

C#

**Premiers pas avec les
WinForms**

Table des matières

<i>Introduction</i>	4
Qu'est ce qu'une WinForm ?	4
Objectif	4
Minimum requis	4
A propos de la portabilité	4
<i>Chapitre 1</i>	5
<i>Premiers pas</i>	5
Application	5
<i>Chapitre 2</i>	9
<i>Une interface plus riche</i>	9
Application	9
Première méthode : A la bourrin !	10
Seconde méthode : Réfléchissez un peu !	11
Qu'est ce qu'une ancre ?	13
Fin de la partie Design	13
Evènements	14
Retour sur les ancres	15
<i>Chapitre 3</i>	16
<i>Interfaces multifenêtres</i>	16
Un cas simple	16
Interface à documents multiples	20
Retour sur la première application	22
<i>Chapitre 4</i>	25
<i>Les contrôles personnalisés</i>	25
Application	25
La base de données	26
User Control	28
Evènements personnalisés	30
La suite...	32
Les attributs	36
Fin	43
Pour aller plus loin...	46

<i>Chapitre 5.....</i>	<i>48</i>
<i>Multifenêtres, suite... ..</i>	<i>48</i>
Application.....	48
Premier cas.....	51
Second cas	52
<i>Chapitre 6.....</i>	<i>57</i>
<i>A vous de jouer.....</i>	<i>57</i>
Help me !	57
Exemple : La TreeView	57
Dernières informations	58
A vous de jouer	59
Exercice 1 : Solo Messenger.....	59
Exercice 2 : Jouez	59
Exercice 3 : Additionneur	60
Exercice 4 : Friends	60
Exercice 5 : Notepad.....	60
Exercice 6 : MyWeb	61
Exercice 7 : QCM	61
Exercice 8 : Faîtes le tri	62
Exercice 9 : PictureViewer	62
Exercice 10 : Notepad (suite)	62
Exercice 11 : Calculatrice	62
Exercice 12 : Explorateur	63

Introduction

Qu'est ce qu'une WinForm ?

Les Windows, Windows Forms, Forms ou encore SWF¹ sont la base de toute application Windows digne de ce nom. Elles sont le support graphique sur lequel vous, développeur, allez placer vos contrôles (boutons, menus, listes, etc.) permettant à l'utilisateur d'interagir facilement avec votre application.

Objectif

Vous verrez ici comment créer des interfaces graphiques (ou GUI²). Ce tutorial est destiné aux débutants qui n'ont jamais ou très peu utilisé les WinForms. Cependant, les derniers chapitres peuvent intéresser des développeurs d'un niveau moyen.

La pédagogie utilisée ici se base essentiellement sur la pratique et l'utilisation des capacités des EDI³ tels que Visual C# ou SharpDevelopp.

Cela peut sembler paradoxal, mais il n'est pas nécessaire de s'y connaître en C# pour commencer ce tutorial. Il suffit d'avoir les connaissances de bases dans un langage de type (C/C++, Java, C# bien sûr, etc.).

Cependant, il est tout de même préférable d'en connaître quelques notions si vous désirez poursuivre agréablement la lecture de ce tutorial.

Très peu de code sera fourni dans les premières parties et les notions abordées sont très intuitives.

Minimum requis

Le .NET Framework 2.0 et Visual C# 2005 ou équivalent (Visual Studio 2005, SharpDevelopp 2.0, etc.). Nous utiliserons ici le .NET Framework 2.0 sous Windows XP ainsi que Visual C# 2005 dans sa version anglaise.

Si vous utilisez encore les .NET Framework 1.1 et Visual Studio 2003 (ou équivalent), vous pourrez tout de même suivre ce tutorial car les différences entre les deux versions ne seront quasiment pas exploitées ici.

A propos de la portabilité

Mono gère depuis fin 2006 les SWF, vos interfaces graphiques seront donc compatibles Windows, Linux, OSX... [Plus d'informations...](#)

¹ System.Windows.Forms

² Graphical User Interface

³ Editeur de Développement Intégré

Chapitre 1

Premiers pas

Nous allons commencer pas créer une petite application qui devrait vous prendre cinq secondes si vous avez déjà un peu touché aux WinForms ou sinon, tout au plus deux minutes.

Essayez de la faire vous-même avant de lire la solution. Si c'est trop facile pour vous, passez donc au chapitre suivant.

Application

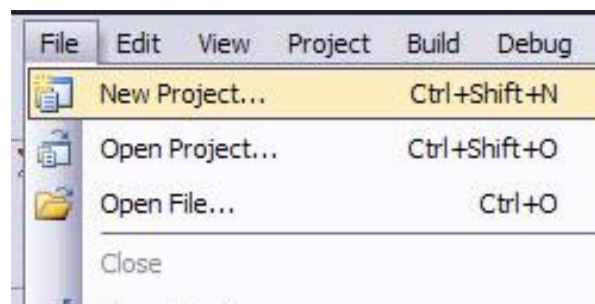
Réalisez une application graphique présentant un unique bouton dont l'étiquette est "**Cliquez ici**" et qui affichera une boîte de dialogue contenant un message de votre cru.



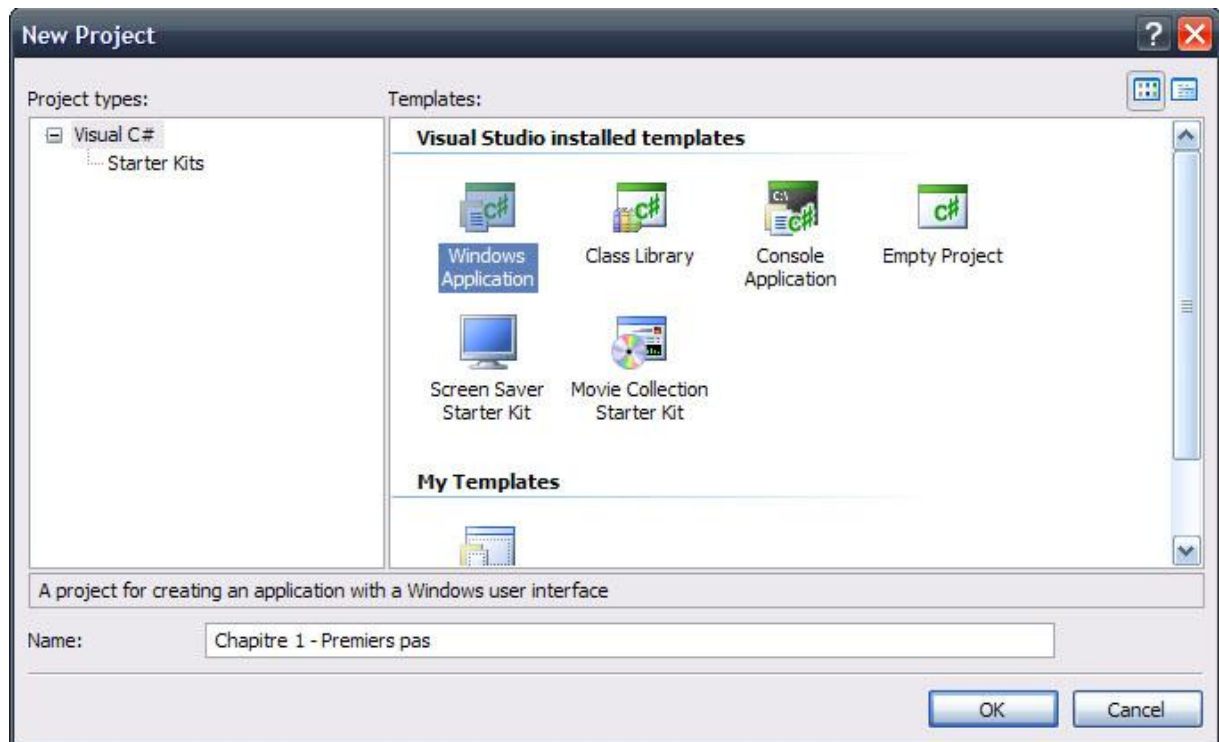
Voyons un peu ce qu'il fallait faire...

Inutile de vous demander d'ouvrir Visual C#, vous l'avez sûrement déjà fait...

Créez une « Application Windows ». Pour cela, allez dans *File* → *New Project*, puis sélectionnez *Windows Application*. Renommez là comme il vous plaît puis validez !

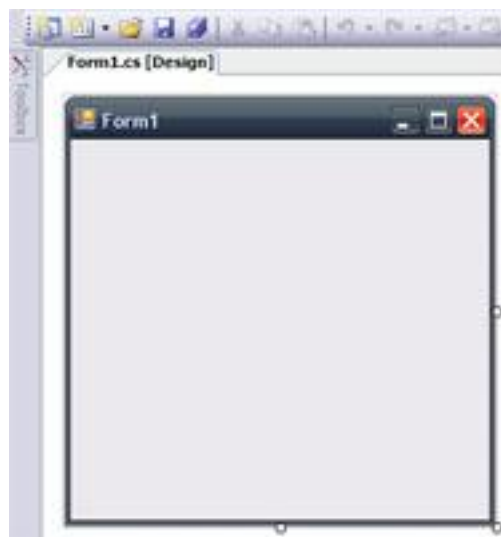


Faites File → New Project

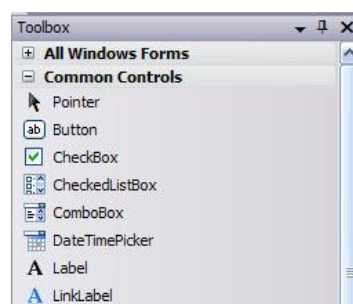


Sélectionnez Windows Application

Vous arrivez alors sur une fiche Windows, appelée communément sous .NET Form, ou formulaire en français :



Dans votre EDI, vous avez à votre disposition un panneau qui se nomme la Toolbox (ou boîte à outils) sous Visual C#. Celle-ci devrait se trouver sur la droite ou sur la gauche et est déployable à souhait.



Remarque : Si la Toolbox n'est pas affichée, allez dans le menu View et cliquez sur la ligne marquée Toolbox.

Vous remarquerez, si vous êtes un peu observateur, que cette Toolbox contient une multitude d'items qui correspondent en fait à des raccourcis vers des **Control** (ou contrôles) tels que des **Label**, **Button**, **ListBox**, etc.

Sélectionnez le contrôle qui vous intéresse et cliquez sur le formulaire à l'endroit où vous souhaitez le placer.

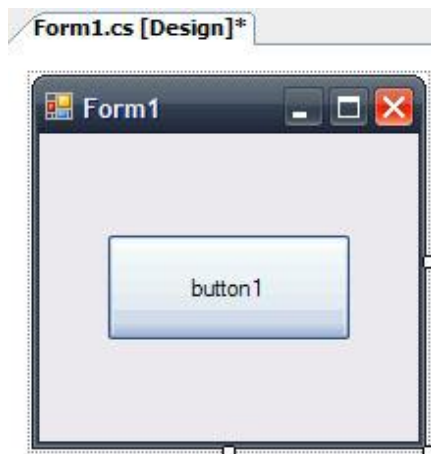
Votre contrôle est en place :



Remarque : Vous pouvez également placer un contrôle sur votre **Form** en effectuant un Drag & Drop (ou glisser-déposer) depuis la Toolbox jusqu'au formulaire (en mode Designer).

Si votre contrôle est mal positionné, vous pouvez toujours le déplacer en le sélectionnant directement sur votre **Form** puis en déplaçant le curseur (lequel est à présent suivi par votre contrôle) à l'endroit désiré...

Redimensionnez à présent la fenêtre et le bouton de manière à obtenir la présentation suivante...



... le bouton étant parfaitement centré.

Pour centrer le bouton, sélectionnez-le et cliquez successivement dans la barre d'outils de mise en forme sur *Center Horizontally* et sur *Center Vertically*.

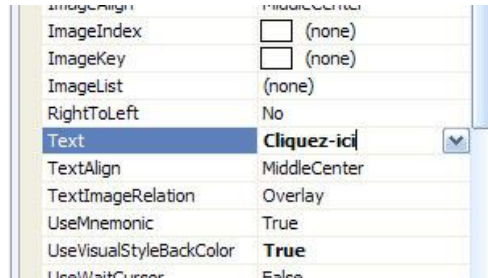


Si cette barre de raccourcis n'est pas affichée, vous pourrez la trouver dans le menu *View* → *Toolbars...*

Votre bouton est donc placé mais il affiche pour le moment **"Button1"**.

Pour modifier son étiquette, sélectionnez votre bouton puis allez dans le panneau des propriétés (si celui-ci n'est pas affiché, vous le retrouverez également dans le menu *View*).

Dans ce dernier, existe une ligne marquée **Text** et contenant le texte **"Button1"**. Modifiez **"Button1"** et tapez **"Cliquez-ici"**.



***Attention :** Les contrôles pouvant afficher du texte ont généralement leurs deux propriétés **Text** et **Name** à la même valeur que le texte affiché par le contrôle. **Name** et **Text** sont deux propriétés distinctes : **Text** permet de modifier le texte qui sera affiché alors que **Name** modifie le nom que porte votre contrôle dans le code source.*

...

Reste maintenant à associer à l'évènement clic de ce bouton, l'affichage du message.

Faites un double clic sur le bouton (depuis le mode Designer). Vous vous retrouvez alors en mode codeur, placé directement dans une fonction portant le nom (**Name**) de votre bouton, suivie de `_Click(...)`.

Ajoutez alors le code ci-après au sein de cette méthode :

```
MessageBox.Show("Merci d'avoir cliqué", "Message");
```

N'attendez plus, l'application est terminée !

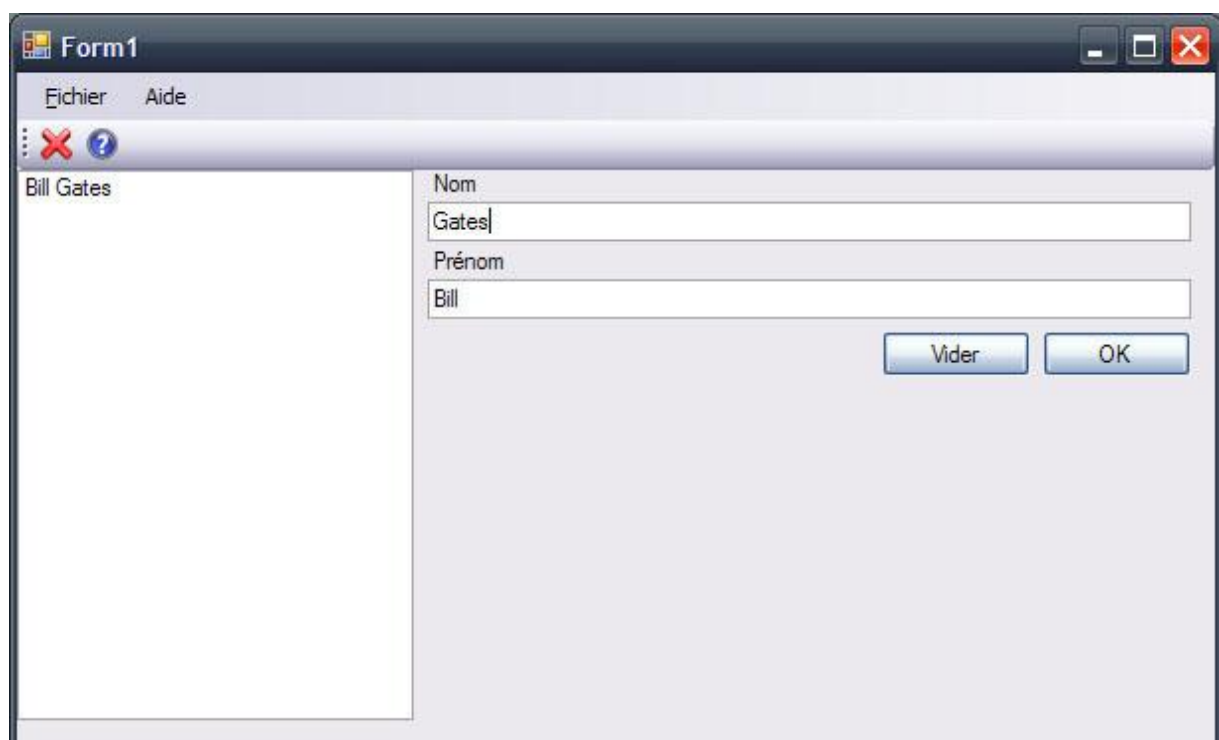
Exécutez le code (raccourci F5 dans Visual Studio) et admirez le résultat !

Chapitre 2

Une interface plus riche

La première application était vraiment simplissime, passons à quelque chose d'un peu plus poussé.

Application





Ci-dessous, les caractéristiques de cette application...

Pour toutes dimensions de la fenêtre, que l'utilisateur peut redimensionner à souhait (comportement par défaut), il faut que :

- la barre de menu reste fixée en haut de la fenêtre
- de même pour la barre de raccourcis (sauf qu'elle est fixée au bas de la barre de menu)
- le `SplitContainer` occupe tout l'espace restant
- la `ListBox` occupe tout l'espace du `Panel` gauche du `SplitContainer`
- les deux `TextBox` occupent toute la largeur du `Panel` droit du `SplitContainer`
- l'étiquette "Nom" reste fixée dans le coin supérieur gauche du `Panel` droit du `SplitContainer`
- l'étiquette "Prénom" reste à la même distance des deux `TextBox` et de la bordure gauche du `Panel` droit du `SplitContainer`
- l'espace séparant les deux boutons (entre eux) soit toujours le même
- l'espace séparant les deux boutons de la `TextBox` soit toujours le même
- l'espace séparant le bouton OK de la bordure droite du `Panel` droit du `SplitContainer` soit toujours le même

Et voici la liste des « actions » :

- le menu fichier contient 3 éléments :
 - "RAZ" pour vider les [TextBox](#) et la [ListBox](#)
 - un séparateur
 - "Quitter" ... pour quitter l'application
- le menu d'aide contient un seul élément :
 - "A propos de" qui affiche quelques informations à propos de cette application (vous mettez ce que vous voudrez) dans une [MessageBox](#) (et pas autre chose)
- le bouton  est un raccourci de "RAZ"
- le bouton  est un raccourci de "A propos de"
- le bouton "Vider" efface simplement le contenu des deux [TextBox](#)
- le bouton "OK" ajoute le contenu des [TextBox](#) associés à "Nom" et à "Prénom" en les séparant par un espace. Concrètement, si on a dans la [TextBox](#) de "Nom" : "Gates" et dans celui de "Prénom" : "Bill", alors en cliquant sur le bouton "OK", on ajoute "Bill Gates" dans la [ListBox](#).

Remarque : Si vous observez bien l'impression écran de l'application, vous constaterez que le F du menu Fichier est souligné... non pas par hasard bien évidemment. Il s'agit là d'un raccourci clavier pour accéder au menu Fichier qui est automatiquement attribué par Windows. Lorsque l'utilisateur fait la combinaison Alt+F, il ouvrira ainsi le menu Fichier (sans touche à la souris bien sûr).

Pour qu'une lettre de menu apparaisse soulignée (et attribuer ainsi un raccourci clavier à votre menu), vous devez la faire précéder du caractère esperluette ('&').

Cela s'applique à tous les menus, sous menus... et la lettre du raccourci peut être n'importe quelle lettre du « mot » du menu, sous menu...

Aide :

- Le splitter est en fait un élément de [SplitContainer](#) (que vous pourrez trouver dans la [ToolBox](#))
- Intéressez vous à la propriété [Dock](#) de vos contrôles
- Intéressez vous également à la propriété [Anchor](#) (pour voir ce qu'elle fait vraiment, vous devrez exécuter l'application, la manipuler un peu dans tous les sens...)

A vous de jouer !

Trop facile ? Dans ce cas passez directement au chapitre suivant.

Pas si simple que ça ? Alors lisez ce qui suit !

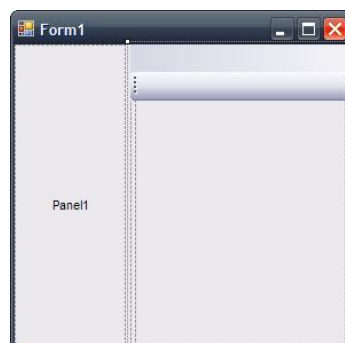
Pour commencer, créez votre application Windows.

Il faut à présent ajouter les contrôles. Nous allons faire suivant deux méthodes :

- la méthode bourrin (pas intelligente du tout)
- la méthode intelligente

Première méthode : A la bourrin !

Comme son nom vous l'a probablement laissé supposer, ne réfléchissez pas quant à la façon logique dont les contrôles sont disposés, placez les tous, on verra après...



Soit vous avez de la chance, soit vous n'en avez pas et vous êtes dans un sacré pétrin, ou du moins, vous vous compliquez sérieusement la tâche en procédant ainsi. Voyons plutôt l'autre méthode.

Seconde méthode : Réfléchissez un peu !

Et vous serez obligé, à un moment ou à un autre, de réfléchir à la manière dont s'organise votre GUI. Si on devait poursuivre avec la première méthode, on sélectionnerait dans l'ordre approprié chaque contrôle afin de lui donner les propriétés souhaitées, de le placer comme il faut, etc.

Ici, nous allons en fait placer les contrôles un à un en les positionnant immédiatement correctement puis leur donner directement leurs propriétés. Cela évitera notamment d'avoir à revenir sur chacun d'entre eux pour en modifier une petite donnée... On gagne donc du temps.

Si on regarde bien l'interface, on s'aperçoit qu'un des contrôles encapsule les autres, qu'il les contient... C'est d'ailleurs son nom vu qu'il s'agit du [SplitContainer](#) (qui contient [ListBox](#), [TextBox](#), [Label](#) et [Button](#)).

Nous n'avons donc que trois contrôles qui se trouvent au même « niveau » : le menu, la barre de raccourcis et le [SplitContainer](#).

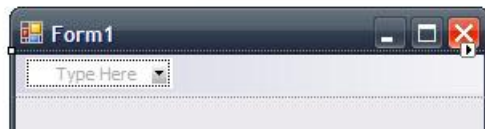
Lequel placer en premier ?

Si on regarde ce que l'on attend de l'application, on doit avoir :

- la barre de menu reste fixée en haut de la fenêtre
- de même pour la barre de raccourcis (sauf qu'elle est fixée au bas de la barre de menu)
- le [SplitContainer](#) occupe tout l'espace restant

Par conséquent, on peut placer le [SplitContainer](#) en premier (sinon où mettrait-on la barre de menu et de raccourcis ?), ni la barre de raccourcis (puisque'elle est fixée à la barre de menu).

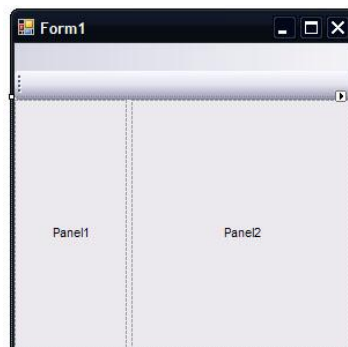
Vous devez par conséquent placer la barre de menu en premier.



Par chance, ou plutôt par commodité et conventionalité, la barre de menu s'est automatiquement placée en haut de la fenêtre. Rien de plus à faire à son sujet...

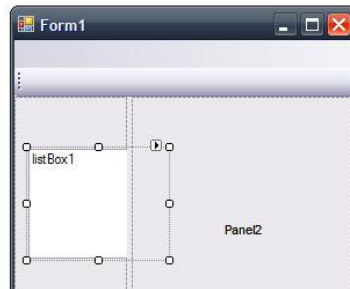
Ajoutez de même la barre de raccourcis.

Placez à présent le [SplitContainer](#). Celui-ci se place également immédiatement comme il faut : il occupe tout de suite tout l'espace restant du formulaire. C'est pour cela qu'il faut réfléchir...



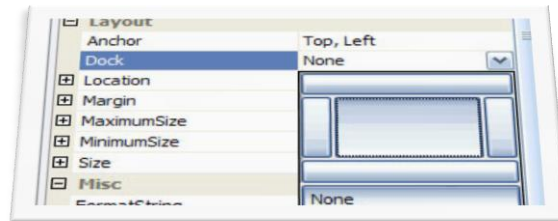
Comme vous pouvez le constater, votre [SplitContainer](#) se divise en deux [Panel](#). Chaque [Panel](#) représente en fait un nouveau support graphique complet qui réagit de la même manière qu'une [Form](#) vide.

Il faut à présent placer la **ListBox** dans le **Panel** de gauche. Commencez donc par l'ajouter.

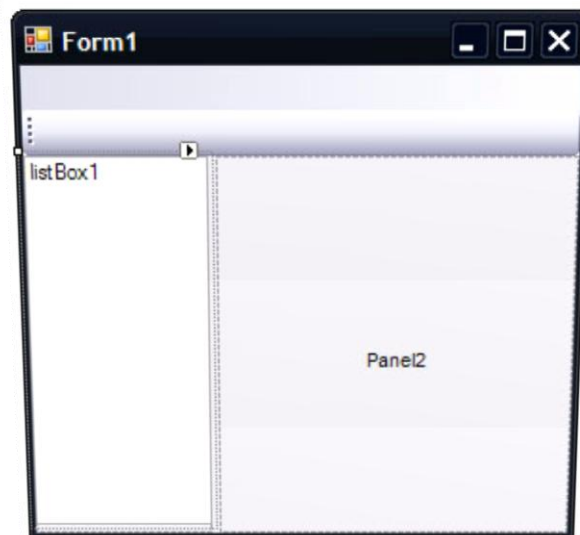


Mais un problème se pose alors. Nous voulons que la **ListBox** occupe tout le **Panel** de gauche et pas seulement une partie.

Pour ce faire, réglez la propriété Dock du contrôle **ListBox** sur Fill :



Voilà, la **ListBox** occupe à présent tout le **Panel** de gauche.



Remarque : Testez également les autres valeurs de la propriété Dock pour voir ce qu'elles font.

Occupons nous à présent du **Panel** de droite.

Placez-y le premier **Label** et modifiez en le texte pour qu'il affiche "**Nom**". Si vous essayez de le placer dans le coin supérieur gauche du **Panel**, vous verrez apparaître des petits pointillés bleus liant les bordures du **Panel** aux parties du **Label** qui en sont le plus proche...



Nous voulons à présent que ce **Label** reste fixé en haut à gauche du **Panel**. Pour cela, il faut définir son ancre (propriété **Anchor**) sur **Top, Left**. Etant donné qu'il s'agit là de la valeur par défaut de cette propriété, vous n'aurez rien à modifier.



Qu'est ce qu'une ancre ?

La définition est en fait très proche de celle que l'on pourrait donner à une ancre de navire...

Anchor : obtient ou définit les bords du conteneur auquel un contrôle est lié et détermine comment un contrôle est redimensionné avec son parent. (Définition tirée de [MSDN 2007](#))

Si vous ne percevez pas encore leur utilité, pas de panique, vous le verrez mieux un peu plus loin.

Fin de la partie Design

Terminons le dessin de notre application !

Placez à présent la première **TextBox** de manière à ce que celle-ci occupe toute la largeur du **Panel2** et qu'elle se trouve juste sous le premier **Label**. Lorsque les dimensions et la position de la **TextBox** seront bonnes, vous verrez trois indicateurs de positions (bleus) s'afficher en essayant de redéplacer le contrôle.



Maintenant que la **TextBox** est convenablement placée, empressez vous de la renommer (propriété **Name** et non pas **Text**) de manière cohérente (en sachant à quoi la **TextBox** va servir). Nous la nommerons ici "**textBoxOfName**" (veuillez respecter la casse). Modifiez ensuite le contrôle de manière à ce que ses ancres soient fixées à gauche, à droite et en haut.

Parfait, c'est presque fini !

Les deux contrôles suivants sont exactement les mêmes que les deux précédents... Faites donc un copier/coller des deux contrôles précédents.

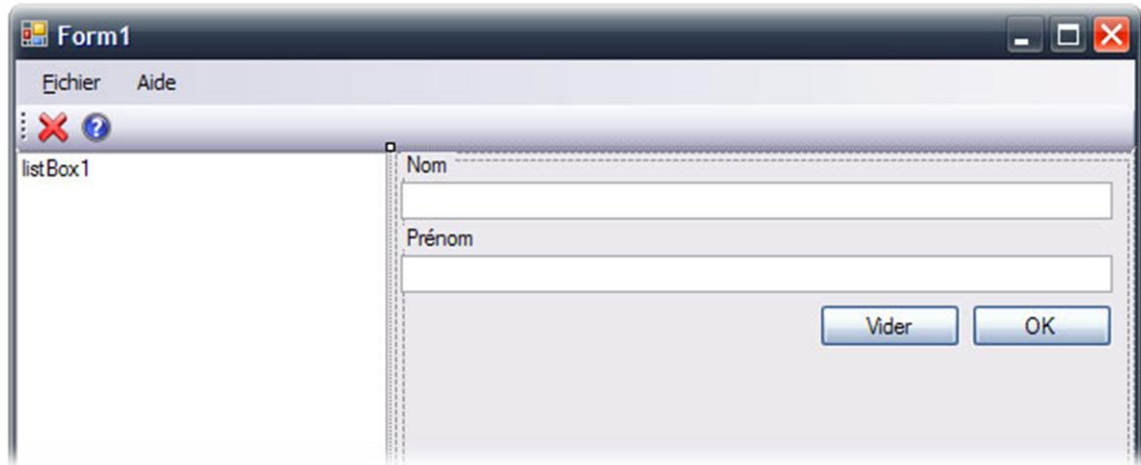
Attention : Au moment de coller vos contrôles, assurez vous que la zone active (cernée par des pointillés) est bien celle où vous voulez coller vos contrôles.

Une fois collés, gardez les deux éléments sélectionnés (ou resélectionnez-les s'ils ne le sont plus tout les deux) et déplacez les tout deux en même temps au bon endroit...

Vérifiez ensuite leurs propriétés d'ancres et modifiez les propriétés Name et Text comme précédemment. Bien entendu, la propriété Text du **Label** vaut à présent "**Prénom**" et la propriété Name de la **TextBox** vaut "**textBoxOfFirstName**".

Ajoutez maintenant les deux boutons. Modifiez en les propriétés Text comme il convient et fixez leurs ancres sur Right et Top.

Reste à ajouter les boutons et les liens dans les barres de menu et de raccourcis, mais ça vous n'aurez pas de mal pour trouver comment faire.



Evènements

Sans même réfléchir, commencez par créer les deux fonctions des boutons OK et Vider... Vous pouvez, si vous le souhaitez, modifier auparavant le nom (propriété Name) afin que les noms des méthodes soient plus explicites (exemple : pour le bouton OK, mettez buttonOK).

Voyons un peu ces deux nouvelles méthodes.

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

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

... et rappelons nous ce qu'elles devaient faire :

- le bouton "**Vider**" efface simplement le contenu des deux **TextBox**
- le bouton "**OK**" ajoute le contenu des **TextBox** associés à "**Nom**" et à "**Prénom**" en les séparant par un espace. Concrètement, si on a dans la **TextBox** de "**Nom**" : "**Gates**" et dans celui de "**Prénom**" : "**Bill**", alors en cliquant sur le bouton "**OK**", on ajoute "**Bill Gates**" dans la **ListBox**.

Effacer le contenu d'une **TextBox** revient à mettre une valeur "" dans sa propriété Text. Par conséquent, la méthode associée au clic sur le bouton vider contient :

```
textBoxOfFirstName.Text = "";
textBoxOfName.Text = "";
```

Pour ajouter un élément à une [ListBox](#), il suffit d'utiliser la méthode membre `Add()` de la propriété `Items`.

```
listBox1.Items.Add(textBoxOfFirstName.Text + " " + textBoxOfName.Text);
```

Les derniers événements à créer sont tous très simples. Vous devriez y arriver tout seul à présent.

Aide : Pour quitter une application qui utilise des [Forms](#), il faut utiliser la méthode `Exit()` de la classe statique [Application](#).

Retour sur les ancrés

Pour voir l'utilité des ancrés, il n'y a pas grand-chose à faire...

Sélectionnez l'une des [TextBox](#) et au lieu de laisser les ancrés fixés sur `Left`, `Right`, `Top`, mettez simplement `Left`.

Exécutez le programme...

Tout va bien, aucune différence.

Maintenant, redimensionnez l'application en en modifiant la largeur (en étirant le côté droit de la fenêtre, vous verrez mieux ce qui se passe) ou la hauteur... Constatez la différence !

Vous pouvez vous amuser comme cela en modifiant les ancrés de vos contrôles, en les redimensionnant pour avoir tel ou tel comportement.

Après un tel exercice, vous devriez avoir parfaitement compris à quoi servent les ancrés.

Chapitre 3

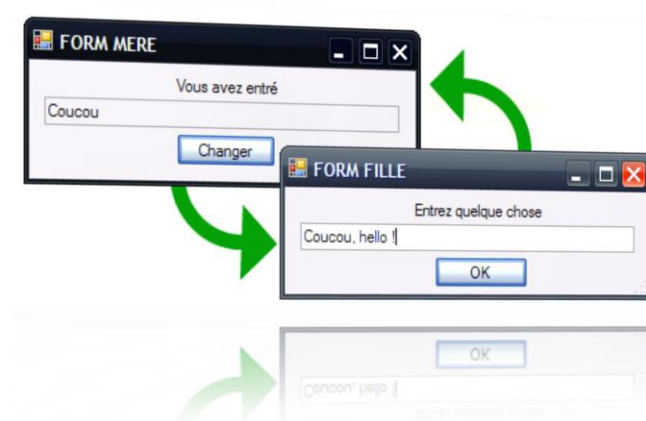
Interfaces multifenêtres

Un seul formulaire pour toute une application, c'est bien souvent insuffisant et à moins de travailler énormément sur le dynamisme de l'interface, c'est tout simplement impossible (dépend de l'application bien entendu).

Dans ce chapitre, nous allons voir comment appeler des formulaires depuis un autre formulaire et comment faire pour que formulaires enfants et parents puissent s'échanger des informations.

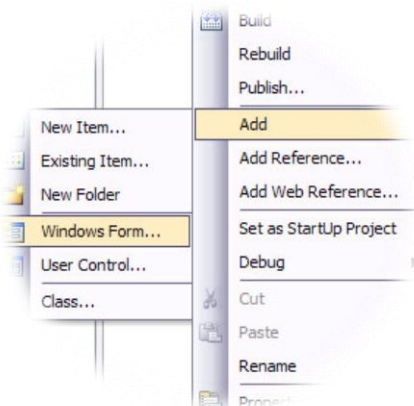
Remarque : La complexité de l'interface ne nous intéresse pas dans ce chapitre.

Un cas simple



Cette application est très basique... Le formulaire principal ne fait rien : il ne sert qu'à afficher le formulaire secondaire ("**FORM FILLE**") qui permet d'entrer du texte. En cliquant sur le bouton OK, le texte est directement transmis dans le champ du formulaire principal... Si on reclique ensuite dans le formulaire principal sur le bouton Changer, on réouvre le formulaire secondaire et on y retrouve le texte que l'on y avait inscrit, etc., etc.

Commencez par créer les deux formulaires et à placer les contrôles nécessaires. Renommez-les et paramétrez-les comme il vous convient.



Note : Pour ajouter un formulaire à votre projet : clic droit dans l'explorateur de solution, Add → Windows Form...

Modifiez également la propriété Name du formulaire enfant et mettez-y "**FormFille**" (tout accroché).

Créez ensuite l'évènement associé au clic sur le bouton Changer et rendez vous dans la méthode créée pour...

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

C'est ici que vous devez commander l'affichage de votre formulaire enfant.

Pour afficher le formulaire enfant, il faut d'abord que celui-ci existe : il faut d'abord le créer !

```
FormFille ff = new FormFille();
```

Ensuite, il suffit de l'afficher :

```
ff.Show();
```

Dans notre méthode, cela donne donc :

```
private void buttonChange_Click(object sender, EventArgs e)
{
    FormFille ff = new FormFille();
    ff.Show();
}
```

Voilà, c'est bon ! Vous pouvez tester...

***Remarque :** Il existe en fait deux méthodes pour afficher un formulaire : Show() et ShowDialog() qui réagissent différemment... Comparez ces deux méthodes.*

Vous remarquerez en les comparant et en réduisant votre application (notamment) que les deux icônes dans la barre des tâches (une pour chaque formulaire) ne sont pas toutes les deux utiles.

Nous utiliserons, pour des raisons pratiques et dans le but de minimiser le code, la méthode ShowDialog() qui renvoie un objet de type [DialogResult](#).

Si vous ne voulez pas qu'un formulaire (n'importe laquelle) ait une icône qui s'affiche dans la barre des tâches, réglez sa propriété ShowInTaskbar sur [false](#).

La propriété ShowIcon qui se trouve juste à côté permet de choisir si l'icône de l'application (en haut à gauche) doit être affichée ou non...

Pour en finir avec le formulaire parent (du moins pour l'instant), réglez la propriété ReadOnly de la [TextBox](#) sur [true](#) (l'utilisateur ne doit pas pour modifier le texte que celle-ci contient).

Allons à présent faire un petit tour dans le formulaire enfant...

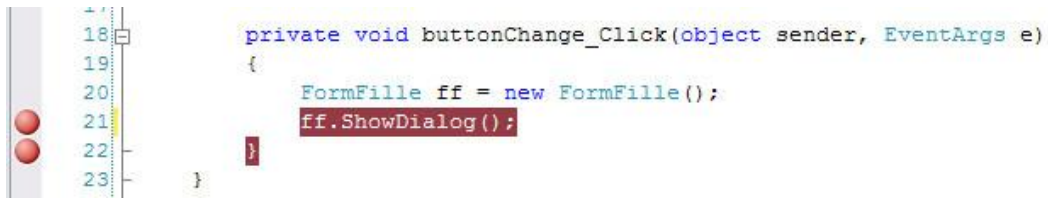
Nous voulons que celui-ci se ferme lorsque l'utilisateur appuie sur OK. Pour cela, il y a plusieurs façons de procéder, mais nous allons utiliser ici la méthode la plus simple et la plus intelligente qui soit...

Il suffit d'utiliser en fait une propriété du formulaire : AcceptButton.

Voyez par vous-même :

- associez à la propriété DialogResult du bouton OK la valeur OK
- associez à la propriété AcceptButton du formulaire enfant le bouton OK
- testez !

Remarque : Cela ne marchera que si vous affichez le formulaire enfant avec la méthode `ShowDialog()`. Alors quel est ce miracle ? Eh bien pour le savoir, rien de mieux que d'effectuer un petit débogage en plaçant judicieusement deux breakpoints.



Note : Un breakpoint se place (sous Visual Studio) en cliquant dans la marge sur la ligne où l'on veut poser le breakpoint. Un breakpoint entraîne nécessairement la « mise en pause » du programme lorsque celui-ci (de breakpoint) est rencontré.

En effectuant le débogage (F5 pour continuer une fois arrêté), vous vous apercevrez alors qu'en appuyant sur le bouton OK, vous revenez dans la méthode associée au clic du bouton Changer. La suite n'est qu'une histoire de portée... En quittant la méthode, vous détruisez tout ce que celle-ci a créé, vous détruisez donc le formulaire enfant et celui-ci disparaît.

Remarque : En fait le formulaire ne disparaît parce qu'il est détruit dans ce cas (cela serait suffisant dans d'autres cas)... Il s'agit en fait du comportement de la méthode `ShowDialog()`, qui, lorsqu'elle « sent » qu'un `DialogResult` est émis, ferme le formulaire...

Il ne nous resterait donc qu'à placer le code nécessaire entre ces deux breakpoints pour récupérer puis afficher le texte du formulaire enfant.

Sauf que l'on ne veut effectuer cette action que si le bouton OK est cliqué (pas si la croix est cliquée).

En sachant que la méthode `ShowDialog()` (ce n'est pas le cas de `Show()`) renvoie un objet de type `DialogResult` et que le `DialogResult` associé au bouton OK est justement OK, il suffit d'ajouter la condition :

```
if (ff.ShowDialog() == DialogResult.OK)
{
}
}
```

A présent, on peut placer le code traitant le texte entre les accolades de la condition.

Mais d'abord, il faut pouvoir accéder à ce texte (qui se trouve dans le formulaire enfant et auquel nous n'avons pas accès par défaut).

Pour cela, deux possibilités :

- la mauvaise : on rend la `TextBox` du formulaire enfant publique (c'est-à-dire accessible depuis l'extérieur) en réglant sa propriété Modifiers sur Public. On a alors directement accès à la `TextBox` depuis le formulaire parent et il ne reste plus qu'à taper :

```
textBox1.Text = ff.textBoxEnfant.Text;
```

- la bonne : on va dans le code source du formulaire enfant et on ajoute la propriété (publique bien sûr) suivante :

```
public string InputText
{
    get { return textBoxEnfant.Text; }
    set { textBoxEnfant.Text = value; }
}
```

Il suffit alors de questionner cette propriété depuis le formulaire parent :

```
textBox1.Text = ff.InputText;
```

Contrôlez ! Tout marche bien...

Reste à présent à passer le texte de la première **TextBox** au formulaire enfant lorsque celui-ci est créé et à insérer ce texte dans la **TextBox** du formulaire enfant.
En utilisant la bonne méthode (la dernière), c'est immédiat.

Il suffit de passer la chaîne de caractères de la première **TextBox** (celle du formulaire parent) en utilisant la propriété créée précédemment.

Pour que cela fonctionne comme souhaité, il faut faire attention de passer la chaîne avant d'afficher le formulaire enfant.














Finalement, le code complet du bouton Changer donne quelque chose du genre :

```
private void buttonChange_Click(object sender, EventArgs e)
{
    FormFille ff = new FormFille();
    ff.InputText = textBox1.Text;
    if (ff.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = ff.InputText;
    }
}
```

Voilà pour le cas simple...

Remarque : Nous traiterons un cas plus complexe (et nécessitant quelques connaissances supplémentaires en C# que vous aurez acquise d'ici là) dans le chapitre 5.

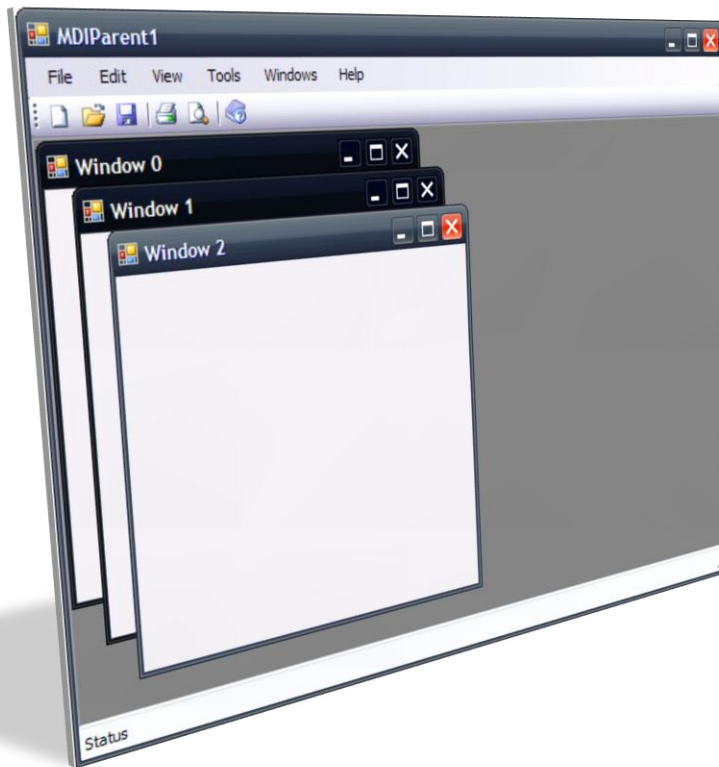
Quelques propriétés :

-  **Enabled** : en réglant cette propriété sur **false** pour une **TextBox**, cela revient à griser la case de la **TextBox** et à interdire la saisie et la sélection du texte qui s'y trouve. Cette propriété existe pour tous les contrôles.
-  **Visible** (**true** ou **false**) : affiche ou masque le contrôle (que ce soit un formulaire, un bouton, une boîte de texte...)
-  **AcceptButton** : vous permet de choisir quel bouton du formulaire sera activé lorsque l'utilisateur appuiera sur la touche Entrer.
-  **CancelButton** : même chose mais avec la touche Echap.
-  **ControlBox** (**true** ou **false**) : affiche ou masque la barre de contrôle du formulaire
-  **BackColor** : permet de choisir la couleur de fond du contrôle (**Form**, **Button**, etc.)
-  **FormBorderStyle** : permet de changer le style de la fenêtre
-  **Icon** : associe une icône au formulaire
-  **MinimizeBox** (**true** ou **false**) : *transparent*
-  **MaximizeBox** (**true** ou **false**) : *transparent*
-  **TopMost** : force le formulaire à rester toujours au premier plan (et ceux par rapport à toutes les applications)
-  **Size** : modifier ou obtenir la taille du contrôle
-  **StartPosition** : obtient ou définit la position initiale du formulaire au moment de l'exécution


Il ne s'agit nullement d'une liste exhaustive des propriétés des propriétés de la classe **Form**... Pour obtenir cette liste, rendez-vous sur [MSDN](https://docs.microsoft.com/fr-fr/dotnet/api/system.windows.forms.form?view=netframework-4.7.2).


Interface à documents multiples


Nous allons nous intéresser à présent à une autre façon de gérer les multifenêtres, il s'agit des MDI⁴. Si le nom ne vous dit rien, l'interface vous rappellera certainement quelques logiciels :





Citons immédiatement les propriétés en rapport avec les MDI :

 ActiveMdiChild : obtient la fenêtre enfant MDI active.


 IsMdiChild : obtient une valeur indiquant si le formulaire est un formulaire enfant MDI.


 IsMdiContainer : obtient ou définit une valeur indiquant si le formulaire est un conteneur de formulaires enfants d'interface multidocument (MDI).

 MdiChildren : obtient un tableau de formulaires représentant les formulaires enfants MDI qui sont apparentés à ce formulaire.

 MdiParent : obtient ou définit le formulaire parent MDI en cours de ce formulaire.

Et les méthodes :

 LayoutMdi : organise les formulaires enfants MDI au sein du formulaire parent MDI.

 ActivateMdiChild : active l'enfant MDI d'un formulaire.

Rien de bien difficile en fin de compte pour créer ce type d'interface...

Il suffit simplement de créer un formulaire qui va contenir les autres et de mettre sur **true** sa propriété `IsMdiContainer` (le style du formulaire change alors immédiatement) et de créer d'autres formulaires en spécifiant quel est leur formulaire parent avec la propriété `MdiParent`.

Par exemple, commencez par réaliser l'interface ci-contre...

... avec le menu Fichier qui ne contient que "Quitter" et un seul bouton sur la barre de raccourcis.

Pour avoir l'aspect inférieur du formulaire ci-contre, mettez sur **true** sa propriété `IsMdiContainer`.

Sur l'unique bouton de la barre de raccourcis, mettez :

```
Form form = new Form();  
form.Text = "Form " + MdiChildren.Length;  
form.MdiParent = this;  
form.Show();
```

Lancez et testez votre application !



⁴ Multiple Document Interface

Rien de plus simple donc...

Vous pouvez bien sûr ajouter des formulaires que vous avez-vous-même fait avec le Designer (ou sans).

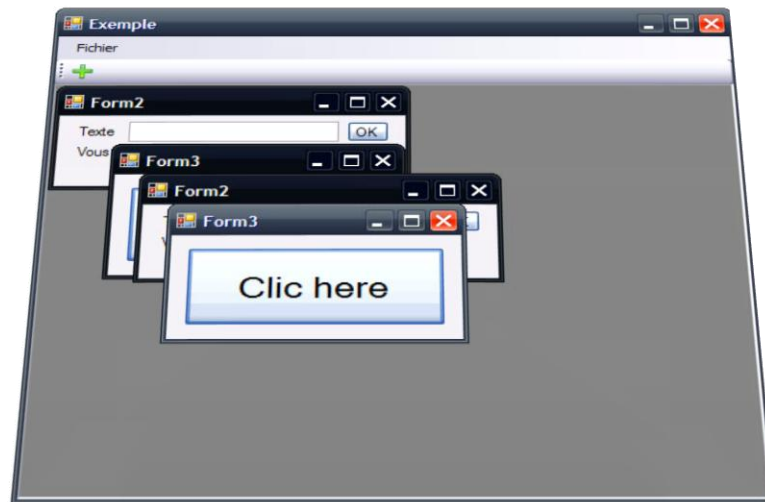
Exemple :

Créez le formulaire suivant...



... et ajoutez le à votre MDI.

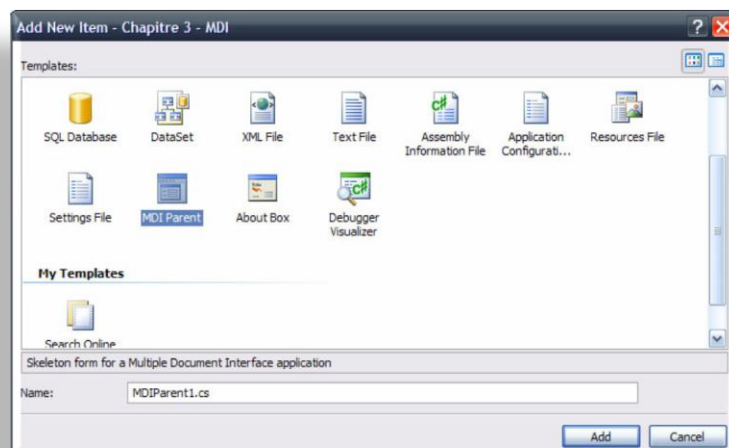
Créez un deuxième formulaire (comme vous le voulez) et faites en sortes qu'en cliquant sur le bouton Nouveau, on ait une fois le formulaire 1 qui soit ajoutée, puis une fois le formulaire 2, une fois le formulaire 1, etc.



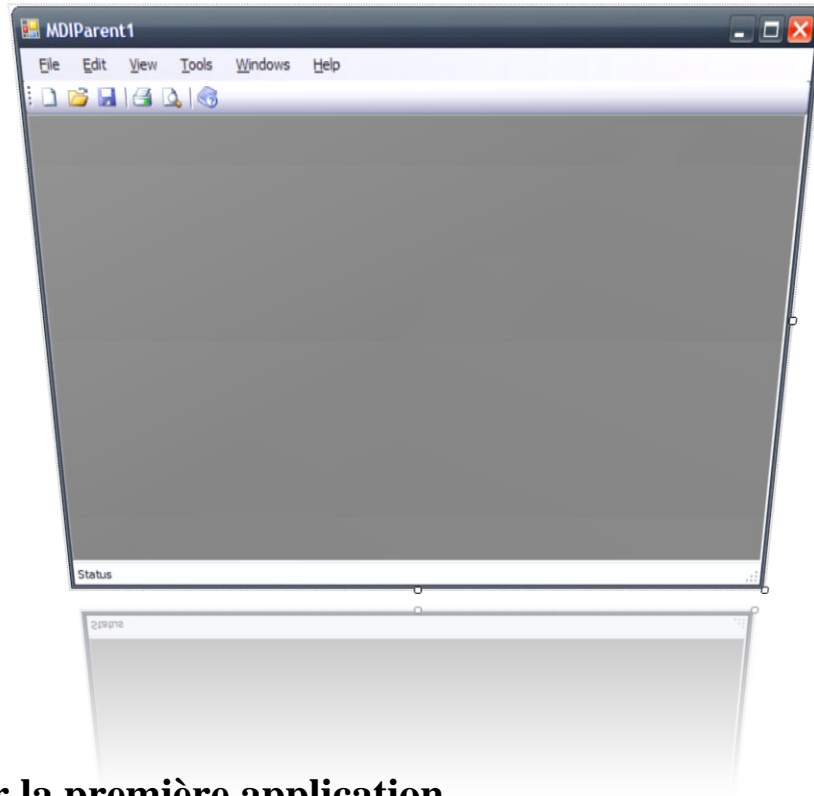
Si vous voulez faire communiquer les formulaires enfants avec le formulaire parent, c'est comme précédemment, créez des propriétés, etc. Les communications entre formulaires enfants seront abordées au chapitre 5.

Remarque : Intéressez vous à la propriété *LayoutMdi* et voyez ce qu'elle permet de faire...

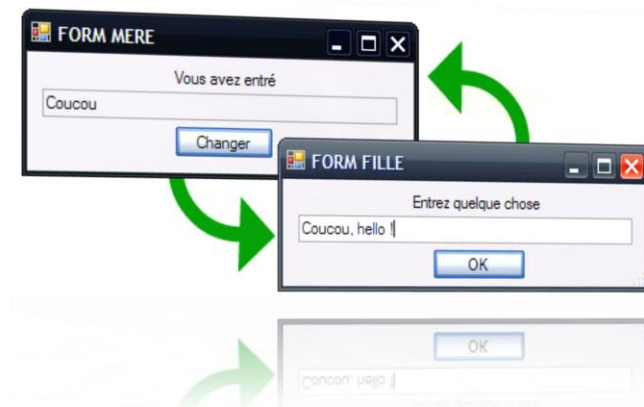
Si vous avez besoin d'une interface relativement standard gérant les MDI, Visual C# offre la possibilité de créer directement un formulaire MDI.



Vous obtenez une interface toute prête qui traite déjà un certain nombre de fonctions (nouveau, enregistrer, ouvrir...).



Retour sur la première application



Nous allons modifier le code de la première application de manière à ce que la fenêtre fille ne soit plus détruite à chaque fois que l'on clique sur le bouton OK.

Pour cela, revoyons comment le formulaire enfant était créée :

```
private void buttonChange_Click(object sender, EventArgs e)
{
    FormFille ff = new FormFille();
    ff.InputText = textBox1.Text;
    if (ff.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = ff.InputText;
    }
}
```

Ici, on crée une instance d'objet de la fenêtre fille avec le mot clé `new`. Cette instance est stockée dans la variable `ff`. Ensuite, on demande à l'instance de la fenêtre fille de s'afficher. On fait des tests... et une fois sorti des tests, plus rien... on sort de la méthode du clic et on perd toute trace de la fenêtre fille : l'instance est détruite et au prochain clic, tout sera reconstruit... ce qui n'est pas toujours souhaitable en fonction de ce que l'on veut réaliser comme application.

Pour y remédier, c'est simple, il suffit d'ajouter `FormFille ff` aux champs de la classe du formulaire parent (ou plus généralement, du formulaire qui sera le « lanceur » de la nouvelle fenêtre)...

```
public partial class Form1 : Form
{
    //Champs
    FormFille ff;

    //Constructeur
    public Form1()
    {
        InitializeComponent();
    }

    //Méthode du bouton Changer
    private void buttonChange_Click(object sender, EventArgs e)
    {
        FormFille ff = new FormFille();
        ff.InputText = textBox1.Text;
        if (ff.ShowDialog() == DialogResult.OK)
        {
            textBox1.Text = ff.InputText;
        }
    }
}
```

... et de créer l'instance de cette `Form` dans le constructeur de la classe mère.

... et de créer l'instance de ce formulaire dans le constructeur de la classe mère.

```
//Constructeur
public Form1()
{
    InitializeComponent();

    ff = new FormFille();
}
```

Notre objet, notre formulaire enfant existe à présent tout le temps... enfin, jusqu'à ce que le formulaire parent soit détruit lui aussi (dans cette application, cela n'arrivera qu'en l'a quittant).

Voyons à présent à quoi doit ressembler le code du bouton Changer :

```
//Méthode du bouton Changer
private void buttonChange_Click(object sender, EventArgs e)
{
    ff.InputText = textBox1.Text;
    if (ff.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = ff.InputText;
    }
}
```

On peut même faire mieux !

En effet, vu que le formulaire enfant existera toujours, les contrôles qu'elle contient aussi et en particulier la **TextBox**.

Autrement dit, entre deux clics sur le bouton Changer, la **TextBox** et par conséquent le texte qu'elle contient (ce qui nous intéresse) ne seront pas modifiés et il devient alors inutile de repasser au formulaire enfant le texte à modifier :

```
//Méthode du bouton Changer
private void buttonChange_Click(object sender, EventArgs e)
{
    if (ff.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = ff.InputText;
    }
}
```

Et on peut également modifier la propriété du formulaire enfant, la modification de la **TextBox** qu'il contient devenant inutile (dans cette application) :

```
public string InputText
{
    get { return textBoxEnfant.Text; }
}
```

Contrôlez ! Cela marche comme auparavant : l'utilisateur ne voit pas la différence... du moins dans cette application... car si vous aviez du ouvrir un formulaire dont le temps de chargement est non négligeable, mieux vaut ne le charger qu'une seule fois et choisir ensuite de l'afficher ou pas.

Chapitre 4

Les contrôles personnalisés

Tout cela est bien gentil mais cela commence à devenir rudement complexe à gérer si l'application comporte sur un même formulaire des dizaines (voire des centaines) de contrôles et qui plus est, ont, malgré vos efforts, tous des noms relativement proches. Il devient difficile de s'y retrouver. Pire ! L'EDI commence à ramer sérieusement lorsque vous lui faites afficher le formulaire en mode Designer.

Imaginez par exemple que vous vouliez réaliser un jeu de mots croisés. Comment créer son interface si ce n'est qu'avec des centaines de `TextBox` ?

La solution à ce problème se nomme les contrôles utilisateurs dont nous allons voir comment en tirer toute la puissance.

Application

Nous n'allons pas faire de mots croisés car cela compliquerait sérieusement le travail à faire.

Voilà notre but :

Chacun des `GroupBox` (encadré en rouge) se trouve dans un contrôle utilisateur différent (nous avons donc trois contrôles utilisateurs à créer).

Comportement :

- La **GroupBox** Mes amis est désactivée tant qu'un contact n'a pas été ajouté à la base de données.
- La **GroupBox** Photo reste désactivée tant qu'un ami n'est pas sélectionné dans la liste des amis du **GroupBox** Mes amis.
- Un contact est ajouté à la base de données lors que le bouton Accepter est cliqué. Le contact est invisible par défaut pour l'utilisateur.
- Lorsque l'utilisateur appuie sur le bouton Rechercher, le ou les contacts dont les noms ou prénoms contiennent le texte marqué dans la **TextBox** de Nom/prénom sont ajoutés à la **ComboBox**. Si la **TextBox** de Nom/prénom est vide, tous les contacts sont ajoutés dans la **ListBox**.
- Lorsqu'un contact est sélectionné dans la **ComboBox**, les boutons Ajouter et Details deviennent cliquables.
- En cliquant sur Ajouter, le contact est ajouté à la liste des amis.
- Si un contact est sélectionné dans la liste des amis, le bouton Details devient cliquable et la **GroupBox** Photo est activée.
- Si le bouton Details est cliqué, la **GroupBox** Contact est remplie par les données du contact en question mais la **GroupBox** est dans un état qui empêche l'utilisateur de modifier le contact. Pour entrer un nouveau contact, il faut absolument désélectionner (perte de focus) le contact en question.
- A l'activation de la **GroupBox** Photo, si l'ami a déjà une photo, celle-ci est immédiatement affichée, sinon, rien n'est affiché.
- En cliquant sur Modifier, une boîte de dialogue d'ouverture de fichier apparaît et l'utilisateur peut sélectionner un fichier de type JPEG ou JPG uniquement pour ajouter l'image choisie en tant que photo de l'ami.

Ca commence à se compliquer un peu...

Essayez de faire seul cette application. Si vous y parvenez, bravo ! Sinon, pas grave, c'est normal, c'est déjà assez difficile à faire comme programme... surtout si vous débutez.

Que vous ayez réussi ou non, la suite devrait certainement vous intéresser...

La base de données

Il est préférable de commencer par mettre en place la base de données avant de s'attaquer à l'interface et plus particulièrement aux contrôles personnalisés.

Ici, nous allons simplement créer une classe **Contact** capable de recevoir toutes les données que nous sommes en mesure de lui fournir (ne pas oublier la photo).

L'ensemble des contacts sera géré en interne par une classe générique : **List**.

*Remarque : Les génériques n'étant disponibles que sous C# 2.0 ou version ultérieure, vous ne pourrez en faire usage en C# 1.1 et vous ne pourrez donc pas utiliser la classe générique **List** en C# 1.1. Pas d'inquiétude, vous pouvez utiliser la classe **ArrayList** à la place. Le code pour **ArrayList** sera également fourni...*

Voyons un peu le code de cette classe **Contact** :

```
public class Contact
{
    //Champs
    private string name;
    private string firstName;
    private string adresse;
    private string email;
    private string telfixe;
    private string telmob;
    private bool isFriend;
```

```

private byte[] photo;

//Constructeur
public Contact(
    string name, string firstName,
    string adresse, string email,
    string telephoneFixe, string telephoneMobile)
{
    this.name = name;
    this.firstName = firstName;
    this.adresse = adresse;
    this.email = email;
    this.telfixe = telephoneFixe;
    this.telmob = telephoneMobile;
    this.isFriend = false;
}

//Propriétés
public string Name
{
    get { return name; }
}

public string FirstName
{
    get { return firstName; }
}

public string Adresse
{
    get { return adresse; }
}

public string Email
{
    get { return email; }
}

public string TelephoneFixe
{
    get { return telfixe; }
}

public string TelephoneMobile
{
    get { return telmob; }
}

public bool IsFriend
{
    get { return isFriend; }
    set { isFriend = value; }
}

public byte[] Photo
{
    get { return photo; }
    set { photo = value; }
}
}

```

Rien de difficile dans cette classe... on ne fait que stocker les données et les récupérer...

Remarque : Pensez à créer un nouveau fichier dédié à cette classe *Contact* et nommez le *Contact.cs*.

Allez à présent sur le formulaire principal et modifiez en le nom du fichier, tapez : *MainForm.cs* et allez immédiatement dans le code source.

Il faut créer cette fameuse liste de contacts. Pour cela, ajoutez le champ suivant dans la classe *MainForm*...

```
List<Contact> myContacts;
```

... et instanciez le dans le constructeur...

```
public MainForm()  
{  
    InitializeComponent();  
    myContacts = new List<Contact>();  
}
```

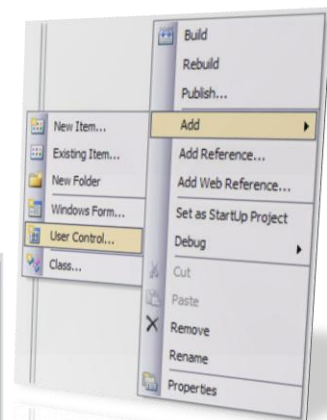
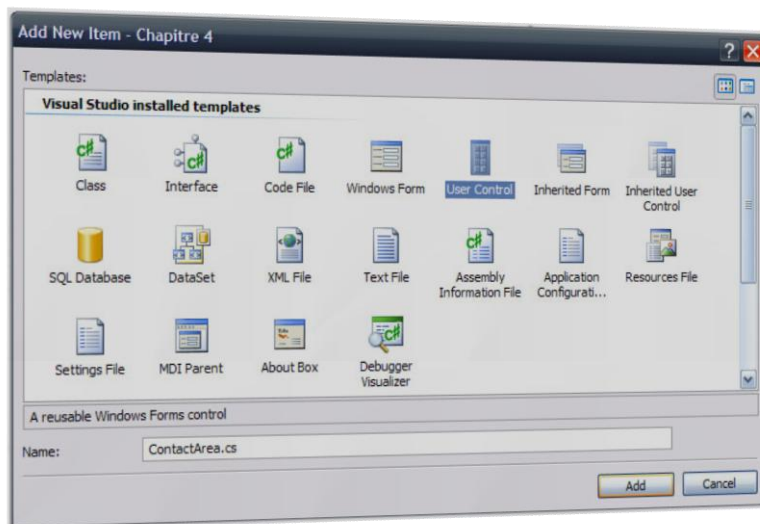
Voilà, on peut à présent s'occuper de l'interface graphique... Pour ajouter un contact à cette liste, il suffit d'utiliser la méthode membre de la classe *List* : *Add()*.

Note : Comme signalé précédemment, le code précédent n'est valable que sous C# 2.0 ou une version ultérieure. Sous C# 1.1, il suffit de remplacer *List<Contact>* par *ArrayList*. *ArrayList* contient de même une méthode *Add()* pour y ajouter des éléments.

User Control

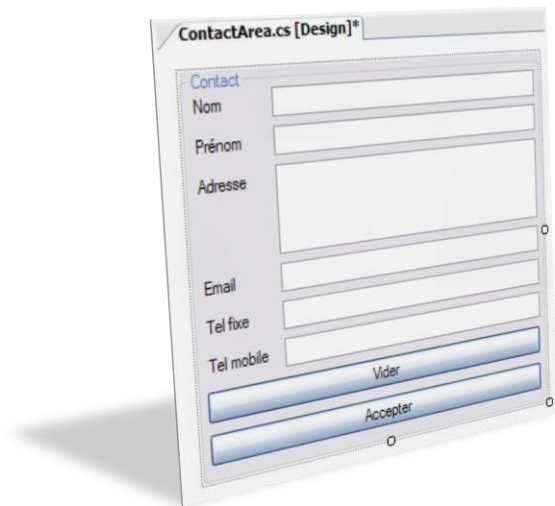
Commencez par ajouter un User Control à votre projet...

Nommez le *ContactArea.cs* :



Vous avez alors à votre disposition un nouveau support de dessin, similaire à celui d'un formulaire mais sans les bordures de fenêtres...

La zone encadrée correspond à votre contrôle. Vous pouvez la redimensionner...



Placez-y les éléments de l'interface de la **GroupBox** Contact...

Redimensionnez le contrôle personnalisé comme vous le voulez pour faciliter le placement des contrôles... Définissez la propriété Dock de la **GroupBox** sur Fill pour que le contrôle personnalisé et la **GroupBox** ne fassent qu'un.

Remarques :

- Lorsque vous réglez la propriété Dock d'un contrôle sur Fill, vous pouvez par la suite avoir du mal à accéder au contrôle parent (ici, il s'agit du support même du contrôle personnalisé). Pour cela, faites un clic droit dans le tas... une petite liste des contrôles accessibles s'affichent alors, sélectionnez celui qui vous intéresse (dans notre cas, ça sera 'Select ContactArea').
- Pensez aux ancres !!!

Occupez vous de l'évènement associé au clic sur le bouton Vider...

Remarque : Etant donné que nous allons également vider tous les champs lorsque le contact sera ajouté à la liste, il est largement conseillé de créer une méthode spécialement dédiée à cette tâche. Il suffira alors d'appeler cette méthode aux moments où l'on en aura besoin. Cela permet de ne pas avoir à réécrire plusieurs fois le même code dans différentes méthodes.

Voyons par contre ce qu'il est possible de faire pour le bouton Accepter...

Lorsque le bouton Accepter est cliqué, on veut qu'un contact soit ajouté à notre liste. Le problème, c'est que le formulaire principal n'a pas et ne doit pas avoir accès au bouton Accepter puisque celui-ci appartient au contrôle utilisateur (et pas au formulaire principal comme s'était le cas dans les applications précédentes).

Pour que le formulaire principal ait connaissance du contenu du contrôle personnalisé, il y a plusieurs solutions mais on peut résumer en disant qu'il faut simplement que le formulaire principal ait accès à ces données par des propriétés (publiques) personnalisées du contrôle utilisateur.

Dans tous les cas, nous voulons ajouter le contact après que le bouton Accepter ait été cliqué. Un clic correspondant à un évènement. Il faudra de même être en mesure de signaler au formulaire principal que le bouton Accepter a été cliqué... mais celui-ci n'ayant pas accès au bouton en question, il va également falloir créer nos propres évènements pour cet User Control.

Commençons par le plus simple : occupons nous d'abord des propriétés.

Il y a différentes façons de procéder :

- Soit on crée une propriété pour chaque champ (cela nous en ferait six en tout)
- Soit on se débrouille pour que toutes les informations soient passées en une seule fois, avec une seule propriété. Cela doit toutes fois rester simple et compréhensible pour quelqu'un qui n'a pas conçu lui-même le contrôle utilisateur.

Ici nous allons pouvoir appliquer la deuxième méthode car nous avons à notre disposition une classe **Contact** qui a été spécialement créée pour cela.

Il suffit finalement de créer une unique propriété renvoyant une instance de **Contact** contenant tous les éléments du contrôle utilisateur.

Simple :

```
public Contact Contact
{
    get
    {
        return new Contact(
            textBoxName.Text,
            textBoxFirstName.Text,
            textBoxAddress.Text,
            textBoxEmail.Text,
            textBoxTelHome.Text,
            textBoxTelMob.Text
        );
    }
}
```

Pensez également que le formulaire principal devra être capable de demander à un contact de s'afficher dans le contrôle personnalisé. Il suffit pour cela d'ajouter l'accesseur **set** à la propriété précédente...

```
set
{
    textBoxName.Text = value.Name;
    textBoxFirstName.Text = value.FirstName;
    textBoxAddress.Text = value.Adresse;
    textBoxEmail.Text = value.Email;
    textBoxTelHome.Text = value.TelephoneFixe;
    textBoxTelMob.Text = value.TelephoneMobile;
}
```

C'est tout ce qu'il y a à faire pour les propriétés. A présent notre User Control est bien équipé pour communiquer convenablement avec n'importe quel contrôle parent.

Reste à avertir le contrôle parent du moment où le bouton Accepter sera cliqué.

Evènements personnalisés

Pour cela il faut créer notre propre évènement.

Voyons d'abord comment on accède aux évènements déjà existant...

```
button2.Click += new EventHandler(button2_Click);

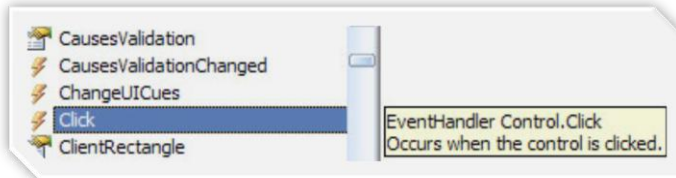
//Plus loin dans le code

//La méthode button2_Click
private void button2_Click(object sender, EventArgs e)
{
}
```

Mais au fait, pourquoi ne pourrions nous pas mettre simplement la méthode `button2_Click` sans paramètre ?

```
//Pourquoi pas ?
private void button2_Click()
{
}
}
```

Alors, qu'est ce qui nous en empêche ?



En fait, l'évènement Click est de type **EventHandler** et **EventHandler** prend en paramètre (*target*) une méthode ne renvoyant rien (type **void**) et prenant elle-même en paramètre, dans cet ordre, un **object** et un **EventArgs**.

Nous pouvons donc créer n'importe quelle méthode, avec n'importe quel nom et la passer en paramètre au **EventHandler**, à la condition que cette méthode prenne en paramètre un type **object** puis un type **EventArgs**.

```
new EventHandler(
EventHandler.EventHandler (void (object, EventArgs) target)
EventHandler.EventHandler (void (object, EventArgs) target))
```

Nous avons donc besoin de créer un évènement qui soit publique, qui indique que le bouton Accepter a été cliqué et qui soit de type **EventHandler** car nous n'allons en fin de compte faire que « transférer » l'évènement Click du bouton Accepter depuis l'User Control vers le contrôle parent (ici, le formulaire principal) car nous n'avons pas d'informations à ajouter à l'évènement...

Placez, de la même manière que pour un champ, la ligne de code suivante :

```
public event EventHandler AcceptButtonClick;
```

Nous avons bien ici ajouté à notre classe **ContactArea** un évènement (**event**) de type **EventHandler** et qui est **public** et qui indique, pour l'instant par son nom, qu'il sera déclenché au moment où le bouton Accepter sera cliqué.

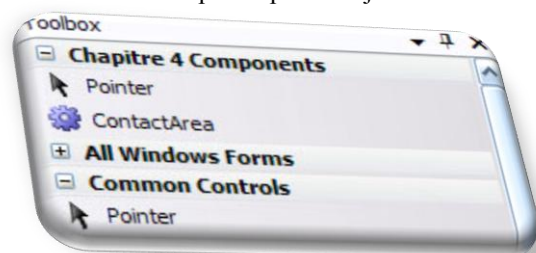
Il ne nous reste plus qu'à le déclencher au bon moment. Allez dans la méthode associée au clic sur le bouton Accepter et placez-y les lignes suivantes :

```
if (AcceptButtonClick != null)
    AcceptButtonClick(sender, e);
```

Rappelons que **AcceptButtonClick** est de type **EventHandler** et que **EventHandler** prend en paramètre une méthode lors de sa construction ! Lorsqu'on veut utiliser **AcceptButtonClick**, il faut donc d'abord vérifier que celui-ci pointe effectivement vers une méthode. Si ce n'est pas le cas, il vaut **null**. Sinon, on peut « appeler » **AcceptButtonClick**, ce qui revient en fait à appeler la méthode vers laquelle pointe **AcceptButtonClick** via le **EventHandler**.

*Remarque : **EventHandler** pointe sur une méthode car il s'agit en fait d'un délégué. Nous n'aborderons pas ici leur fonctionnement et nous nous limiterons à leur utilisation via les évènements, ce qui est beaucoup plus intuitif et donc simple à comprendre. Retenez simplement qu'un délégué est un « pointeur de méthode ».*

C'est tout... On peut à présent ajouter le nouveau User Control au formulaire principal...



Retournez sur le formulaire principal et regardez dans la Toolbox. Il devrait y avoir un nouveau contrôle portant le nom **ContactArea** et représenté par une icône d'engrenage tout en haut... S'il n'y est pas, reconstruisez (*Build* → *Build Solution* ou F6) le projet. S'il n'y a pas d'erreurs, il devrait à présent être visible...

Ne vous reste donc plus qu'à l'ajouter à votre formulaire comme n'importe quel autre contrôle.

Remarquez dès à présent que votre contrôle personnalisé se comporte dans le Designer comme n'importe quel autre contrôle que vous avez l'habitude de manipuler...

Allons à présent nous occuper de cette liste de contacts.

Commencez par créer la méthode associée à l'évènement `AcceptButtonClick`. Attention, ici pas de double clic sur le contrôle, il va falloir aller faire un petit tour dans le code... Cette méthode prend en paramètre, rappelez-vous, un type `object` et un type `EventArgs`...

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

Ajoutez ensuite cette méthode à l'évènement `AcceptButtonClick` dans le constructeur du formulaire principal.

```
public MainForm()
{
    InitializeComponent();
    myContacts = new List<Contact>();
    contactArea1.AcceptButtonClick += new EventHandler(addContact);
}
```

Il ne reste plus qu'à ajouter les instructions à la méthode `addContact` pour que celle-ci récupère les informations contenues dans le contrôle `ContactArea` et les ajoute à la liste `myContacts`. Rien de bien difficile une fois de plus puisque nous avons codé le contrôle `ContactArea` de manière à ce que celui-ci puisse directement nous renvoyer un objet de type `Contact` contenant toutes ces informations. La méthode ressemble donc à quelque chose du genre :

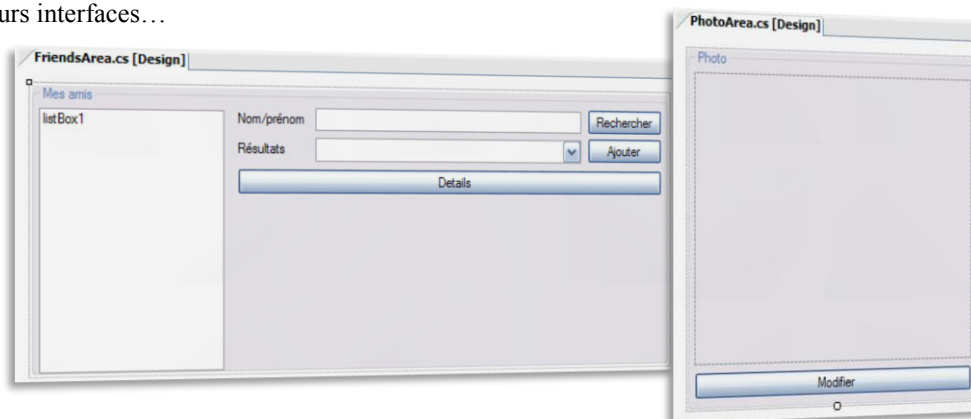
```
private void addContact(object sender, EventArgs e)
{
    myContacts.Add(contactArea1.Contact);
    //Sans oublier de nettoyer tous les champs du contrôle
    contactArea1.Clear();
}
}
```

Vous pouvez vérifier, tout marche comme prévu, l'évènement envoyé depuis le contrôle personnalisé est bien intercepté par le formulaire et les champs sont vidés depuis ce dernier...

Passons à la suite à présent...

La suite...

Ne perdez pas de temps, créez immédiatement les deux autres contrôles personnalisés et occupez vous de leurs interfaces...



Nommez les `FriendsArea` et `PhotoArea`.

Traitons-les dans l'ordre logique du programme : commençons par le contrôle `FriendsArea`.

Rappelons-nous ce qu'il y a à faire pour ce contrôle :

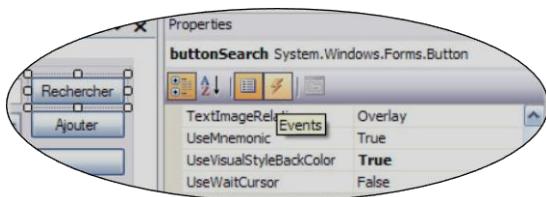
- Lorsque l'utilisateur appuie sur le bouton Rechercher, le ou les contacts dont les noms ou prénoms contiennent le texte marqué dans la `TextBox` de Nom/prénom sont ajoutés à la `ComboBox`. Si la `TextBox` de Nom/prénom est vide, tous les contacts sont ajoutés dans la `ListBox`.
- Lorsqu'un contact est sélectionné dans la `ComboBox`, les boutons Ajouter et Details deviennent cliquables.
- En cliquant sur Ajouter, le contact est ajouté à la liste des amis.
- Si un contact est sélectionné dans la liste des amis, le bouton Details devient cliquable et la `GroupBox` Photo est activée.
- Si le bouton Details est cliqué, la `GroupBox` Contact est remplie par les données du contact en question mais la `GroupBox` est dans un état qui empêche l'utilisateur de modifier le contact. Pour entrer un nouveau contact, il faut absolument désélectionner (perte de focus) le contact en question.

Commençons par la recherche... Pour effectuer une recherche, deux possibilités : on la fait en interne, dans le User Control, ou dans le formulaire. Ces deux cas sont possibles et donnent un contrôle utilisateur plus ou moins facile à mettre en place et plus ou moins personnalisable depuis « l'extérieur » du contrôle. Nous allons la faire en interne pour simplifier...

Le problème, c'est que le contrôle n'a pas à disposition la liste de contacts. Qu'à cela ne tienne, il suffit de la lui passer. Créez pour cela une propriété qui permet de définir la liste de contacts pour le contrôle personnalisé. N'oubliez pas de créer le champ dans ce contrôle au passage...

```
List<Contact> myList = new List<Contact>();  
  
//Plus loin dans le code...  
  
public List<Contact> List  
{  
    set { myList = value; }  
}
```

Maintenant que nous avons la liste, nous n'avons plus qu'à lancer la recherche lorsque le bouton Rechercher est cliqué. Pas de double clic sur le bouton, on va faire autrement pour changer...



Sélectionnez le bouton Rechercher puis allez dans l'inspecteur de propriétés. Tout en haut de celui-ci, vous avez une icône en forme d'éclair...

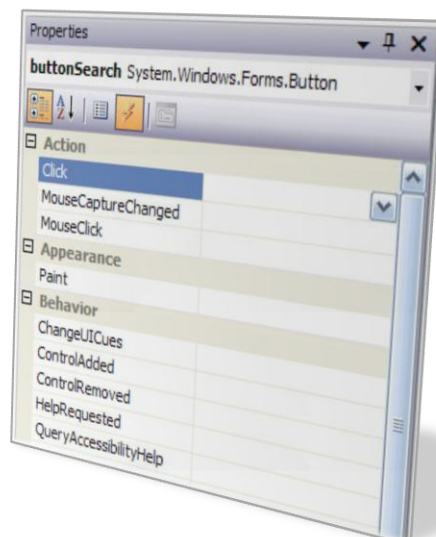
Cliquez dessus pour basculer sur la liste des événements du bouton Rechercher...

S'offre alors à vous une grande liste des événements du bouton Rechercher, tous accessibles en un clic !

Pour créer un événement, n'importe lequel de cette liste, effectuez un double clic dans la case vide à droite de l'événement que vous voulez créer.

Une méthode est alors automatiquement créée dans le code.

C'est donc aussi facile que le simple double clic sur le bouton pour créer un événement Click, à la différence qu'ici vous avez beaucoup plus d'événements sous la main...



Rendez vous dans la méthode associée au clic sur le bouton Rechercher.

Il faut à présent effectuer la recherche dans cette méthode... sans oublier que la liste n'a peut-être pas été passé au contrôle, peut-être qu'elle vaut toujours `null`...

```
comboBoxResults.Items.Clear();
//Ajouter un élément neutre
comboBoxResults.Items.Add("");
if (myList != null)
{
    //Pour chaque contact de la liste...
    foreach (Contact c in myList)
    {
        /* ... si celui-ci a son nom ou prénom de tapé dans la
         * TextBox de recherche... */
        if (textBoxSearch.Text.ToLower().Contains(c.Name.ToLower()) ||
            textBoxSearch.Text.ToLower().Contains(c.FirstName.ToLower()) ||
            textBoxSearch.Text == "")
        {
            //... on l'ajoute à la ComboBox
            comboBoxResults.Items.Add(c.Name + " " + c.FirstName);
        }
    }
}
```

Occupons nous maintenant de la **ComboBox**... Lorsqu'un élément est sélectionné, si cet élément n'est pas l'élément neutre, on doit activer le bouton Ajouter et le bouton Details. Il faut pour cela définir l'évènement `SelectedIndexChanged` (utilisez l'inspecteur de propriétés comme montré précédemment pour créer la méthode) et valider ou non la propriété `Enabled` des deux boutons cités.

***Remarque :** Si vous effectuez un double clic sur la **ComboBox**, vous n'allez non pas créer l'évènement `Click` qui lui est associée mais créer l'évènement `SelectedIndexChanged` qui est l'évènement par défaut de ce contrôle (alors que pour un bouton, l'évènement par défaut est `Click`).*

```
private void comboBoxResults_SelectedIndexChanged(object sender, EventArgs e)
{
    bool activer = comboBoxResults.SelectedItem.ToString() == "" ? false : true;
    buttonAdd.Enabled = activer;
    buttonDetails.Enabled = activer;
}
```

Rappel : Les codes suivants sont équivalents :

- `<type> variable = condition ? valeur1 : valeur2;`
- `<type> variable;`
`if (condition)`
`variable = valeur1;`
`else`
`variable = valeur2;`

Pensez alors à désactiver par défaut les deux boutons Ajouter et Details en passant par l'inspecteur de propriétés et en réglant leurs propriétés `Enabled` sur `false` (sélectionnez les deux boutons pour aller plus vite).

Occupons nous à présent du bouton Ajouter...

Lorsque l'utilisateur clique sur ce bouton, il suffira d'ajouter le texte qui se trouve dans la **ComboBox** dans la liste et de modifier la propriété `IsFriend` du contact sélectionné pour lui donner la valeur `true`. N'oubliez pas qu'il faut également vider la **ComboBox** (items et texte) et la **TextBox** ainsi que désactiver les boutons Ajouter et Details...

Remarques :

- Il est inutile de prévoir le cas dans le bouton Ajouter où la **ComboBox** sera sélectionné sur l'élément neutre puisque cela n'arrivera jamais ! En effet, le bouton sera alors inactif et l'évènement Click rendu inaccessible.
- Vu que l'on ajoute que les noms et prénoms du contact dans la **ComboBox**, on n'a pas directement accès à celui-ci et il va falloir effectuer une nouvelle recherche pour le retrouver et modifier sa propriété IsFriend. En fait, on pourrait se passer de cette recherche en sachant que les items de la **ComboBox** ne sont non pas de type **string** mais de type **object** ! On pourrait donc passer l'instance de **Contact** toute entière lors de l'ajout de l'item mais il y aurait des surcharges de méthodes à faire (juste une en fait) pour que l'affichage du contact soit bon. Nous verrons ce cas tout à la fin de ce chapitre.
- Le même problème sera rencontré par la suite pour récupérer un contact de la **ListBox** : il faudra encore effectuer une recherche.
- En attendant la fin du chapitre, comme nous aurons besoin plusieurs fois d'un code très similaire dans différentes méthodes, nous allons coder une méthode privée générale permettant à partir d'un texte de la forme nom, espace, prénom de récupérer le contact qui possède exactement les mêmes noms et prénoms (le cas où plusieurs contacts auraient mêmes noms et prénoms ne sera pas traité mais sera englobé dans le code vu qu'on s'arrêtera au premier contact valide rencontré).

Suffit pour les remarques, voici le code de la méthode générale de recherche précise de contact :

```
private Contact getContact(string description)
{
    if (myList != null && description != null)
        foreach (Contact c in myList)
            if (c.Name + " " + c.FirstName == description)
                return c;

    return null;
}
```

Et voici le code à exécuter lorsque le bouton Ajouter est cliqué :

```
listBoxFriends.Items.Add(comboBoxResults.SelectedItem.ToString());
Contact cct = getContact(comboBoxResults.SelectedItem.ToString());
cct.IsFriend = true;

textBoxSearch.Text = "";
comboBoxResults.Items.Clear();
comboBoxResults.Text = "";
buttonAdd.Enabled = false;
buttonDetails.Enabled = false;
```

Occupons nous du bouton Details... Ce bouton n'est actif que si un contact est sélectionné dans la **ComboBox** (ça, c'est fait) ou dans la **ListBox** (pas fait).

Petit détour par la **ListBox** :

```
private void listBoxFriends_SelectedIndexChanged(object sender, EventArgs e)
{
    buttonDetails.Enabled = listBoxFriends.SelectedItems.Count == 1 ? true : false;
}
```

Profitez-en pour interdire la sélection multiple dans la **ListBox** via la propriété SelectionMode (à régler sur One bien entendu).

Voyons maintenant ce bouton Details... Lorsque celui-ci est cliqué, il doit d'abord avertir le contrôle parent qu'un contact doit être affiché et c'est ensuite le formulaire principal qui a pour tâche d'afficher (comme il peut) le contact sélectionné.

Nous avons donc besoin de deux choses, les mêmes que pour le contrôle précédent :

- Une propriété retournant le contact sélectionné
- Un évènement pour signaler que le bouton Details a été cliqué

Comme précédemment, définissez la propriété renvoyant le contact...

```
public Contact SelectedContact
{
    get
    {
        string description = null;
        //Les amis d'abord
        if (listBoxFriends.SelectedItems == 1)
            description = listBoxFriends.SelectedItem.ToString();
        //il y a forcément un élément de sélectionné dans la ComboBox
        else
            description = comboBoxResults.SelectedItem.ToString();

        //Dans tous les cas, on va appeler la méthode getContact...
        return getContact(description);
        //...qui renvoie directement un Contact
    }
}
```

... puis l'évènement personnalisé...

```
List<Contact> myList = null;
public event EventHandler DetailsButtonClick;
```

... reste à déclencher l'évènement au moment opportun...

```
private void buttonDetails_Click(object sender, EventArgs e)
{
    if (DetailsButtonClick != null)
        DetailsButtonClick(sender, e);
}
```

... et le contrôle est alors prêt à être ajouté au formulaire principal.

Mais nous allons encore le perfectionner...

Les attributs

Mais que reste-t-il à perfectionner ? Eh bien comparons un peu notre contrôle personnalisé à ceux fournis par défaut, comme la [TextBox](#) par exemple...

Focalisons-nous surtout sur ce que notre contrôle n'a pas, sur ce qui lui manque par rapport aux autres :

- Les propriétés définies dans le code de notre contrôle ne sont pas triées dans l'inspecteur de propriété de Visual Studio (mais elles sont présentes mais dans la catégorie Misc), elles n'ont pas de valeurs par défaut, de description...
- De même pour les évènements personnalisés.
- En faisant un double clic sur le contrôle, une méthode associée à l'évènement Load serait créée alors que ce n'est pas l'évènement intéressant de notre contrôle. Il faudrait que cela soit l'évènement DetailsButtonClick qui soit l'évènement par défaut.
- Il est impossible de donner des valeurs par défaut au contrôle, comme par exemple, remplir la [ListBox](#) des amis avec des contacts sans que l'utilisateur clique sur le bouton Ajouter.

Tous ces défauts peuvent être bien sûr corrigés en utilisant les attributs.

Les attributs sont des sortes de balises que l'on place entre crochets devant une propriété, un évènement, une classe, etc. Elles ne jouent aucun rôle lors de l'exécution du programme mais permettent de fournir des informations sur la classe ou la fonction lors de l'édition ou de la compilation du programme.

Nous aurons besoins de quelques attributs pour modifier notre contrôle. Voici ceux que nous étudierons ici :

- **BrowsableAttribute** : spécifie si la propriété doit être affichée ou non dans l'inspecteur de propriétés.
- **CategoryAttribute** : attribue une catégorie à la propriété ou à l'évènement (exemples de catégories : Design, Behavior, Data...)
- **DefaultEventAttribute** : attribue l'évènement par défaut associée à une classe
- **DefaultValueAttribute** : attribue une valeur par défaut à la propriété
- **DescriptionAttribute** : donne une description de la propriété ou de l'évènement
- **DesignerSerializationVisibilityAttribute** : pour gérer des collections d'objets

Tous ces attributs se trouvent dans l'espace de nom System.ComponentModel. Pensez donc à vérifier que cet espace de nom est bien déclaré dans votre code.

Remarque : La suffixe **Attribute** est facultatif. Par exemple, vous pouvez utiliser l'attribut de catégorie en tapant **Category** ou bien **CategoryAttribute**. Nous omettrons par la suite ce suffixe.

Ajoutez le contrôle **FriendsArea** au formulaire principal mais avant de faire quoi que ce soit d'autre, retournez dans le code source de ce contrôle personnalisé.

Occupons nous de l'évènement **DetailsButtonClick**... Nous voulons le placer dans la catégorie "**Action**" de « l'inspecteur d'évènements » et lui donner la description suivante : "**Occurs when the Details button is clicked.**". Pour cela, rien de difficile, repérez la ligne où vous avez déclaré l'évènement en question et ajoutez lui les attributs suivants :

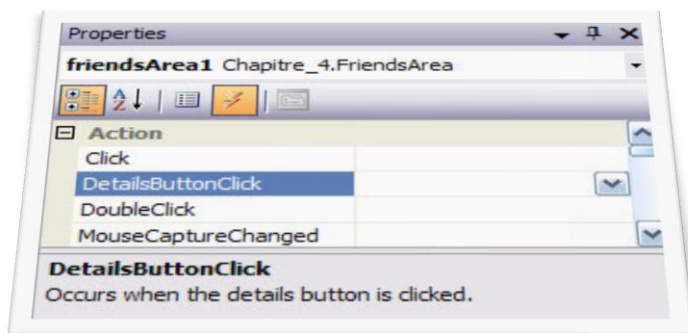
```
[Category("Action")]
[Description("Occurs when the Details button is clicked.")]
```

Vous pouvez également mettre tous les attributs dans un même crochet en les séparant par des virgules :

```
[Category("Action"), Description("Occurs when...")]
```

Cela donne donc pour notre évènement :

```
[Category("Action")]
[Description("Occurs when the Details button is clicked.")]
public event EventHandler DetailsButtonClick;
```



Vérifiez !

Si vous retournez à présent sur le formulaire principal, que vous sélectionnez le contrôle **FriendsArea** et que vous allez dans la zone des évènements de l'inspecteur de propriétés, vous pourrez voir dans la catégorie **Action** votre évènement avec la description souhaitée...

Nous voulons également que cet évènement soit l'évènement par défaut du contrôle. Pour ce faire, placez l'attribut **DefaultEvent** devant la classe du contrôle en précisant le nom de l'évènement par défaut :

```
[DefaultEvent("DetailsButtonClick")]
public partial class FriendsArea : UserControl
```

Reconstruisez le projet et vérifiez : le double clic sur le contrôle **FriendsArea** crée à présent la méthode associée à l'évènement DetailsButtonClick.

Nous voulons aussi qu'il soit possible d'ajouter depuis l'inspecteur de propriétés une liste d'amis. Vu que les amis sont forcément parmi les contacts, il faut que l'on puisse ajouter des contacts (nom, prénom, adresse...) et préciser si ceux-ci sont des amis ou pas...

Nous avons déjà codé une propriété permettant d'établir la liste des contacts dans le contrôle :

```
public List<Contact> List
{
    set { myList = value; }
}
```

Mais il s'avère que cette définition est doublement insuffisante. En effet, si on ajoute une liste de contacts qui contient des amis en passant par cette propriété, la liste ne sera pas filtrée et les amis ne seront pas affichés dans la **ListBox**.

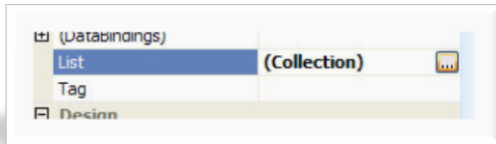
Il faut donc effectuer une première modification :

```
public List<Contact> List
{
    set
    {
        myList = value;
        foreach (Contact c in myList)
            if (c.IsFriend)
                listBoxFriends.Items.Add(c.Name + " " + c.FirstName);
    }
}
```

Maintenant que les amis sont ajoutés à la **ListBox**, il faut pouvoir afficher cette propriété dans l'inspecteur de propriétés. Mais pas d'affichage si la propriété ne peut pas être lu (logique). Il faut donc ajouter un accesseur **get** à cette propriété. Ajoutons au passage une petite description et classons la dans une catégorie.

```
[Category("Data")]
[Description("The contacts in the list")]
public List<Contact> List
{
    get { return myList; }
    set
    {
        myList = value;
        foreach (Contact c in myList)
            if (c.IsFriend)
                listBoxFriends.Items.Add(c.Name + " " + c.FirstName);
    }
}
```

Reconstruisez le projet et allez voir dans la catégorie Data du contrôle **FriendsArea** qui se trouve sur le formulaire principal.



Attention : Pour pouvoir ajouter des éléments dans une collection, il faut impérativement que le mot (**Collection**) soit inscrit dans la case à droite sinon, cela signifie que votre collection est **null** et vous n'arriverez rien à y mettre !

Cliquez sur le petit bouton qui apparaît dans la partie droite... Une fenêtre s'ouvre et vous propose d'ajouter des éléments à la liste de contacts. Super !

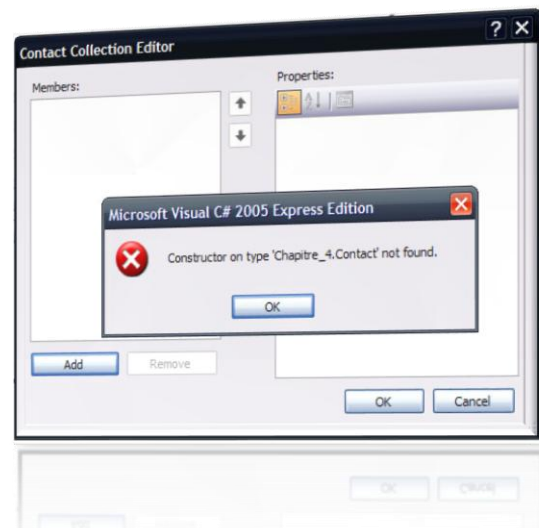
Cliquez sur le bouton Ajouter et ... Visual Studio vous crache un message à la figure.

Le message est très clair : le constructeur de notre classe **Contact** n'a pas été trouvé. Pourtant il existe, mais uniquement avec des paramètres.

Visual Studio a besoin d'un constructeur sans paramètre pour pouvoir créer un objet via n'importe quelle interface. Il suffit donc d'ajouter à notre classe **Contact** un constructeur sans paramètre :

```
public Contact()
{
}
```

Et maintenant ça marche !



Vous pouvez ajouter autant de contacts que vous le voulez. A chaque contact ajouté, vous pouvez définir chacune des propriétés du contact... enfin presque... celles-ci sont en effet grisées, impossible de modifier ces propriétés.

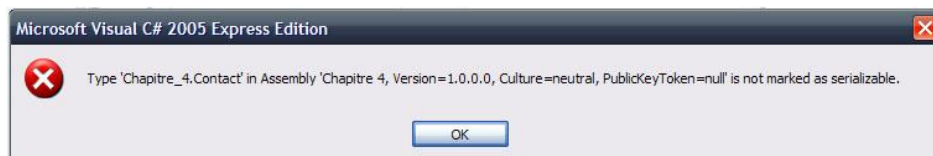
Et c'est parfaitement normal puisque celles-ci sont en lecture seule.

```
public string Name
{
    get { return name; }
}
```

Modifiez les toutes de manière à ce qu'elles puissent définir chacun des champs de la classe **Contact**.

```
public string Name
{
    get { return name; }
    set { name = value; }
}
```

Mais ce n'est toujours pas fini (cela serait-il donc sans fin ?) car si vous tentez d'ajouter un **Contact** maintenant, vous aurez une insulte d'un nouveau genre :

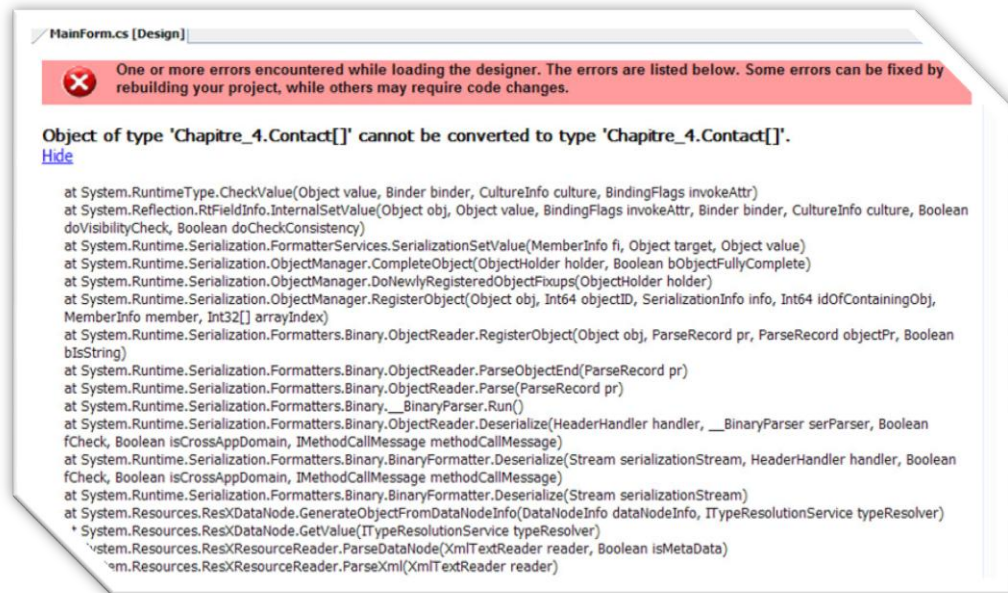


Et pour résoudre ce dernier problème, il faut, comme indiqué, marquer la classe **Contact** avec l'attribut **Serializable**.


```
[Serializable]
public class Contact
```

Ajoutez à présent un ami à cette fameuse liste, qui nous a posé tant de problèmes jusque là (ce n'est malheureusement pas fini).

Reconstruisez le projet et ... boum ! Une nouvelle erreur :



Apparemment l'éditeur n'aime pas convertir des listes de contact en tableau de contacts et vice-versa. On va donc se simplifier le travail et demander à notre propriété List de ne renvoyer qu'un tableau plutôt qu'une liste.

```
[Category("Data")]
[Description("The contacts in the list")]
public Contact[] List
{
    get { return myList.ToArray(); }
    set
    {
        myList = new List<Contact>(value);
        foreach (Contact c in myList)
            if (c.IsFriend)
                listBoxFriends.Items.Add(c.Name + " " + c.FirstName);
    }
}
```

Reconstruisez le projet et corrigez les erreurs et les avertissements en supprimant le code problématique.

Ici le code qui pose problème est lié au fichier ressource du formulaire principal. Allez dans le fichier MainForm.Designer.cs et supprimez donc la ligne :

```
this.friendsArea1.List = ((System.Collections.Generic.List<Chapi ...
```

Allez également dans le fichier ressource MainForm.resx et supprimez la ressource ayant un lien avec le contrôle **FriendsArea**.

Other Add Resource Remove Resource			
Name	Type	Value	Comment
contactArea1.Contact	Chapitre_4.Contact	(Value cannot be displayed)	
friendsArea1.List	(Nothing/null)	(Nothing/null)	

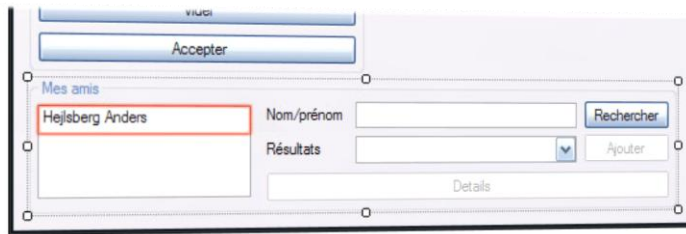
A présent, la reconstruction du projet fonctionne correctement et vous pouvez ajouter un ami à votre liste d'amis sans que la formulaire principal ne parte en vrille...

Ajoutez donc un ami (pas seulement un contact : un ami !) à la liste des contacts depuis l'inspecteur de propriétés.

L'ami est bien ajouté dans la liste mais vous ne le voyez pas s'afficher dans la [ListBox](#) comme il devrait le faire lorsque le programme est exécuté.

On peut faire cela en ajoutant encore un attribut (encore un) devant la propriété List : il s'agit de [DesignerSerializationVisibility](#)...

`[DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)]`



Si vous retournez maintenant sur le formulaire principal, vous pouvez voir que votre ami (ou vos amis si vous en avez mis plusieurs dans la liste) s'affiche dans la [ListBox](#).

Finalement notre propriété ressemble à quelque chose du genre :

```
[Category("Data")]
[Description("The contacts in the list")]
[DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)]
public Contact[] List
{
    get { return myList.ToArray(); }
    set
    {
        myList = new List<Contact>(value);
        foreach (Contact c in myList)
            if (c.IsFriend)
                listBoxFriends.Items.Add(c.Name + " " + c.FirstName);
    }
}
```

Voyons au passage comment utiliser les attributs [DefaultValue](#) et [Browsable](#). Nous allons ajouter ces attributs aux propriétés de notre classe [Contact](#) et voir ce qu'ils font.

Par exemple, modifiez les propriétés suivantes...

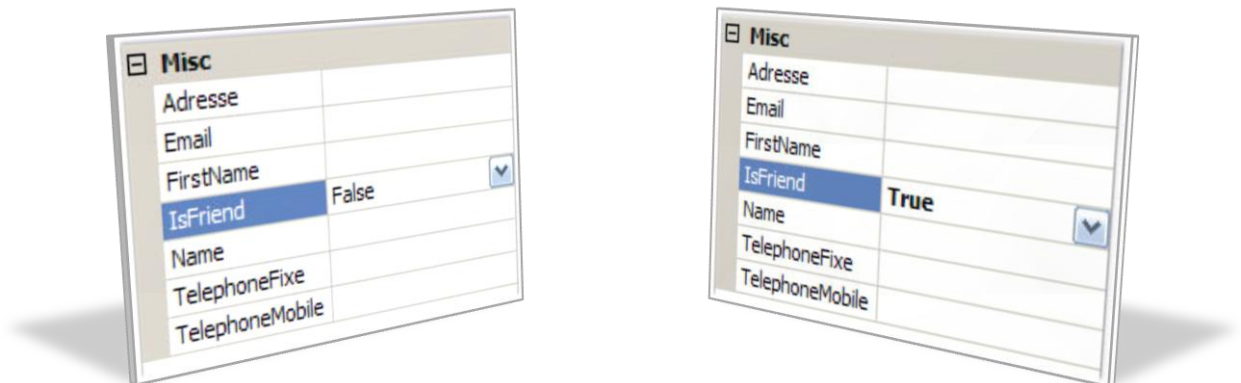
```
[DefaultValue(false)]
public bool IsFriend
{
    get { return isFriend; }
    set { isFriend = value; }
}
```

```

[DefaultValue(null)]
[Browsable(false)]
public byte[] Photo
{
    get { return photo; }
    set { photo = value; }
}

```

... et retournez dans la propriété List du contrôle **FriendsArea** via l'inspecteur de propriété. Ajoutez un contact et regardez bien :



Vous constaterez deux choses : la propriété Photo a disparu et la valeur par défaut de la propriété est affichée sans gras alors que si l'on modifie cette valeur et que celle-ci n'est pas celle établit par défaut, elle s'affiche en gras.

Outre l'aspect esthétique, définir des valeurs par défaut permet de réduire le code car les valeurs par défaut ne sont pas redéfinies dans le code généré automatiquement avec l'éditeur...

Ajoutez également des attributs **DefaultValue** aux autres propriétés de la classe **Contact**...

Voilà pour les attributs, nous avons fait le tour des principaux...

Finissez-en avec le contrôle **FriendsArea** en traitant son évènement par défaut dans le formulaire principal...

```

private void friendsArea1_DetailsButtonClick(object sender, EventArgs e)
{
    contactArea1.Contact = friendsArea1.SelectedContact;
}

```

... et n'oubliez pas qu'il faut que la liste de contact du formulaire principal soit en permanence dans le contrôle **FriendsArea**...

```

//Dans le constructeur de MainForm
friendsArea1.DynList = myContacts;

//Dans la classe FriendsArea
[Browsable(false)]
public List<Contact> DynList
{
    set { myList = value; }
}

```

Testez votre application !

Fin

Occupez vous maintenant du dernier contrôle : [PhotoArea](#). Rien de difficile ici, au clic sur un bouton, vous ouvrez une [OpenFileDialog](#), récupérez l'image, la stockez dans la propriété Photo du [Contact](#) sous forme binaire et vice-versa.

Le code du bouton Modifier n'est donc guère compliqué, il suffit de savoir un petit peu utiliser une [OpenFileDialog](#) - que nous traiterons ici en code, même s'il est possible de la créer avec le Designer (essayez ensuite avec le Designer) - et comment transformer une image [Image](#) en un tableau de [bytes](#) (il faut passer par la classe [MemoryStream](#) de l'espace System.IO). Le code donne alors quelque chose du genre :

```
/* Il faut que le contact existe, sinon la suite n'a pas besoin
 * d'être exécutée. */
if (currentContact != null)
{
    //Création de l'OpenFileDialog
    OpenFileDialog ofd = new OpenFileDialog();
    //On récupère le chemin des "Mes Documents"
    string myDocs = Environment.GetFolderPath(
        Environment.SpecialFolder.MyDocuments);

    //On configure l'OpenFileDialog
    ofd.Title = "Choisissez une image";
    ofd.Multiselect = false;
    ofd.InitialDirectory = myDocs;
    ofd.Filter = "Fichiers JPEG(*.jpg,*.jpeg) | *.jpeg;*.jpg";

    /* On affiche l'OpenFileDialog et on attend que l'utilisateur
     * appuie sur OK. */
    if (ofd.ShowDialog() == DialogResult.OK)
    {
        /* On vérifie que l'utilisateur à bien choisi un fichier
         * existant. */
        if (System.IO.File.Exists(ofd.FileName))
        {
            /* On récupère le chemin de ce fichier et on demande
             * à la PictureBox d'en afficher l'image. */
            pictureBox.Image = Image.FromFile(ofd.FileName);

            /* On crée une zone tampon en mémoire pour contenir les
             * données binaires du fichier image... */
            System.IO.MemoryStream ms = new System.IO.MemoryStream();
            /* ... que l'on récupère via la PictureBox... */
            pictureBox.Image.Save(ms,
                System.Drawing.Imaging.ImageFormat.Jpeg);
            /* ... et qu'on stocke finalement sous forme binaire dans
             * la propriété Photo du contact actuel. */
            currentContact.Photo = ms.ToArray();
        }
        else
        {
            /* Si le fichier n'existe pas, on informe l'utilisateur
             * du problème. */
            MessageBox.Show("Le fichier spécifié n'existe pas",
                "Erreur",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
        }
    }
}
```

Remarque : N'oubliez pas de créer le champ `currentContact` et sa propriété associée. La propriété devant gérer des images... cadeau !

```
[Browsable(false)]
public Contact Contact
{
    get { return currentContact; }
    set
    {
        currentContact = value;

        /* Ne pas oublier de traiter l'affichage automatique
         * des images. */
        if (currentContact != null && currentContact.Photo != null)
        {
            //On récupère les données binaire de la classe Contact
            System.IO.MemoryStream ms = new System.IO.MemoryStream(
                currentContact.Photo);
            /* La classe Bitmap a un constructeur prenant une Stream
             * en paramètre, profitons en ! */
            pictureBox.Image = new Bitmap(ms);
        }
        else
        {
            /* Si le contact n'a pas d'image, on en met une vide
             * à la place. */
            pictureBox.Image = new Bitmap(10, 10);
        }
    }
}
```

Remarque : L'image pouvant être absolument n'importe quoi et de toutes dimensions, réglez la propriété `SizeMode` de la `PictureBox` sur `Zoom`. Cela redimensionnera l'image à afficher de manière à ce que ses dimensions restent proportionnelles et inférieures à celles de la `PictureBox`.

Ajoutez ensuite ce contrôle au formulaire principal... Tout va bien.

Maintenant, il faut ajouter le code dans le formulaire principal pour passer le `Contact` en question à `PhotoArea` lorsqu'un élément de la `ListBox` est sélectionné...

Nous avons déjà la propriété `SelectedContact` dans la classe `FriendsArea` mais nous ne savons pas dans le formulaire principal quand un élément est sélectionné dans la `ListBox`... Il nous manque en effet un évènement dans le contrôle `FriendsArea` :

```
//Il faut ajouter cet évènement personnalisé...
[Category("Action")]
[Description("Occurs when a Contact is selected in the ListBox.")]
public event EventHandler SelectedContactInBoxChanged;

private void listBoxFriends_SelectedIndexChanged(object sender,
EventArgs e)
{
    // [...]
    //... et cet "appel"...
    if (SelectedContactInBoxChanged != null)
        SelectedContactInBoxChanged(sender, e);
}
```

On peut à présent traiter cet événement depuis le formulaire principal.

```
private void friendsArea1_SelectedContactInBoxChanged(object sender, EventArgs e)
{
    photoArea1.Contact = friendsArea1.SelectedContact;
}
```

Aurions-nous enfin terminé ?

Presque, il reste à activer et désactiver les différents **GroupBox** aux bons moments... Tout cela se gère bien sûr dans le formulaire principal.

Initialement, la liste étant vide, il faut que les **GroupBox** Mes amis et Photo soient inactives. Réglez donc la propriété Enabled des contrôles **FriendsArea** et **PhotoArea** sur False depuis l'inspecteur de propriété.

Le reste se passe dans le code...

```
private void addContact(object sender, EventArgs e)
{
    myContacts.Add(contactArea1.Contact);
    //Un contact a été ajouté, on peut activer FriendsArea
    friendsArea1.Enabled = true;
    //Sans oublier de nettoyer tous les champs du contrôle
    contactArea1.Clear();
}

private void friendsArea1_DetailsButtonClick(object sender,
    EventArgs e)
{
    //Des détails sont demandés, on désactive ContactArea
    contactArea1.Enabled = false;
    contactArea1.Contact = friendsArea1.SelectedContact;
}

private void friendsArea1_SelectedContactInBoxChanged(object sender,
    EventArgs e)
{
    /* Un élément a été sélectionné dans la ListBox,
     * il faut activer la zone de photo. */
    photoArea1.Enabled = true;
    photoArea1.Contact = friendsArea1.SelectedContact;
}
```

Mais il manque encore la réactivation de la zone Contact lorsque la **ListBox** perd le focus. Il va donc falloir ajouter un événement dans le contrôle **FriendsArea** qui signale au formulaire que la **ListBox** a perdu le focus...

Et bien allons-y :

```
//L'évènement personnalisé...
[Category("Focus")]
[Description("Occurs when the ListBox is no longer the active control of the form.")]
public event EventHandler LeaveBox;

//L'évènement Leave de la ListBox
private void listBoxFriends_Leave(object sender, EventArgs e)
{
    if (LeaveBox != null)
        LeaveBox(sender, e);
}
```

Et ensuite, dans le formulaire principal :

```
private void friendsArea1_LeaveBox(object sender, EventArgs e)
{
    //... on vide la zone contact et on la réactive ...
    contactArea1.Clear();
    contactArea1.Enabled = true;
}
```

Ne pas oublier également que les détails du contact peuvent être affichés pour un contact appartenant à la [ComboBox](#). Nous appellerons donc l'évènement `LeaveBox` dans la méthode traitant l'évènement `SelectedIndexChanged` de la [ComboBox](#) ainsi que sur le clic du bouton Ajouter...

Après cela, l'application est terminée, vous pouvez la tester totalement.

Remarque : *Cependant, certaines erreurs n'ont pas été anticipées. Essayez donc de les corriger.*

Pour aller plus loin...

Vous souvenez vous de cette remarque ?

Remarque : [...]

- *Vu que l'on ajoute que les noms et prénoms du contact dans la [ComboBox](#), on n'a pas directement accès à celui-ci et il va falloir effectuer une nouvelle recherche pour le retrouver et modifier sa propriété `IsFriend`. En fait, on pourrait se passer de cette recherche en sachant que les items de la [ComboBox](#) ne sont non pas de type [string](#) mais de type [object](#) ! On pourrait donc passer l'instance de [Contact](#) toute entière lors de l'ajout de l'item mais il y aurait des surcharges de méthodes à faire (juste une en fait) pour que l'affichage du contact soit bon. Nous verrons ce cas tout à la fin de ce chapitre.*
- *Le même problème sera rencontré par la suite pour récupérer un contact de la [ListBox](#) : il faudra encore effectuer une recherche.*

Nous voici à la fin du chapitre et nous allons voir, rapidement, comment nous aurions pu faire autrement et nous passer ainsi de cette méthode de recherche...

[ListBox](#) et [ComboBox](#) prennent dans leur liste d'items des objets. Toutes les classes étant des objets, nous pouvons directement passer notre [Contact](#) à la liste d'items de la [ComboBox](#) et de la [ListBox](#).

```
private void buttonSearch_Click(object sender, EventArgs e)
{
    comboBoxResults.Items.Clear();
    /* [...] */

    //Dans la condition de la boucle...
    /* Version "Pour aller plus loin" */
    comboBoxResults.Items.Add(c);
    //On ajoute ici TOUT le contact à la ComboBox

    /* [...] */
}
```

De même au moment de l'ajout dans la [ListBox](#).

```
private void buttonAdd_Click(object sender, EventArgs e)
{
    /* Version "Pour aller plus loin" */
    Contact cct = (Contact)comboBoxResults.SelectedItem;
    cct.IsFriend = true;
    listBoxFriends.Items.Add(cct);
    //On ajoute ici TOUT le contact à la ListBox
    /* [...] */
}
```

La méthode `getContact` devient alors inutile et la propriété `SelectedContact` est simplifiée :

```
public Contact SelectedContact
{
    get
    {
        /* Version "Pour aller plus loin" */
        if (listBoxFriends.SelectedItems.Count == 1)
            return (Contact)listBoxFriends.SelectedItem;
        //else
        return (Contact)comboBoxResults.SelectedItem;
    }
}
```

Mais il y a un nouveau problème. En effet, si vous exécutez votre application avec ce nouveau code, vous ne verrez pas, ni dans la `ListBox` ni dans la `ComboBox`, d'écrit le nom du contact suivi de son prénom mais un texte incompréhensible...

C'est normal car vous n'avez fait que passer une classe à vos `ListBox` et `ComboBox`. Ces contrôles utilisent, pour afficher des objets sous forme de texte, la méthode `ToString()` : ils appellent pour chaque objet la méthode `ToString()` et affichent la chaîne renvoyée par la méthode.

Etant donné que notre classe `Contact` n'a pas sa propre méthode `ToString()`, c'est la méthode par défaut qui est appelée par nos `ListBox` et `ComboBox` et cette méthode renvoie l'affichage que vous avez pu voir.

Pour résoudre ce problème, il suffit simplement de surcharger la méthode `ToString()` : de la réécrire dans notre classe `Contact` pour qu'elle renvoie un affichage nous convenant.

Dans la classe `Contact`, vous devrez donc ajouter...

```
public override string ToString()
{
    return Name + " " + FirstName;
}
```

A présent tout fonctionne convenablement.

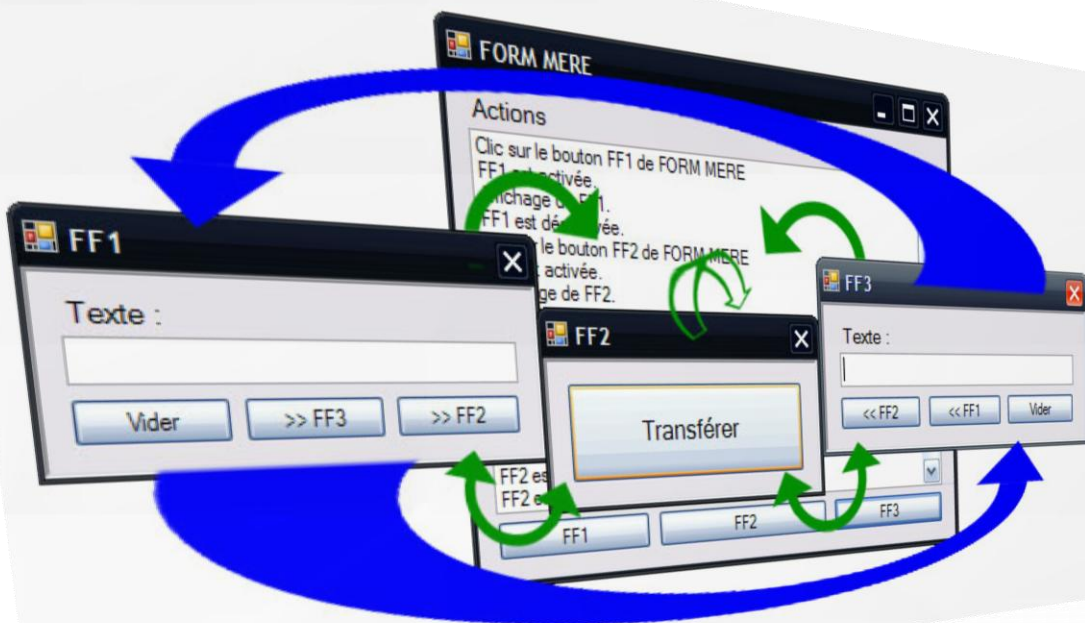
Cette dernière technique est largement préférable à la première (qui consistait à réeffectuer une recherche de contact à chaque fois) et sera privilégiée dans ce type d'application. Cela ne veut pas dire qu'il faut passer à chaque fois un objet à vos `ComboBox`, `ListBox` ou autre, mais simplement que si cela s'y prête bien (simplifie le code, augmente la vitesse d'exécution, etc.).

Chapitre 5

Multifenêtres, suite...

Maintenant que nous avons vu, massivement, comment créer ses propres évènements, que vous commencez à devenir un virtuose de la propriété à un tel point que vous en mettez de partout et que les interfaces un peu complexes à gérer ne vous font plus peur, nous allons pouvoir revenir sur les applications multifenêtres et voir comment gérer les communications entre formulaires enfants.

Application



Créez et remplissez les quatre formulaires. Modifiez-en les noms... Nous nommerons ici :

- dans FORM MERE :
 - la **ListBox** : "listBox"
 - le bouton FF1 : "buttonFF1"
 - le bouton FF2 : "buttonFF2"
 - le bouton FF3 : "buttonFF3"
- dans FF1 :
 - la **TextBox** : "textBoxFF1"
 - le bouton Vider : "buttonVider1"
 - le bouton >> FF3 : "buttonFF13"
 - le bouton >> FF2 : "buttonFF12"
- dans FF2 :
 - le bouton Transférer : "buttonTransfert"
- dans FF3 :
 - la **TextBox** : "textBoxFF3"
 - le bouton Vider : "buttonVider1"
 - le bouton >> FF1 : "buttonFF31"
 - le bouton >> FF2 : "buttonFF32"

Les trois formulaires enfants ont les propriétés suivantes en commun :

- MinimizeBox sur False
- MaximizeBox sur False
- StartPosition sur CenterParent
- FormBorderStyle sur Fixed3D
- TopMost sur True

A chaque fois qu'un bouton est cliqué, l'évènement est signalé dans la [ListBox](#). Quand du texte est modifié, quand le formulaire est affiché, fermé, activé, désactivé, l'évènement est signalé dans la [ListBox](#).

Il va donc y avoir beaucoup d'évènement à traiter ici.





Lorsque le bouton >> FF2 est cliqué, le texte de la [TextBox](#) du formulaire est transmis au formulaire FF2. On attend alors que l'utilisateur clique sur l'unique bouton de FF2 pour que le transfert se termine et que le texte passe dans le formulaire opposé au formulaire d'origine...

...

Commencez par créer tous les évènements Click de tous les boutons et occupez vous du code nécessaire à l'ouverture des formulaires enfants depuis FORM MERE (un clic sur FF1 ouvre le formulaire FF1, un clic sur ...).

Remarque : Utilisez ici la méthode Show() pour affichez les formulaires enfants. En effet, si vous utilisez ShowDialog(), vous ne pourrez plus accéder au formulaire principal pour ouvrir les autres formulaires enfants et ne pourrez donc jamais avoir plus d'un formulaire enfant affiché simultanément.

Nous allons nous occuper tout de suite des évènements directement accessibles depuis le formulaire parent. Ce sont les évènements standards des formulaires enfants :

-  Activated
-  Deactivate
-  Shown
-  FormClosed

Pour alléger le code, nous utiliserons les méthodes anonymes de C# 2.0.

```
ff1.Activated += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("FF1 est activée.");
};
```

Remarque : Si vous êtes encore sous C# 1.1, c'est pareil, sauf que vous créez une méthode pour chaque nouvel évènement et placez le code ajoutant du texte à la [ListBox](#) en son sein. Utiliser les méthodes anonymes va nous permettre ici de nous passer de plus d'une vingtaine méthodes (nommées).

Ci-dessous, le code traitant l'un des formulaires enfant, à placer dans le constructeur du formulaire principal :

```
public MainForm()
{
    InitializeComponent();

    //Les évènements associés à ff1
    ff1.Activated += delegate(object sender, EventArgs e)
    {
        listBox.Items.Add("FF1 est activée.");
    };
};
```

```

ff1.Deactivate += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("FF1 est désactivée.");
};
ff1.Shown += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("Affichage de FF1.");
};
ff1.FormClosed += delegate(object sender, FormClosedEventArgs e)
{
    listBox.Items.Add("Fermeture de FF1.");
};

//...
}

```

Remarque : Pensez à créer les instances des formulaires enfants...

Le code traitant les autres formulaires est du même genre...

Nous allons ensuite avoir besoin d'informations sur les contrôles que contiennent nos formulaires enfants. Pour cela, vous devez savoir à présent qu'il est nécessaire de créer nos événements personnalisés...

Dans le formulaire FF1, nous aurons donc besoin d'un événement personnalisé par contrôle (puisque nous n'allons traiter qu'un seul des événements de ces contrôles).

```

public event EventHandler TextInBoxChanged;
public event EventHandler ButtonViderClick;
public event EventHandler ButtonFF2Click;
public event EventHandler ButtonFF3Click;

```

Vous ne devriez pas avoir de mal ensuite à appeler ces événements aux bons moments...

Nous allons ensuite avoir besoin de récupérer le texte qui se trouve dans la [TextBox](#). A cela, vous devez répondre : « Propriété » ! Il faut écrire une propriété qui renvoie ou modifie ce texte.

```

[DefaultValue("")]
public string TextInBox
{
    get { return textBoxFF1.Text; }
    set { textBoxFF1.Text = value; }
}

```

Le formulaire enfant FF3 se traite de la même manière...

Pour le formulaire FF2, c'est encore plus simple vu qu'il n'y a qu'un seul bouton :

```

public event EventHandler ButtonTransfertClick;

/* [...] */

private void buttonTransfert_Click(object sender, EventArgs e)
{
    if (ButtonTransfertClick != null)
        ButtonTransfertClick(sender, e);
}

```

On peut alors ajouter dans le formulaire principal, encore une fois en utilisant les méthodes anonymes, le code affichant dans la **ListBox** quand le bouton Vider est cliqué, quand le texte d'une des **TextBox** change...

Par exemple, pour le changement de texte...

```
ff1.TextInBoxChanged += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("Le texte dans FF1 a été modifié : " +
ff1.TextInBox);
};
```

Faites de même pour tous les autres événements restant à traiter...

Nous allons voir à présent comment organiser les transferts de données.

Deux possibilités se présentent :

- la **TextBox** n'est pas vidée lorsqu'on clique sur l'un des boutons de transfert
- la **TextBox** est vidée depuis le formulaire enfant lorsqu'un des boutons de transfert est cliqué

Nous allons étudier ici ces deux cas.

Premier cas

Le premier cas se traite assez rapidement...

Pour les transferts s'effectuant directement de FF1 à FF3, il suffit de modifier certains des événements déjà utilisés :

```
ff1.ButtonFF3Click += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("Le bouton >> FF3 a été cliqué.");
    ff3.TextInBox = ff1.TextInBox;
};
```

De même pour l'opération inverse :

```
ff3.ButtonFF1Click += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("Le bouton << FF1 a été cliqué.");
    ff1.TextInBox = ff3.TextInBox;
};
```

L'étape intermédiaire (le passage par le formulaire FF2) demande un peu plus de réflexion car il faut tenir compte du sens et de « l'unicité » du transfert (celui-ci ne peut avoir lieu qu'une fois et cliquer plusieurs fois sur le bouton Transférer ne devra rien faire une fois la première passée).

Mais cela reste assez simple tout de même.

Nous avons besoin de connaître sur le formulaire FF2 la provenance du transfert : est-ce le bouton << FF2 ou le bouton >> FF2 qui a été cliqué ?

Pour cela, on peut par exemple stocker l'information dans une énumération.

Commençons par la créer... ajoutez la dans votre programme au même rang qu'une classe...

```
public enum ComingFrom
{
    None,
    FF1,
    FF3
}
```

Définissons-la ensuite dans notre formulaire principal (à ajouter parmi les champs) :

```
ComingFrom from = ComingFrom.None;
```

Il ne reste plus qu'à s'en servir aux bons endroits...

```
ff1.ButtonFF2Click += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("Le bouton >> FF2 a été cliqué.");
    from = ComingFrom.FF1;
};

//... plus loin...
ff3.ButtonFF2Click += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("Le bouton << FF2 a été cliqué.");
    from = ComingFrom.FF3;
};

//... plus loin...
ff2.ButtonTransfertClick += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("Le bouton Transfert a été cliqué.");
    if (from == ComingFrom.FF1)
        ff3.TextInBox = ff1.TextInBox;
    else if (from == ComingFrom.FF3)
        ff1.TextInBox = ff3.TextInBox;
    //Le transfert est unique...
    from = ComingFrom.None;
};
```

Voilà, c'est tout ce qu'il y a à faire dans ce cas là...

Passons au second cas qui va nous demander un peu plus d'efforts.

Second cas

Dans ce cas ci, nous devons nettoyer les `TextBox` dès qu'un des boutons de transfert est cliqué...

```
private void buttonFF13_Click(object sender, EventArgs e)
{
    if (ButtonFF3Click != null)
        ButtonFF3Click(sender, e);
    textBoxFF1.Text = "";
}
```

Faites de même pour les autres évènements...

La propriété `TextInBox` va donc rarement servir pour extraire le texte de la `TextBox`... et il va falloir trouver un autre moyen de conserver le texte.

Et le seul moyen qui semble se présenter, c'est d'ajouter de faire passer la chaîne de caractères dans l'évènement... après, c'est trop tard !

Il va donc falloir créer de nouveaux types d'évènements, qui n'existent pas encore. Fini les [EventHandler](#) traditionnels, à nous de créer notre propre Handler.

Pour cela, on a encore deux choix :

- soit on crée un Handler comme le [EventHandler](#) mais avec un troisième paramètre de type [string](#).
- soit on crée un Handler d'une forme un peu plus standard qui ne prend que deux paramètres.

Nous allons coder ici la deuxième possibilité, qui, d'une certaine manière, englobe la première...

Revoyons un peu les paramètres que demandait le [EventHandler](#). Il fallait :

- un premier paramètre de type [object](#)
- un second de type [EventArgs](#)

A quoi correspondaient donc ces paramètres ?

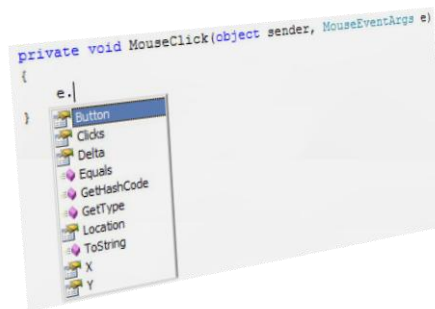
Le premier, l'[object](#), que nous nommions `sender`, contenait en fait l'objet émetteur de l'évènement. Par exemple, l'évènement Click d'un bouton contient dans son `sender`, le bouton lui-même mais sous forme d'un objet.

Remarque : Pour récupérer le contrôle contenu dans le `sender`, il suffit de faire un transtypage.







Le second paramètre, de type [EventArgs](#), est celui qui va nous intéresser davantage.

Bien que [EventArgs](#) en lui-même n'ait pas beaucoup d'intérêt (car il ne contient aucune méthode, propriété de plus que la classe [Object](#)), les autres classes qui s'y apparentent commencent à être intéressantes.

Par exemple, la classe [MouseEventArgs](#), utilisée pour les évènements du style `MouseClick`, `MouseDown` ou autre, fournit des informations intéressantes sur l'état de la souris lorsque l'évènement est survenu.



De nouvelles propriétés apparaissent dans cette classe :

-  `Button`
-  `Clicks`
-  `Delta`
-  `Location`
-  `X`
-  `Y`

Nous pourrions donc en faire autant avec notre propre [EventArgs](#).

Commençons donc par écrire notre propre [EventArgs](#) : il s'agit d'écrire une classe, dont le nom se terminerait, par convention, par `EventArgs`, et héritant de la classe de base [EventArgs](#).

Cette classe doit être capable de stocker notre chaîne de caractères et devra donc comporter un champ de type [string](#) et une propriété renvoyant sa valeur.

La classe ressemblera donc à quelque chose du genre :

```
/* On crée notre classe et on la fait hériter de
 * EventArgs (avec les deux points). */
public class StringTransfertEventArgs : EventArgs
{
    //Notre champs pour stocker la chaîne
    private string chaine;
```

```

//Un constructeur par défaut
public StringTransfertEventArgs()
    : this(String.Empty) /* On appelle ici le
                          * constructeur ci-
                          * dessous. */
{
}

/* Un autre qui permet de donner une chaîne
 * à notre classe StringTransfertEventArgs. */
public StringTransfertEventArgs(string chaine)
    : base() /* On appelle ici le constructeur
            * de la classe dont on hérite :
            * on appelle donc le constructeur
            * de la classe EventArgs. */
{
    this.chaine = chaine;
}

//La propriété qui renvoie la chaîne
public string Chaine
{
    get { return chaine; }
    /* Pas d'accessor set, nous ne voulons
     * pas que quelqu'un d'autre à part nous
     * puisse modifier la valeur de la chaîne :
     * cela ne voudrait plus rien dire si celle-ci
     * peut être modifiée n'importe quand... */
}
}

```

Remarques :

- Pensez à créer un fichier spécialement pour cette classe (et nommez le comme la classe).
- L'implémentation de la classe est ici intégralement traitée, avec la présence des deux constructeurs et de la propriété accessible uniquement en lecture seule.
- `String.Empty` est équivalent à `""`

Maintenant, nous aimerions bien que nos événements personnalisés sur nos boutons >> FF2 et >> FF3 ressemble à ceci...

```

public event StringTransfertEventHandler ButtonFF2Click;
public event StringTransfertEventHandler ButtonFF3Click;

```

... où le nouvel Handler en question prendrait en paramètre un type `object` et un type `StringTransfertEventArgs`.

Mais ce Handler n'existe pas encore... Si vous avez bonne mémoire et que vous lisez attentivement, vous vous rappellerez peut-être qu'il était écrit quelque part que `EventHandler` était en fait un délégué : un « peintre de méthode ».

Nous devons donc créer un délégué qui, comme `EventHandler`, ne renverra rien mais devra prendre deux arguments en paramètre (l'`object` et le `StringTransfertEventArgs`).

En voici la déclaration, que vous placerez dans votre code au même rang qu'une classe...

```

public delegate void StringTransfertEventHandler(
    object sender,
    StringTransfertEventArgs e
);

```

Modifiez ensuite les deux événements ButtonFF2Click et ButtonFF3Click comme indiqué ci-dessus.

Ce qui entraîne qu'il va falloir modifier nos « appels » sur ces événements et créer un objet `StringTransfertEventArgs`...

```
private void buttonFF13_Click(object sender, EventArgs e)
{
    if (ButtonFF3Click != null)
        ButtonFF3Click(
            sender,
            /* On crée un objet StringTransfertEventArgs et on lui
            /* passe (via son constructeur) le texte de la TextBox...*/
            new StringTransfertEventArgs(textBoxFF1.Text)
        );

    textBoxFF1.Text = "";
}
```

Modifiez de même les autres événements similaires dans les formulaires FF1 et FF3... Vous allez également devoir effectuer quelques petites modifications dans le formulaire principal. Si vous ne voyez pas lesquelles, recompilez et consultez la liste d'erreurs...

La suite est très semblable au premier cas...

```
//champs
ComingFrom from = ComingFrom.None;
string temp = String.Empty;

//... un peu plus loin...
ff1.ButtonFF2Click += delegate(object sender, StringTransfertEventArgs e)
{
    listBox.Items.Add("Le bouton >> FF2 a été cliqué.");
    temp = e.Chaine; //Stockage temporaire de la chaîne (sinon on la perd)
    from = ComingFrom.FF1;
};
ff1.ButtonFF3Click += delegate(object sender, StringTransfertEventArgs e)
{
    listBox.Items.Add("Le bouton >> FF3 a été cliqué.");
    ff3.TextInBox = e.Chaine; //On récupère désormais la chaîne via e

    /* Premier cas */
    //ff3.TextInBox = ff1.TextInBox;
};
```

... et surtout...

```
ff2.ButtonTransfertClick += delegate(object sender, EventArgs e)
{
    listBox.Items.Add("Le bouton Transfert a été cliqué.");

    if (from == ComingFrom.FF1)
        ff3.TextInBox = temp;
    else if (from == ComingFrom.FF3)
        ff1.TextInBox = temp;
    /* La suite peut varier selon le comportement que l'on veut
    * avoir pour le bouton Transférer... */
    temp = String.Empty; //Facultatif puisqu'on a à la suivante...
    //Le transfert est unique...
    from = ComingFrom.None;
};
```

Remarque : A propos de l'énumération *ComingFrom*.

Celle-ci ne servant que dans la classe du formulaire principal, vous pouvez, si vous voulez que celle-ci soit rendue inaccessible aux autres classes de votre programme, l'ajouter à la classe *MainForm* plutôt qu'à l'espace de nom de votre application...

```
public partial class MainForm : Form
{
    FF1 ff1 = new FF1();
    FF2 ff2 = new FF2();
    FF3 ff3 = new FF3();
    ComingFrom from = ComingFrom.None;
    string temp = String.Empty;

    protected enum ComingFrom
    {
        None,
        FF1,
        FF3
    }
}
```

Voilà, c'est terminé...

Testez l'application et vérifiez de ne rien avoir oublié (en la testant dans tous les sens possibles et imaginables)...

Chapitre 6

A vous de jouer

Dans ce chapitre, fini les corrections et l'aide permanente, à vous de vous débrouiller tout seul... enfin presque.

Help me !

Pas de panique, vous n'êtes fort heureusement pas seul dans votre tâche.

Vous avez, entre autres, pour vous aider :

- [Microsoft Developer Network](#) qui liste l'intégralité des classes de C# dans sa [library](#) en fournissant des exemples dans la plupart des cas. MSDN est à la fois disponible en local (installation avec Visual C#) et sur internet.
- Les nombreux forums de programmation que vous pourrez trouver sur internet et notamment le site de [Moteurprog](#).

Cependant, si vous êtes un peu dégourdi et patient (c'est là généralement que c'est le plus difficile), MSDN vous fournit toutes les informations nécessaires pour que votre application puisse aboutir.

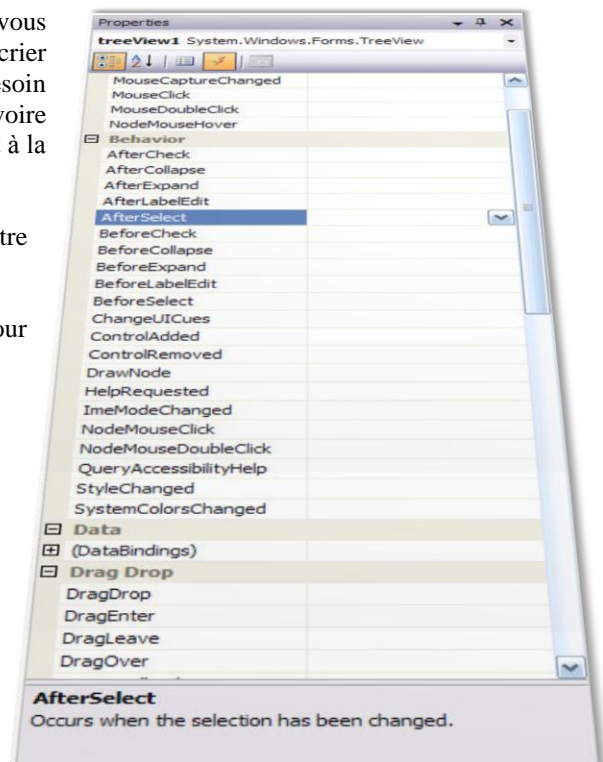
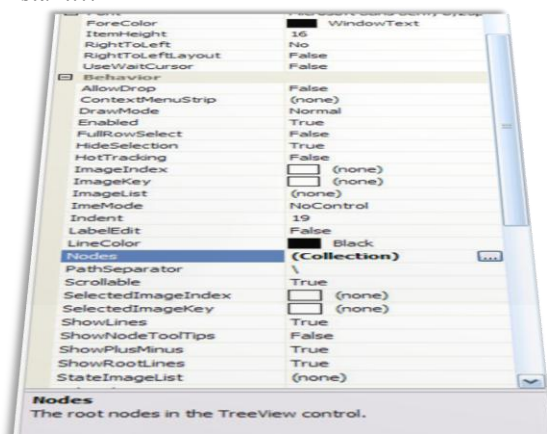
A noter cependant que l'aide MSDN ne fournit pas d'exemples directs d'utilisation pratique de Visual Studio pour tel ou tel contrôle (vous avez le texte mais pas les images). Tout est donné en code. Après, c'est à vous de faire le lien entre les éléments du code et l'inspecteur de propriétés.

Exemple : La TreeView

Vous ne vous êtes jamais servi d'une [TreeView](#) et vous voulez savoir comment faire. Plutôt que de crier immédiatement sur tous les toits que vous avez besoin d'aide, commencez par mener votre petite enquête, voire si vous ne pourriez pas vous en sortir tout seul. C'est à la fois gratifiant et instructif.

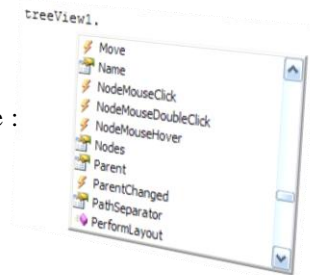
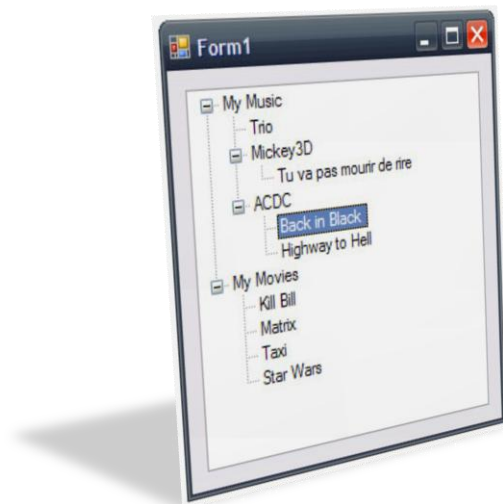
Pour cela, commencez par ajouter la [TreeView](#) à votre application via le Designer.

Et voyez si vous avez vraiment besoin de MSDN pour l'instant...



Essayez différentes propriétés qui se trouvent dans l'inspecteur de propriétés pour voir ce qu'elles font...

Vous devriez arriver assez facilement ainsi à créer un arbre dans ce style :



Mais cela ne vous suffit pas, vous voulez que votre arbre réagisse aux entrées utilisateurs, que votre arbre évolue pendant que votre application s'exécute... Il va donc falloir obligatoirement passer par le code... et c'est là que MSDN va surement devenir très utile.

En effet, même si vous avez à disposition toutes les méthodes de la [TreeView](#), vous aurez certainement du mal, si vous n'êtes pas habitué à utiliser le contrôle, de passer par MSDN pour obtenir quelques informations quant à la façon de s'en servir.

Effectuez donc une recherche sur la [TreeView](#) dans MSDN (local ou en ligne)...

Vous devriez trouver entre autres :

- [TreeView, classe](#)
- [TreeView, membres](#)
- [TreeNode, classe](#)

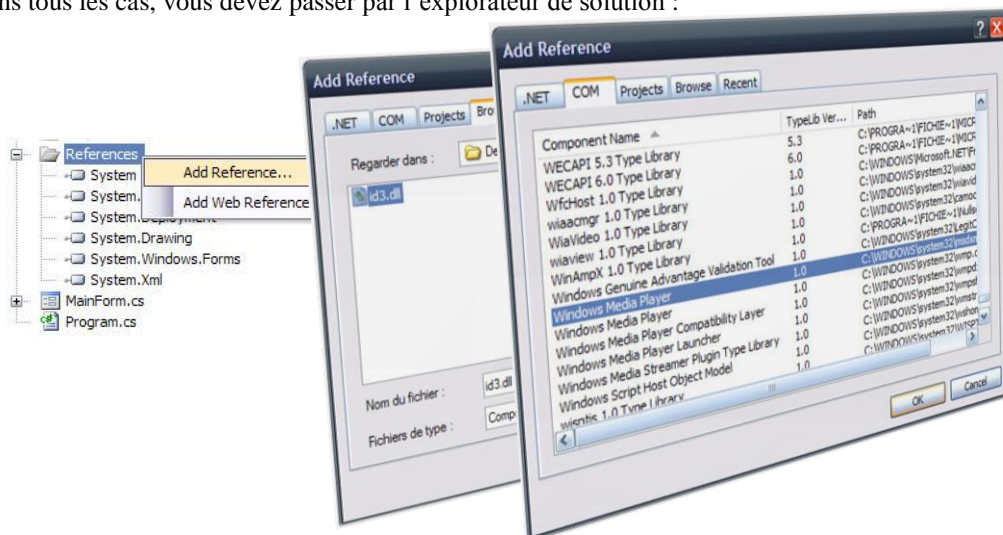
Dernières informations

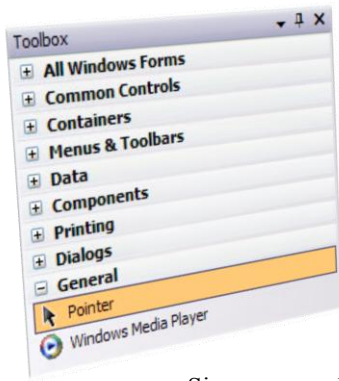
Il se peut que vous ayez besoin pour votre application d'ajouter des composants externes, non inclus dans le .NET Framework à la base.

C'est tout à fait possible bien sûr mais il y a deux cas à distinguer :

- Le composant a été fait dans un langage non .NET mais utilise la technologie COM pour être exploitable sous .NET. Dans ce cas, il suffit d'ajouter la référence de l'application depuis l'onglet COM...
- Le composant a été fait pour .NET et se trouve dans une DLL, un fichier OCX ou autre... Dans ce cas, pas de problème, ajoutez votre fichier DLL au projet (pensez également à ajouter la DLL dans le répertoire où l'exécutable est compilé).
-

Dans tous les cas, vous devez passer par l'explorateur de solution :





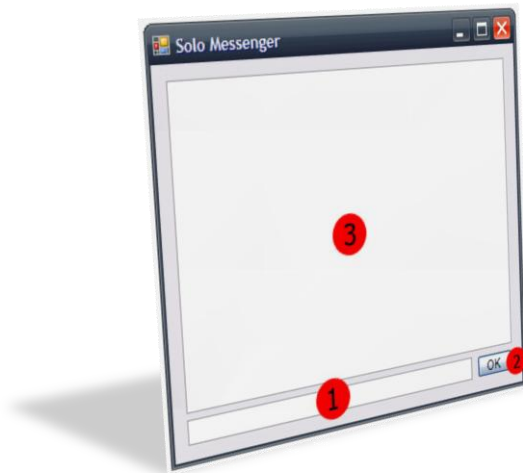
Si vous voulez ajouter plus particulièrement des contrôles (externes), qu'ils soient .NET ou non... Vous pouvez ajouter des contrôles à votre Toolbox en y faisant un *clic droit* → *Choose Items*. Des fenêtres semblables aux précédentes s'affichent alors et vous pouvez choisir le contrôle que vous voulez ajouter à la Toolbox...

Si vous avez besoin d'un composant non fourni par défaut avec le .NET Framework, effectuez une recherche sur internet. Certains sites proposent des lots de composants (généralement payant), d'autres le composant qui vous intéresse seul (gratuit ou payant).

A vous de jouer

Ci-après, une série d'exercices, classés par difficulté croissante, qui vous fera progresser petit à petit. Plus vous avancerez dans les exercices, plus vous aurez de code à manipuler.

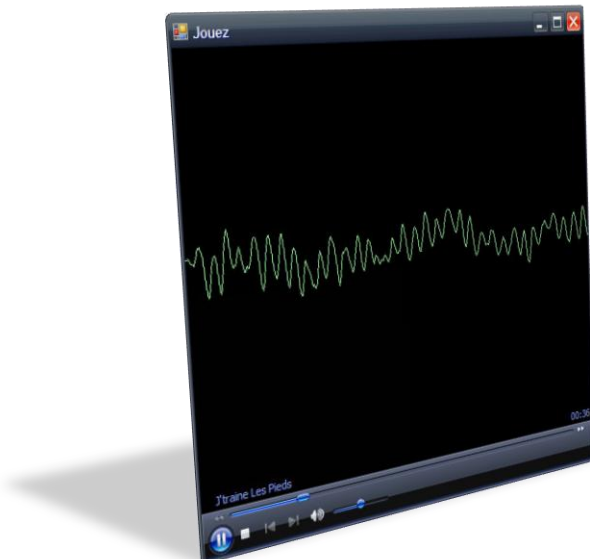
Exercice 1 : Solo Messenger



Comportement : Vous tapez du texte dans le contrôle 1 et en cliquant sur le contrôle 2, le texte est ajouté au contrôle 3 avec en plus marqué devant : "J'ai dit : ". Le texte du contrôle 1 est alors automatiquement effacé.

Conseil : Rien de plus, rien de moins. Libre à vous de rendre l'interface la plus agréable possible (notamment dans les moments de redimensionnement de la fenêtre).

Exercice 2 : Jouez

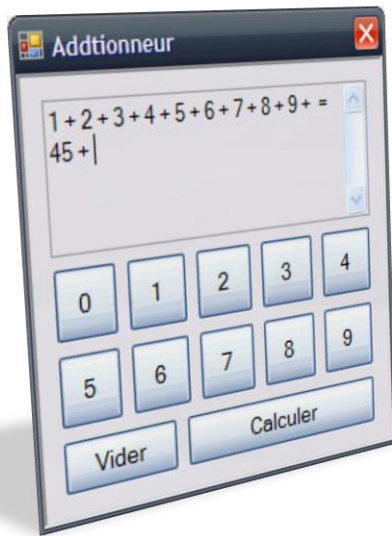


Comportement : En lançant l'application, une de vos musiques est automatiquement jouée par le lecteur Windows Media Player.

Conseil : Intéressez vous aux propriétés du composant en question.

Remarque : L'apparence du lecteur ne dépend que de sa version. Ici, c'est la version 11 qui a été utilisée.

Exercice 3 : Additionneur

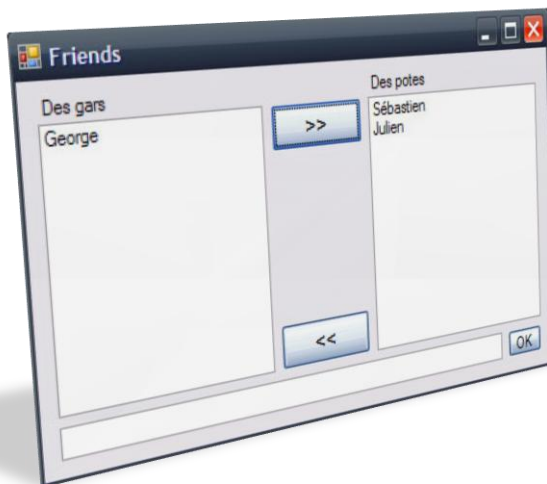


Comportement : L'utilisateur ne peut additionner que des chiffres. A chaque clic sur un chiffre, le chiffre suivi de " + " est ajouté à la [TextBox](#) (multiligne et avec scrollbars). En cliquant sur le bouton Calculer, le résultat final de l'addition est affiché (suivi de " + " lui aussi). En cliquant sur le bouton Vider, on vide la [TextBox](#) et remet à zéro le compteur. Le contenu de la [TextBox](#) ne peut pas être modifié autrement que par les boutons...

Le formulaire est verrouillé : impossible d'en changer les dimensions.

Conseil : Effectuez les additions au fur et à mesure que l'utilisateur clique sur un chiffre, c'est beaucoup plus simple et tout à fait adapté à ce cas très particulier.

Exercice 4 : Friends



Comportement : Vous ajoutez le nom de la personne tapée dans le contrôle 1 dans le contrôle 3 en cliquant sur le 2. Après sélection (unique !) dans le contrôle 3, vous pouvez faire passer (elle n'existera plus alors dans le contrôle d'origine) la sélection dans le contrôle 5 en cliquant sur le 4 et vice-versa en cliquant sur le 6.

Conseil : Pensez à vérifier que la personne n'a pas déjà été ajoutée dans les contrôles dans l'une des deux listes avant de l'ajouter dans la liste de gauche.

Exercice 5 : Notepad

Ouvrez Notepad (ou Bloc-notes si vous préférez) et refaites exactement le même logiciel en C# en ne codant cependant pas les fonctions :

- Mise en page
- Imprimer
- Rechercher
- Rechercher le suivant
- Remplacer
- Atteindre
- Police
- Barre d'état
- Rubrique d'aide
- Le choix d'encodage

Attention : N'oubliez surtout pas les raccourcis claviers, ni les boîtes de sauvegarde, d'ouverture de documents, etc. et aussi que lorsque vous ouvrez/enregistrez un fichier, son nom s'affiche en titre de form...

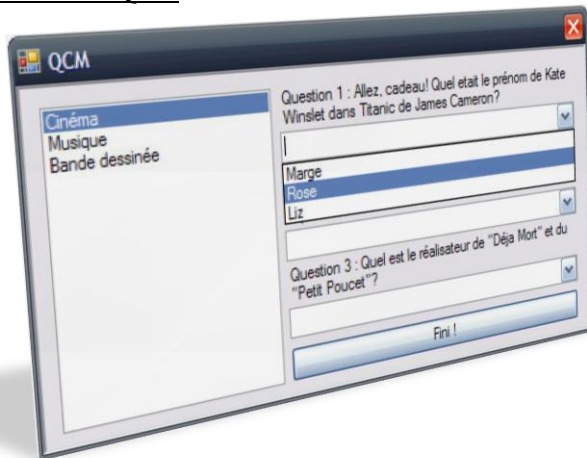
Conseil : Intéressez vous au contrôle [RichTextBox](#), à la classe [DateTime](#) et à la propriété [Shortcuts](#).

Exercice 6 : MyWeb



Comportement : L'utilisateur entre l'adresse internet du site qu'il veut visiter dans la **TextBox**, valide avec le bouton OK ou en appuyant sur la touche Entrer. Le chargement s'affiche alors dans la barre d'état avec marqué à côté "**Chargement en cours**". Une fois le chargement terminé, la barre d'état disparaît et la page s'affiche automatiquement... L'utilisateur peut alors basculer à tout moment entre la page Page, où il voit la page web comme dans n'importe quel navigateur internet, et la page Source, où tout le code source de la page chargée est affiché. L'utilisateur ne doit pas pouvoir modifier le code source !

Exercice 7 : QCM



Comportement : La **ListBox** propose trois QCM différents. En en choisissant un, l'utilisateur change instantanément les questions et réponses se trouvant à droite. Les réponses sont uniques. Quand l'utilisateur a fini, il clique sur le bouton et une boîte de dialogue apparaît pour lui indiquer le nombre d'erreurs qu'il a commis ou s'il a fait tout juste.

Le premier QCM de la liste est sélectionné par défaut au lancement de l'application.

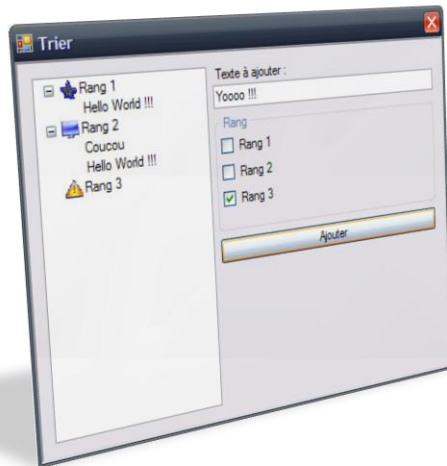
Vous trouverez divers QCM sur ce site : <http://quizz.e-qcm.net/>. Limitez vous à cinq questions par QCM pour ne pas rendre l'exercice trop long... mais vous pouvez faire plus bien entendu.

***Conseil :** N'utilisez pas ici de contrôles personnalisés : l'interface est très simple et ne change jamais. Seuls les textes changent ! Allez donc voir du côté du code.*

Exercice 7 bis

Même exercice que précédemment mais remplacez les **ComboBox** par des **RadioButton** (et effectuez les modifications qui s'imposent).

Exercice 8 : Faites le tri



Comportement : L'utilisateur entre du texte, choisit le « rang » de ce texte (il a droit à plusieurs choix) et en validant, le texte est ajouté à l'arbre aux rangs spécifiés.

Par défaut, au lancement de l'application, l'arbre contient trois nœuds de bases : rang 1, rang 2, rang 3.

Exercice 9 : PictureViewer



Comportement : En ouvrant un fichier de type image (et uniquement de type image, pas d'autre choix), une nouvelle fenêtre incluse dans la principale est ajoutée. Cette nouvelle fenêtre ne contient que l'image, affichée à 100% (et la fenêtre fait la taille de l'image). L'utilisateur peut alors zoomer sur chacune des images sélectionnées. Le zoom se fait par étape : 10, 20, 33, 50, 66, 75, 80, 90, 100, 150, 200, 300, 400, 500%.

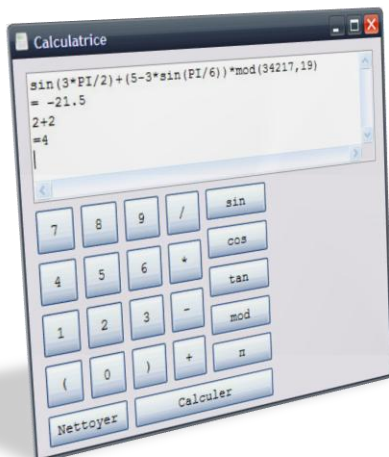
Attention : En passant d'une fenêtre à l'autre, l'ancien zoom est automatiquement rétabli. Le zoom dépend de l'image (et donc de la fenêtre) et n'est pas globalisé à toutes les images.

Conseil : Utilisez une [PictureBox](#).

Exercice 10 : Notepad (suite)

Reprenez l'exercice 5 mais codez à présent toutes les fonctionnalités de Notepad et ajoutez-y même la fonction rétablir avec son raccourci : Ctrl+Y.

Exercice 11 : Calculatrice



Calculatrice effectuant toutes les opérations de base sur des nombres (plusieurs chiffres) positifs ou négatifs, fonctions de trigonométrie, etc. mod correspond à l'opérateur modulo et prend deux paramètres : mod(5,2) signifie 5 modulo 2 (en mathématiques, on sait que $5 \equiv 1[2]$).

Comportement : L'utilisateur entre son opération en appuyant sur les boutons ou sur les touches de son clavier (ne pas oublier sin, cos, ... pi).

L'opération s'affiche à gauche dans le contrôle **TextBox** au fur et à mesure que l'utilisateur l'entre. En appuyant sur le bouton Calculer, l'opération est effectuée et le résultat, précédé du signe égal est affiché sur la ligne d'après. Le prochain calcul commence sur la ligne suivante.

Le bouton Nettoyer vide l'écran.

Il est impossible de revenir sur une opération précédente : le calcul ne s'effectue que sur la dernière opération entrée et non encore calculée !

Belle calculatrice !

Remarques :

- Ici la difficulté n'est pas de créer l'interface mais plutôt d'établir l'algorithme traitant les éléments fournis par l'interface et de relier correctement ces deux ensembles.
- Les angles seront exprimés en radians !
- Pensez à traiter un maximum d'erreurs (par exemple : les nombres entrés dans la fonction mod ne sont pas des entiers, etc.)

Exercice 12 : Explorateur



Comportement : Le logiciel se lance par défaut sur le Poste de Travail.

Lorsque l'utilisateur clique sur un fichier dans la **TreeView** ou dans la **ListView**, les propriétés du fichier sont affichées dans la zone verte.

S'il fait un simple clic sur un dossier dans la **TreeView**, le contenu est affiché dans la **ListView**. S'il clique sur un dossier dans la **ListView**, son nom, sa taille (totale), le nombre de fichiers, de dossiers qu'il contient est affiché dans zone verte. S'il double clic sur un dossier dans la **TreeView** ou dans la **ListView** et que celui-ci n'est pas vide, il est développé dans la **TreeView** et dans la **ListView**.

La sélection multiple est interdite.

Conseils : Utilisez des contrôles personnalisés pour chacun de vos éléments séparés par un **Splitter** et n'hésitez pas à créer de nouveaux fichiers, de nouvelles classes...

Notez que le temps de calcul de taille d'un dossier peut prendre un certain temps (car il faut parcourir chaque sous dossier, fichier, etc.). Vous devrez donc utiliser les **Thread** pour ne pas paralyser votre application et rafraîchir les données affichées (taille, nombre de sous dossiers, fichiers...) régulièrement pour montrer que votre application travaille...