligerait à vous plonger dans l'apprentissage d'un nouveau langage…pas cool donc !

msdn

Pour tout savoir sur l'espace de nom **System.Xml.XPath** :
http://msdn.microsoft.com/fr-fr/library/system.xml.xpath.aspx
Bibliothèque de référence du langage **XPath** :
http://msdn.microsoft.com/fr-fr/library/ms256115.aspx

Avec Visual Studio 2008 et le Framework 3.5, nous disposons maintenant de la technologie **LINQ** qui apporte justement des fonctions de requête puissantes à la syntaxe du langage Visual Basic lui-même (et C#). En clair, pas besoin d'apprendre un nième langage pour faire des requêtes sur la source de données. LINQ gère d'ailleurs plusieurs fournisseurs pour vous permettre d'attaquer plusieurs types de sources de données. Dans notre cas, la source de données est un document XML.
Dans cet exercice nous allons voir que LINQ to XML s'appuie sur la technologie LINQ pour nous permettre d'interroger nos arborescences XML.

msdn

Pour en savoir plus sur la technologie **LINQ** :
http://msdn.microsoft.com/fr-fr/library/bb397926.aspx

Pour rechercher des éléments enfants dont on connait le nom, on dispose d'une méthode nommée **Descendants**. Celle-ci retourne en réalité une collection (et non un seul) d'éléments descendants pour le document XML.

Vous trouverez à cette adresse des procédures pour effectuer des requêtes de base avec **LINQ to XML** :
http://msdn.microsoft.com/fr-fr/library/bb943906.aspx

- Reprenez le code du gestionnaire **ExporterToolStripMenuItem_Click**.
- A la suite de la déclaration de l'élément **coachElement**, ajoutez une requête qui recherche tous les éléments descendants **<Client>** de l'objet **clientsXML** de type **XElement** :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
        Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
        Dim coachElement = xmlDoc.<Coach>.First

        Dim q = From element In clientsXML.Descendants("Client") _
                Select element

        coachElement.Add(clientsXML)
    Else
        'Aucun client affiché dans l'éditeur
    End If
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

Explorons en détail la requête LINQ que nous venons d'écrire :
La première étape d'une requête LINQ consiste à spécifier la *source de données* et la *variable* utilisée pour faire *référence* à un élément de la source. C'est le rôle de la clause **From**. La variable de référence est appelée *variable de portée*.

Qu'elle est la *source de données* que nous interrogeons ?
Notre objectif est de retrouver tous les éléments **<Clients>** dans l'élément **clientsXML**. Pour cela nous utilisons la méthode **Descendants** qui renvoie tous les descendants potentiels portant le nom passé en paramètres. La source de données est donc le résultat de l'expression :

clientsXML.Descendants("Client")

Qu'est ce qu'une *variable de portée* ?
La variable **element** immédiatement déclarée après la clause **From** est

appelée *variable de portée*. Elle sert à référencer chaque élément consécutif de la source de données. Il n'est pas nécessaire de préciser son type puisque le compilateur le déduit automatiquement de la source de données indiquée. Positionnez la souris sur le mot **element** pour vérifier qu'il s'agit bien d'un élément XML de type **XElement** :

```
Dim q = From element In clientsXML.Descendants("Client") _
        Selec
```
element As System.Xml.Linq.XElement

On pourrait comparer la variable de portée à une variable d'itération dans une boucle **For Each…Next**. Mais attention, aucune itération ne se produit réellement dans l'expression de requête que nous avons écrite.

D'ailleurs il faut comprendre que la requête n'est tout simplement pas exécutée au moment où nous l'écrivons. En réalité, nous n'avons fait que préparer une requête en vue de son *exécution différée*. Imaginez comme c'est pratique et puissant comme mécanisme ! LINQ n'exécutera notre requête que quand nous le déciderons, plus précisément quand nous commencerons à utiliser la requête, à savoir la variable que nous avons nommée **q** (pour **q**uery signifiant requête en anglais).

La deuxième étape de notre requête simple est de produire le résultat de la requête en indiquant la forme et le type de chaque élément retourné. C'est l'objet de la clause **Select** qui permet de projeter le résultat de la sélection dans la source de données. On parle de *projection* car la clause **Select** vous permet de produire toute autre chose qu'une copie de l'élément source. Dans notre cas, la variable de portée indique que le résultat de la requête sera de type **IEnumerable(Of T)** ou **T** est le type de **element**. Positionnez la souris sur la variable **q** pour vérifier qu'il s'agit bien d'un élément de type **IEnumerable(Of XElement) :**

```
Dim q = From element In clientsXML.Descendants("Client") _
```
Dim q As System.Collections.Generic.IEnumerable(Of System.Xml.Linq.XElement)

Cette requête nous renverra donc tous les clients de la liste. Or nous ne voudrions que les clients basés par exemple en Angleterre c'est-à-dire ceux dont l'information pays est « UK ».

Pour rappel, l'information du pays est accrochée en tant qu'attribut de l'élément XML **<Client>** :

```
<Client Id="16" Enterprise="Consolidated Holdings" Nom="Consolidated Holdings" Titre="Sales Representative"
  Adresse="Berkeley Gardens 12Brewery" Ville="London" Region="" CodePostal="WX1 6LT" Pays="UK" Telephone="(171)
  555-2282" Telecopie="" />
```

Nous allons donc modifier légèrement notre requête pour y ajouter une condition de filtrage. C'est là qu'entre en jeu une troisième clause, nommée **Where** du langage Visual Basic.

2. Modifiez la requête LINQ pour n'extraire que les éléments <Client> dont l'attribut Pays à la valeur « UK » :
   - Ajoutez une clause **Where** entre la clause **From** et la clause **Select** comme suit :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
       Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
       Dim coachElement = xmlDoc.<Coach>.First

       Dim q = From element In clientsXML.Descendants("Client") _
               Where (element.@Pays = "UK") _
               Select element

       coachElement.Add(clientsXML)
    Else
       'Aucun client affiché dans l'éditeur
    End If
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

L'expression de condition renvoie une valeur booléenne **True** ou **False**. Notez qu'on utilise la variable de portée **element** et une propriété d'axe d'attribut XML (**@Pays**) pour établir la condition d'évaluation de chaque élément de la source.
Si la condition prend la valeur **True**, l'élément est inclus dans le résultat de la requête, sinon il en est exclu.

- Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.

Bravo ! Cette fois nous avons extrait les éléments **<Client>** corrects à exporter.
Mais il reste encore une petite chose à faire. En effet, nous ne voulons pas nous charger avec l'ensemble des attributs de l'élément **<Client>**. Nous ne voudrions récupérer que les informations **Entreprise**, **Nom** et **Telephone**.

Comment faire ?
En fait, cela reviendrait à reconstruire un nouveau jeu d'éléments XML <Client> au format demandé. Construire un élément XML ? Ca tombe bien, nous savons faire. Il suffit d'utiliser les classes **XElement** et **XAttribute** de LINQ to XML.
Nous allons donc combiner tout ce que nous avons vu jusqu'à maintenant

pour construire un élément XML tout neuf à partir des éléments <Client> extraits au moyen de notre requête.

- Modifiez la ligne de requête comme suit :

Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                    Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
       Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
       Dim coachElement = xmlDoc.<Coach>.First

       Dim exportClients = _
          <UKClients>
             <%= From element In clientsXML.Descendants("Client") _
                Where element.@Pays = "UK" _
                Select _
                <Client
                   Entreprise=<%= element.@Enterprise %>
                   Nom=<%= element.@Nom %>
                   Tel=<%= element.@Telephone %>>
                </Client> %>
          </UKClients>

       coachElement.Add(clientsXML)
    Else
       'Aucun client affiché dans l'éditeur
    End If
    xmlDoc.Save(OUTPUTFILE)
    Process.Start(OUTPUTFILE)
End Sub
```

Prenons un peu de recul sur cette ligne de code et vous allez voir que c'est beaucoup plus simple qu'il n'y parait ☺ :

- Est-ce que vous reconnaissez la structure de notre document cible écrite en utilisant les littéraux XML ? Elle est de la forme :

```
Dim exportClients = _
    <UKClients>
        <Client
            Entreprise=""
            Nom=""
            Tel="">
        </Client>
    </UKClients>
```

D'ailleurs de quel type va être la variable **exportClients** ?
Nous sommes en train de créer un objet de type **XElement** !

```
Dim exportClients = _
    Dim exportClients As System.Xml.Linq.XElement
```

- Ensuite on utilise les expressions incorporées pour ajouter dynamiquement les informations correspondant à l'élément courant. Elles sont du type **<%=*expression*%>**. C'est là que c'est génial !! LINQ nous autorise à utiliser la requête (donc à l'exécuter) directement au bon milieu de notre construction d'objet **XElement**.

```
Dim exportClients = _
    <UKClients>
        <%= From element In clientsXML.Descendants("Client") _
            Where element.@Pays = "UK" _
            Select _
```

- N'oubliez pas que c'est la clause **Select** qui définit la forme et le type du résultat de la requête :

```
Select _
 <Client
     Entreprise=""
     Nom=""
     Tel=""
 </Client> %>
```

- Enfin vous reconnaissez les propriétés d'axe d'attribut XML qui nous permettent de construire dynamiquement les nouveaux attributs de l'élément **<Client>** résultant.

```
<Client
    Entreprise=<%= element.@Enterprise %>
    Nom=<%= element.@Nom %>
    Tel=<%= element.@Telephone %>>
</Client> %>
```

- Il ne nous reste plus qu'à modifier la ligne de code qui ajoute le jeu de **<Client>** au document XML d'export pour qu'il insère notre élément **exportClients** (plutôt que **clientsXML** comme précédemment) :

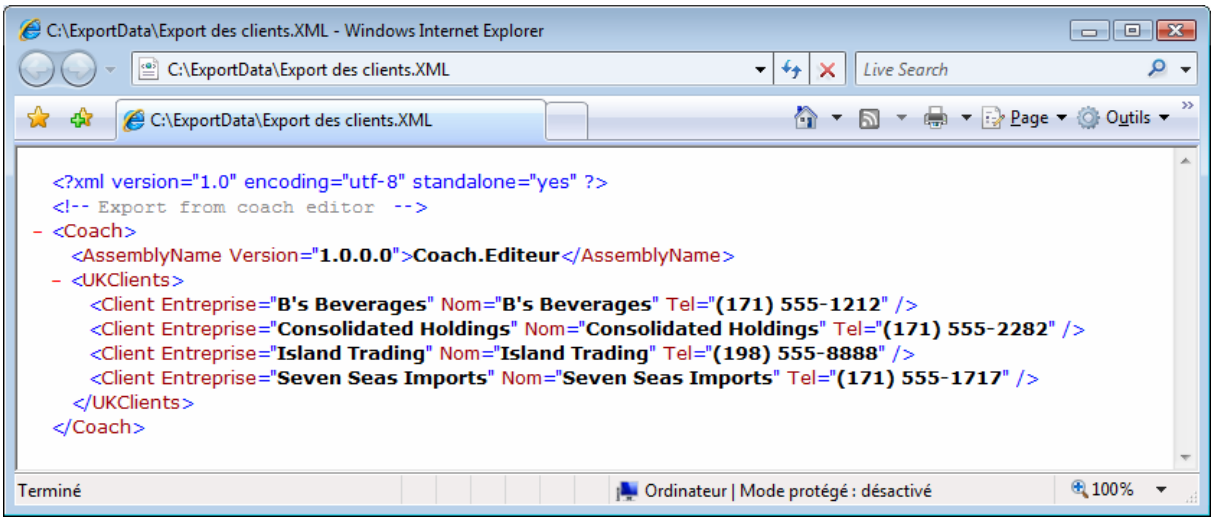
Code VB

```
Private Sub ExporterToolStripMenuItem_Click(…) _
                                Handles ExporterToolStripMenuItem.Click
    Dim xmlDoc As XDocument = <?xml …>
    Dim clientsDataSource = CType(MainBindingSource.DataSource, Clients)
    If Not clientsDataSource Is Nothing Then
        Dim clientsXML = XElement.Parse(clientsDataSource.Xml)
        Dim coachElement = xmlDoc.<Coach>.First

        Dim exportClients = _
```

```
            <UKClients>
               <%= From element In clientsXML.Descendants("Client") _
                  Where element.@Pays = "UK" _
                  Select _
                  <Client
                     Entreprise=<%= element.@Enterprise %>
                     Nom=<%= element.@Nom %>
                     Tel=<%= element.@Telephone %>>
                  </Client> %>
            </UKClients>

         coachElement.Add(exportClients)
      Else
         'Aucun client affiché dans l'éditeur
      End If
      xmlDoc.Save(OUTPUTFILE)
      Process.Start(OUTPUTFILE)
End Sub
```

Pour en savoir plus sur les requêtes LINQ :
http://msdn.microsoft.com/fr-fr/library/bb397906.aspx
Pour tout savoir sur les clauses
- **Select** : http://msdn.microsoft.com/fr-fr/library/bb384607.aspx
- **Where** : http://msdn.microsoft.com/fr-fr/library/bb384663.aspx
- et **From** : http://msdn.microsoft.com/fr-fr/library/bb384802.aspx

3. Testez le contenu du document exporté :
   - Enregistrez tous vos changements en cliquant sur dans la barre d'outils de Visual Studio.
   - Exécutez l'application en cliquant sur (ou touche F5).
   - Cliquez le menu **Fichier** > **Ouvrir**.
   - Sélectionnez le fichier d'exemple **Clients.XML** fourni avec le code de cet atelier dans le répertoire **\Atelier 7\Fichiers utiles**.
   - Cliquez **Ouvrir** pour afficher la liste de clients dans l'éditeur.
   - Cliquez le menu **Fichier** > **Exporter**. Vous obtenez la liste des clients anglais comme suit :


C:\ExportData\Export des clients.XML - Windows Internet Explorer
C:\ExportData\Export des clients.XML
Live Search
C:\ExportData\Export des clients.XML
Page
Outils
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<!-- Export from coach editor -->
<Coach>
<AssemblyName Version="1.0.0.0">Coach.Editeur</AssemblyName>
<UKClients>
<Client Entreprise="B's Beverages" Nom="B's Beverages" Tel="(171) 555-1212" />
<Client Entreprise="Consolidated Holdings" Nom="Consolidated Holdings" Tel="(171) 555-2282" />
<Client Entreprise="Island Trading" Nom="Island Trading" Tel="(198) 555-8888" />
<Client Entreprise="Seven Seas Imports" Nom="Seven Seas Imports" Tel="(171) 555-1717" />
</UKClients>
</Coach>
Terminé
Ordinateur | Mode protégé : désactivé
100%

4 Pour aller plus loin…

Voici quelques références pour aller toujours plus loin avec Visual Basic :

- Le guide de programmation Visual Basic comprend une rubrique dédiée à l'utilisation de LINQ en Visual Basic : http://msdn.microsoft.com/fr-fr/library/bb385100.aspx
- Le site **Microsoft Audio & Video** propose de nombreux webcasts sur le thème de LINQ to XML : http://www.microsoft.com/France/Vision/List.aspx?Qry=Linq+to+XML
- Et enfin, pensez au site **CodePlex** de la communauté Microsoft sur lequel vous trouverez de nombreux projets en open source et donc des exemples de code. En voici un : http://www.codeplex.com/net2csgenxlinq