

1. Utilisation des formulaires.

1.1. Création d'un formulaire.

Dans les applications Windows, le formulaire est l'élément primaire avec lequel l'utilisateur va agir. Lorsque l'on désire créer sous Visual Studio .NET une application Windows sous forme d'un projet, on retrouvera par défaut un formulaire contenant la forme minimal comprenant une barre de titre, une boîte de contrôle et les boutons Réduire, agrandir et fermer.

Pour ce formulaire, nous pouvons accéder à un ensemble de propriétés dont voici les plus communes:

Name	C'est le nom du formulaire. Ce ne sera pas le nom devant apparaître lors de l'exécution de l'application mais le nom de l'objet associé au formulaire et permettant d'accéder aux propriétés pour les modifier lors de l'exécution du programme.
AcceptButton	Permet de définir le bouton sur lequel on clique lorsque l'utilisateur appuie sur la touche ENTER
CancelButton	Permet de définir le bouton sur lequel on clique lorsque l'utilisateur appuie sur la touche ESC
ControlBox	Permet de définir si un formulaire doit comprendre une boîte de contrôle incluant les boutons réduire, agrandir ou fermer. Pourra prendre la valeur False ou True.
FormBorderStyle	Contrôle l'apparence du bord du formulaire. Cela peut aussi avoir un effet sur la barre de titre et sur les boutons qui s'y trouvent
MaximizeBox	Permet de rendre le bouton d'agrandissement dans la barre de titre accessible. Peut prendre comme valeur False ou True.
MinimizeBox	Permet de rendre le bouton de réduction dans la barre de titre accessible. Peut prendre comme valeur False ou True.
StartPosition	Détermine la position que le formulaire doit occuper sur l'écran lorsqu'il apparaît la première fois.
Text	Détermine le titre devant apparaître dans la barre de titre.

Ces propriétés peuvent être définies lors de la création de l'application ou en temps réel lors de son exécution. Dans les deux cas, l'accès à ces propriétés s'effectue par l'objet associé au formulaire. Nous retrouvons dans le code, le constructeur Form1:

```
public Form1()  
{  
    InitializeComponent();  
}
```

La méthode `InitializeComponent` comprend les différentes initialisations correspondant aux différents choix effectués dans la boîte de dialogue propriétés du formulaire. En voici le contenu:

```
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);  
this.ClientSize = new System.Drawing.Size(292, 266);  
this.HelpButton = true;
```

```
this.Name = "Form1";
this.Text = "Form1";
```

Comme nous l'avons vu dans le langage C#, l'opérateur `this` renvoie une référence sur l'objet ayant provoqué l'appel à la méthode dans laquelle on se trouve.

Nous allons retrouver pour notre formulaire un ensemble d'événements et de méthodes.

New	L'événement Initialize est typiquement utilisé pour préparer une application à l'emploi en assignant des valeurs initiales aux variables, en déplaçant les contrôles ou en les redimensionnant. Dans l'environnement .NET, le code d'initialisation peut être ajouté dans le constructeur ou dans la méthode InitializeComponent.
Show	Cette méthode inclut un appel implicite à la méthode load si le formulaire n'a pas encore été chargé, cette dernière méthode provoquant son chargement en mémoire. La méthode Show peut afficher un formulaire en modal (avec la méthode Show) ou en modeless (avec la méthode ShowDialog).
Load	Cet événement se produit lorsque le formulaire est chargé en mémoire.
Activated/ Desactivated	L'événement se produit lorsque la fenêtre associée à un formulaire reçoit le focus. Recevoir le focus signifie que c'est la fenêtre qui reçoit au niveau des entrées/sorties les informations provenant du clavier et de la souris. Ce focus peut se recevoir par programmation à partir d'un autre formulaire ou par l'utilisateur via le clavier ou la souris. Par le code, nous pouvons utiliser la méthode Focus.
Closing	Cet événement se produit lorsque le formulaire reçoit une requête de fermeture. Si il y a un besoin de maintenir le formulaire ouvert pour des questions d'enregistrement de données, l'événement de fermeture peut être annulé.
Closed	Cet événement se produit juste après que le formulaire soit fermé mais avant l'événement Dispose.
Dispose	Le code de nettoyage des ressources doit être placé dans la méthode Dispose avant l'appel à la même méthode appartenant à la classe de base.
Hide	Méthode permettant de cacher un formulaire sans l'enlever de la mémoire. Un formulaire caché ne peut pas être accessible par l'utilisateur ni recevoir le focus mais l'application reste active.

Nous allons créer un nouveau projet et dans le formulaire associé par défaut à ce projet, placer un bouton que nous utiliserons pour tester certaines des méthodes. La méthode `MessageBox.Show` permettra d'afficher une boîte à message affichant le texte passé en paramètre.

Exemple 1: utilisation de l'opérateur `new`. Nous devons associer à l'événement `click` du bouton le fait d'afficher un nouveau formulaire.

1. Création d'un nouveau formulaire. Pour créer ce formulaire, nous accéderons au menu `File / Add New Item / Windows Form`
2. Dans le formulaire de départ appelé par défaut `Form1`, nous placerons un bouton en utilisant la boîte à outils.

3. En double cliquant sur le bouton dans le formulaire, sera créé automatiquement la méthode associée à l'événement Click. En voici le code:

Dans la méthode InitializeComponents, nous retrouvons la ligne de code suivante:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Dans les propriétés du bouton, nous retrouvons la propriété Name qui correspond à l'objet nous permettant dans le code de pouvoir accéder aux autres propriétés. Par défaut, lors de la création du bouton, le nom `button1` a été assigné à la propriété Name. De ce fait, comme le bouton appartient au formulaire, nous pourrions accéder à l'objet `button1` à partir d'une des méthodes de la classe du formulaire en utilisant la syntaxe `this.button1`. Comme paramètre de la méthode `EventHandler`, nous indiquerons le nom de la méthode associée à l'événement. Cette méthode doit comprendre des paramètres déterminés que l'on retrouve dans la syntaxe de définition:

```
1     private void button1_Click(object sender, System.EventArgs e)
2     {
3         Form2 test = new Form2();
4         test.ShowDialog(this);
5     }
```

Commentaires:

A la ligne 3, nous retrouvons la création d'un nouvel objet de type `Form2`. Une fois la référence initialisée, nous pouvons appeler la méthode `ShowDialog` pour un formulaire de type `modless` pour lequel nous devons indiquer la fenêtre propriétaire de laquelle ce formulaire dépendra. Nous pouvons remplacer cette méthode par `Show()` pour un formulaire de type `modal`. Le fait de cliquer sur le bouton provoque l'affichage du deuxième formulaire.

Nous placerons également deux boutons sur le deuxième formulaire. Le fait de cliquer sur le premier bouton doit cacher le premier formulaire et le fait de cliquer sur le deuxième bouton doit le faire réapparaître. Voici le contenu du code:

1	<code>private void button1_Click(object sender, System.EventArgs e)</code>
2	<code>{</code>
3	<code> this.Owner.Hide();</code>
4	<code>}</code>
5	
6	<code>private void button2_Click(object sender, System.EventArgs e)</code>
7	<code>{</code>
8	<code> this.Owner.Show();</code>
9	<code>}</code>

Commentaires:

La ligne 3 comprend un appel à la méthode `Hide` permettant de cacher le premier formulaire correspondant en fait au propriétaire du deuxième formulaire d'où l'utilisation de la propriété `Owner`.

La ligne 8 comprend un appel à la méthode `Show` permettant de réafficher le premier formulaire si celui-ci était caché.

Exemple 2:

Dans ce deuxième exemple, nous allons envisager la façon dont nous pouvons initialiser le contenu d'une zone d'édition que nous placerons dans notre formulaire. La première méthode est de placer le code d'initialisation dans la méthode `InitializeComponent` et nous retrouverons alors la syntaxe suivante:

```
this.Edit1 = new System.Windows.Forms.TextBox();
this.Edit1.Location = new System.Drawing.Point(88, 80);
this.Edit1.Name = "Edit1";
this.Edit1.Size = new System.Drawing.Size(120, 20);
this.Edit1.TabIndex = 1;
this.Edit1.Text = "textBox1";
```

Nous retrouvons le code inséré automatiquement par le Visual Studio .NET dont la dernière ligne provoquant l'affichage du texte "textBox1" lors de l'exécution du programme. Pour rappel, l'appel de la méthode `InitializeComponent` se retrouve dans le constructeur du formulaire.

Nous pouvons également modifier le contenu de la zone d'édition en plaçant cette dernière ligne dans la méthode correspondant à l'événement `Load`.

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.Edit1.Text="Changement";
}
```

Exemple 3: Reprenons l'exemple précédent et modifions le contenu de la zone d'édition en fonction du fait que le formulaire ait le focus ou pas.

Nous allons mettre en place la gestion des événements `Activated` et `Deactivated`. Nous retrouverons alors les codes suivant dans la méthode `InitializeComopnent`:

```
this.Activated += new System.EventHandler(this.Active);
this.Deactivate += new System.EventHandler(this.Desactive);
```

Nous avons créé deux méthodes `Active` et `Desactive` dont les contenus seront:

```
1     private void Activate(object sender, System.EventArgs e)
2     {
3         this.Edit1.Text="Focus";
4     }
5
6     private void Desactivate(object sender, System.EventArgs e)
7     {
8         this.Edit1.Text="Focus perdu";
9     }
```

Nous retrouvons aux lignes 3 et 8 l'accès à la propriété `Text` permettant de modifier le contenu de la zone d'édition correspondant à l'objet `Edit1`.

Exercice 1: Veuillez mettre en évidence grâce à la méthode `MessageBox.Show` la génération de l'événement `Closing` lors de la demande de fermeture de votre formulaire.

1.2. Création d'un menu.

Les menus apportent un moyen structuré aux utilisateurs d'accéder à de commandes et outils contenus dans votre application. Une planification et un design corrects des menus et barres d'outils sont essentiels pour assurer une fonctionnalité et une accessibilité de vos applications aux utilisateurs.

Nous pouvons ajouter un menu à notre application en utilisant la boîte à outils. Il suffira de double cliquer sur le contrôle MainMenu et l'on retrouvera juste sous la barre de titre de notre formulaire, une barre de menu. Ce menu étant vu dans notre code comme un objet, nous devons lui donner un nom en modifiant la propriété Name.

Exemple 1: Nous allons considérer un menu comprenant une option à cocher. Si cette option est cochée, une zone d'édition présente dans le formulaire est accessible tandis que si elle n'est pas cochée, la zone est en lecture seule. Par défaut, lors du démarrage de l'application, la zone d'édition doit être en lecture seule.

Pour rendre la zone d'édition accessible en lecture seule, nous en modifierons la propriété ReadOnly, ce qui provoquera l'insertion automatique de la ligne suivante dans le code:

```
this.Edit1.ReadOnly = true;
```

Pour choisir qu'une option d'un menu doit pouvoir être cochée, il suffit d'éditer les propriétés de cet objet et de modifier la propriété Checked en la plaçant à true. Nous retrouverons alors dans notre code l'insertion automatique suivante:

```
this.Visible.Checked = true;
```

Un double clic sur cette option nous permet d'inclure la gestion de l'événement Click sur cet objet. Nous retrouvons alors l'insertion du code suivant dans notre source:

```
this.Visible.Click += new System.EventHandler(this.ReadOnly_Click);
```

La méthode ReadOnly_Click comprendra le code suivant:

```
1     private void ReadOnly_Click(object sender, System.EventArgs e)
2     {
3         this.Visible.Checked=!this.Visible.Checked;
4         this.Edit1.ReadOnly=this.Visible.Checked;
5     }
```

Commentaires:

La ligne 3 permet de changer l'état de l'option. Si elle était cochée, elle ne doit plus l'être et inversement. La ligne 4 nous permet de modifier la propriété ReadOnly de la zone d'édition pour qu'elle suive l'état de l'option du menu:

Option cochée – zone d'édition en lecture seule.

Option non cochée – zone d'édition en lecture seule.

Dans le cadre des menus, il est parfois intéressant de travailler avec des menus contextuels qui apparaissent avec le click droit de la souris.

Exemple 2: Soit un formulaire comprenant une liste, un menu contextuel avec deux options: ajouter et retirer ainsi qu'une boîte de dialogue.

Lorsque un item dans la liste est sélectionné et que l'on choisit l'option retirer du menu contextuel, l'item doit être retiré. Si l'on choisit l'option ajouter de ce même menu, une boîte de dialogue modale doit apparaître pour demander le texte devant apparaître sous l'item ajouté dans la liste.

Pour ajouter un menu contextuel dans le formulaire, il faut prendre dans la boîte d'outil le contrôle `ContextMenu` et le glisser sur le formulaire. Une fois le menu contextuel ajouté, nous pouvons définir des items et leur donner un nom d'objet et le texte associé à cette option dans le menu. Nous choisissons le même nom pour les propriétés `Name` et `Text`: `Ajouter` et `Retirer`.

Une fois ces propriétés définies, nous pouvons définir la méthode qui sera liée à chaque option une fois celle-ci sélectionnée dans le menu contextuel. L'événement que l'on doit gérer est `Click`.

Une fois ces événements paramétrés, nous retrouverons les codes suivants dans notre source:

```
this.Ajouter.Text = "Ajouter";
this.Ajouter.Click += new System.EventHandler(this.ContexteAjouter);
this.Retirer.Text = "Retirer";
this.Retirer.Click += new System.EventHandler(this.ContexteRetirer);
```

Voici le contenu des deux méthodes `ContexteAjouter` et `ContexteRetirer`:

```
1 private void ContexteAjouter(object sender, System.EventArgs e)
2 {
3     Form3 diag = new Form3();
4     if (diag.ShowDialog(this)==DialogResult.OK)
5     {
6         this.listBox.Items.Add(diag.EditBox.Text);
7     }
8 }
9
10 private void ContexteRetirer(object sender, System.EventArgs e)
11 {
12     int i = this.listBox.SelectedIndex;
13     if (i==-1) return;
14     this.listBox.Items.RemoveAt(i);
15 }
```

Commentaires:

A la ligne 3, nous créons une instance d'un formulaire que nous utilisons comme boîte de dialogue. Cette boîte de dialogue sera de type modale du fait de l'utilisation de la méthode `ShowDialog(this)`.

Nous avons ajouté deux boutons dans le formulaire: le bouton `Enter` et le bouton `Cancel`. `Enter` et `Cancel` sont des noms que nous avons choisis pour les propriétés `Name` et `Text`. Nous avons paramétré la propriété `DialogResult` des boutons pour que la valeur `Ok` soit retournée si l'on clique sur le bouton `Enter` et que ce soit la valeur `Cancel` si c'est le bouton `Cancel` sur lequel nous cliquons.

Nous pouvons même définir les propriétés `AcceptButton` et `CancelButton` du formulaire `Form3` pour que l'appui sur la touche `ENTER` soit associé à un clic sur le bouton `Enter` et que l'appui sur la touche `ESC` soit associé à un clic sur le bouton `Cancel`. Nous retrouvons le code suivant inséré automatiquement dans notre source:

```
this.AcceptButton = this.Enter;  
this.CancelButton = this.Cancel;
```

A la ligne 4, nous testons la valeur de retour de la boîte de dialogue. Si elle correspond à Ok, c'est que l'utilisateur a cliqué sur le bouton Enter pour en sortir et nous pouvons dès lors récupérer le contenu de la zone d'édition pour l'ajouter comme nouvel item dans la liste. Nous avons choisi EditBox pour la propriété *Name* de la zone d'édition.

Si nous désirons que le menu contextuel apparaisse quel que soit la position du curseur de la souris dans le formulaire, nous devons en modifier la propriété ContextMenu pour l'associer à l'objet de notre menu contextuel.

Si nous souhaitons que le menu contextuel n'apparaisse que pour la liste, nous devons modifier la propriété ContextMenu mais uniquement de la liste. Nous retrouvons le code suivant inséré automatiquement:

```
this.listBox.ContextMenu = this.contextMenu1;
```

1.3. Organisation des contrôles dans un formulaire.

Lorsque nous plaçons des contrôles dans un formulaire, nous devons être attentifs aux aspects suivants:

- Faire en sorte de choisir l'ordre de parcourt des contrôles lorsque nous appuyons sur la touche de tabulation, le travail d'encodage s'en trouvant facilité si nous pouvons éviter l'emploi de la souris.
- Lorsque l'utilisateur décide de redimensionner sa fenêtre, il est intéressant de pouvoir figer la position d'un contrôle par rapport à un bord du parent conteneur.

1. Mise en forme lors de la conception à partir du menu Format

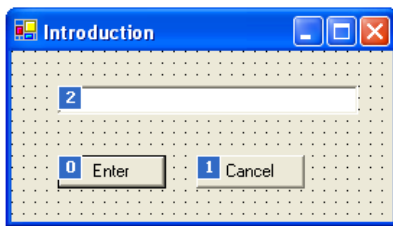
La difficulté revient parfois à garantir un alignement de deux contrôles, de choisir une même dimension... Si nous sélectionnons deux contrôles dans un formulaires en choisissant la touche CTRL conjointement au clic de souris, nous pouvons ensuite retrouvez des choix de positionnement dans le menu Format que l'on retrouve dans le tableau suivant:

Align	Permet d'aligner tous les contrôles en respect avec le contrôle primaire.
Make Same Size	Redimensionne les contrôles dans un formulaire pour leur donner la même taille.
Horizontal spacing	Augmente, diminue, supprime ou rend égal les espacements horizontaux entre plusieurs contrôles
Vertical Spacing	Augmente, diminue, supprime ou rend égal les espacements verticaux entre plusieurs contrôles
Center in Form	Permet de centrer verticalement ou horizontalement les contpoles sélectionnés
Lock Controls	Verrouille les contrôles dans un formulaire
Order	Lorsque deux contrôles se superposent, cette option permettra de définir le contrôle devant se situer au dessus ou au dessous.

Vous pouvez retrouver la correspondance de ces options dans une des barres d'outils que l'on peut ajouter à l'affichage standard.

2. Comment définir l'ordre des tabulations.

La première chose à effectuer est de forcer l'affichage des numéros indiquant l'ordre actuel de vos contrôles directement sur le formulaire lors de sa conception: vous devez sélectionner l'option View/Tab Order dans la barre de menu.



Pour modifier la valeur affectée à un contrôle, vous pouvez pratiquer de deux façons:

- 1- Dans les propriétés de chaque contrôle, vous pouvez changer la valeur affectée à la propriété TabIndex et éventuellement choisir si on doit s'y arrêter lors des différentes tabulations par la propriété TabStop.
 - 2- En cliquant sur le numéro affiché près du contrôle, ce qui permettra de faire défiler les différentes valeurs autorisées.
3. Comment ancrer un contrôle dans un formulaire.

Si lors de l'exécution de votre programme vous ajoutez dynamiquement des contrôles à un formulaire pouvant être redimensionné par l'utilisateur, ceux-ci devront être redimensionnés et repositionnés correctement. Lorsque un contrôle est ancré à un formulaire et que le formulaire est redimensionné, le contrôle maintient ses distances par rapport au formulaire conteneur. Vous pouvez déterminer les bords sur lesquels cet ancrage s'effectue en modifiant la propriété *Anchor* des différents contrôles.

4. Comment fixer un contrôle dans un formulaire (dock).

Vous pouvez fixer un contrôle à un bord de votre formulaire; pensons notamment au navigateur Windows dans lequel l'arborescence de vos disques apparaît exclusivement fixé sur le bord gauche de la fenêtre. Pour figer un de vos contrôles, il suffira de modifier la propriété *Dock* de ce contrôle.

1.4. Utilisation de l'héritage visuel.

L'héritage virtuel offre beaucoup d'avantages aux programmeurs. Si des formulaires ont déjà été développés dans des projets différents et que ces formulaires sont semblables à ceux que vous désirez recréer, vous pouvez hériter de ceux-ci. On peut donc créer des formulaires de base comme modèles pour un usage ultérieur sans devoir les recréer à partir de rien. Si vous effectuez des modifications aux formulaires de base, ces modifications se reporteront sur l'ensemble des héritages sans devoir envisager un travail fastidieux de modification dans l'ensemble de vos projets.

Prenons par exemple un projet dans lequel nous disposons d'un formulaire du type à propos de. Pour mettre en évidence la possibilité d'une réutilisation possible de ce code, nous allons créer une solution dans laquelle nous ajouterons deux projets: le premier comprenant le formulaire de base et un deuxième héritant de ce formulaire.

Pour créer un formulaire héritant d'un formulaire existant en utilisant l'interface graphique nous devons suivre les étapes suivantes:

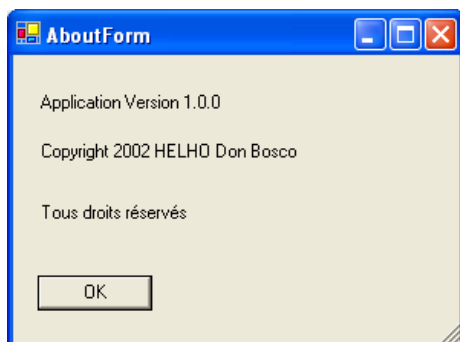
- A partir du menu *Project*, choisir *Add Inherited Form*. La boîte de dialogue *Add New Item* apparaît.
- Sélectionner *Local Project Items* dans le panneau *Categories* sur la partie gauche et sélectionner *Inherited Form* dans le panneau *Templates* de droite. Cliquer alors sur le bouton *Open*. La boîte de dialogue *Inheritance Picker* apparaît.
- Si vous désirez choisir un formulaire présent dans un des projets de la même solution, ceux-ci apparaîtront dans la liste des composants. Si le formulaire est présent ailleurs, il suffit de cliquer sur le bouton *browse* permettant de naviguer dans le projet contenant le formulaire, de cliquer sur le fichier .dll et de cliquer sur le bouton *Open*. Cette opération vous permettra de voir les composants maintenant listés et le projet dans lequel ils figurent
- Sélectionner le composant.

Si le composant correspond à une interface utilisateur, les différents contrôles qui font partie du formulaire hérité seront marqués d'un symbole. La sélection du contrôle fera apparaître un bord dont les caractéristiques dépendront du niveau de sécurité défini dans le formulaire de base.

Modificateur d'accès	Description
Private	Lecture seule. Tous les aspects du contrôle seront considérés comme étant en lecture seule. Nous ne pouvons pas déplacer ou redimensionner le contrôle ou changer ses propriétés. Si le contrôle contient d'autres contrôles comme un <i>group box</i> , de nouveaux contrôles ne pourront pas être ajoutés et les contrôles existants ne pourront pas être enlevés.
Public	C'est le niveau le moins restrictif. Les contrôles peuvent être redimensionnés et déplacés et nous pouvons changer la valeur des propriétés. Les contrôles peuvent être accédés de façon interne par la classe qui les déclare et de façon externe par une autre classe.
Protected	Les contrôles peuvent être déplacés et redimensionnés. Ils peuvent être accédés de façon interne dans la classe qui les déclare mais par des classes externes.

Exemple:

Dans une première étape, nous allons créer un projet dans lequel nous définirons un formulaire de type à propos de dont voici une copie d'écran.

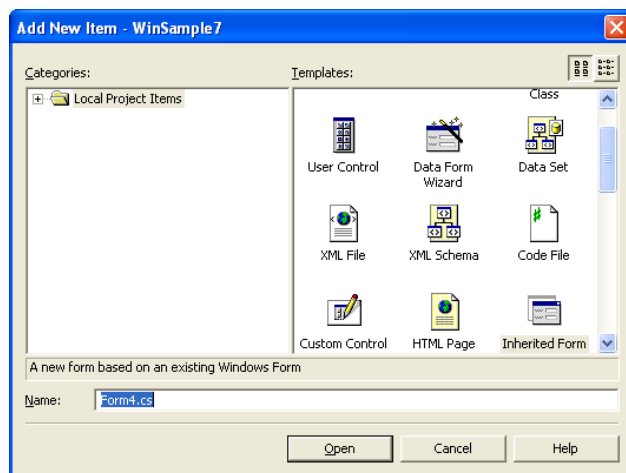


Nous retrouvons trois contrôles de type *label* dont la propriété *modifiers* sera choisie comme *private*. Nous pourrons créer un menu dont l'option Aide/A propos de permettra de faire apparaître le formulaire ainsi créé. Par défaut, ce formulaire sera associé au nom *Form2* et le nom du projet sera choisi comme étant *Winbase*.

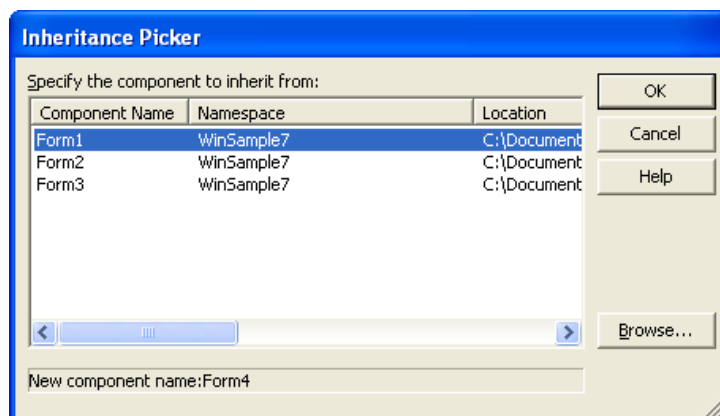
Lorsque notre projet est fonctionnel, nous visualiserons les propriétés du projet accessibles par le navigateur de solution, clic sur le bouton droit de la souris, *properties*.

Nous modifierons dans les propriétés communes le type de sortie que nous devons obtenir lors de la compilation. Par défaut, comme nous avons choisi une application Windows, nous devons retrouver *Windows Application*. Nous changerons cette valeur par *Class Library* de sorte d'obtenir un fichier ayant comme extension dll.

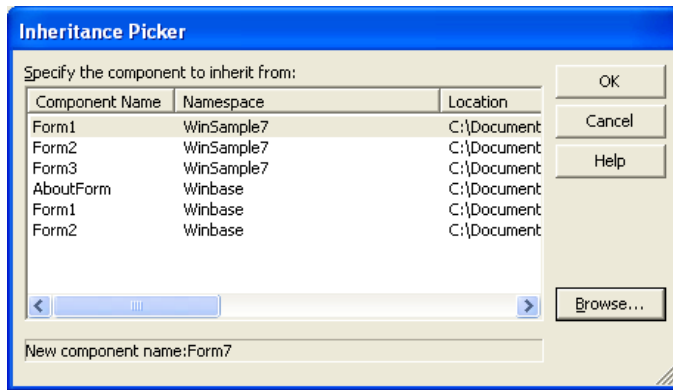
Dans une deuxième étape, nous allons recréer un deuxième projet. Dans le menu Project, nous pourrons choisir l'option *Add Inherited Form* permettant de faire apparaître la boîte de dialogue *Add New Item*.



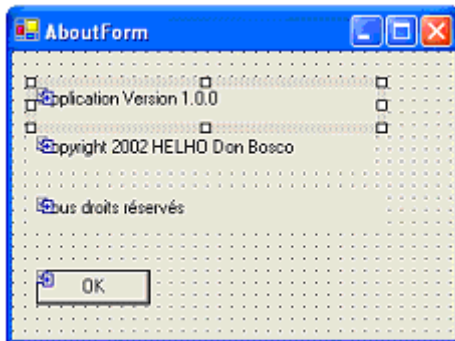
Nous pourrons choisir dans le panneau de gauche *Local Project Items* et dans le panneau de droite *Inherited Form*. Un clic sur le bouton *Open* nous permettra d'obtenir la boîte de dialogue *Inheritance Picker*.



Nous retrouverons l'ensemble des composants présent dans notre projet courant. Nous pourrons alors cliquer sur le bouton *Browse* nous permettant de parcourir nos différents projets et choisir alors le fichier dll précédemment créé. Nous obtiendrons alors la liste suivante:



Nous retrouvons alors l'espace de nom dans lequel sont créés les différents formulaires associés à notre projet *Winbase*. Nous choisirons le composant correspondant au formulaire à propos de. L'affichage du formulaire hérité sera alors le suivant:



Exercice 1: Inclure dans votre formulaire de démarrage un menu dans lequel figurera l'option Aide/A propos de permettant de provoquer l'affichage du formulaire hérité. Essayer de modifier les propriétés des différents contrôles et en tirer les conclusions de ce que permet le modificateur choisi pour ces contrôles dans le formulaire de base, *protected*.

Exercice 2: Changer le modificateur d'un des contrôles de type label en *private* et tenter de modifier de nouveau les propriétés de ce contrôle dans le formulaire hérité.

Exercice 3: Ajouter un nouveau contrôle dans le formulaire de base et en tirer les conclusions pour le formulaire dérivé.

1.5. Création d'applications MDI

Lorsque nous créons une application basée Windows, nous pouvons utiliser un style différent pour l'interface utilisateur. Nous pouvons choisir une application ayant une interface simple document (SDI), une interface multiple document (MDI) ou utiliser une interface avec un style navigateur. Pour bien distinguer les deux premières interfaces, nous pouvons faire référence à des applications Windows connues comme le bloc-notes qui est une application SDI et comme Microsoft Excel qui est une application MDI.

Comment créer une application MDI?

- Dans une première étape, il faut créer un formulaire parent capable de contenir les différents documents. Pour ce, il suffit de placer la valeur de la propriété *IsMdiContainer* à *true*.

- Dans une deuxième étape, il faut ajouter un menu permettant d'invoquer les différents formulaires enfant.
- Dans une dernière étape, créer un ou plusieurs formulaires enfant et les appeler à partir du parent.

Exemple:

Nous créons un nouveau projet de type application Windows. Pour le formulaire créé par défaut et portant le nom Form1, nous en éditons les propriétés et nous définissons la propriété *IsMdiContainer* à *True*.

Nous ajoutons un menu principal dont le nom de l'objet par défaut est mainMenu1. Nous ajoutons un item et nous gérons l'événement du simple clic pour cet item.

Nous créons maintenant un deuxième formulaire devant apparaître lorsque l'on effectuera le simple clic sur l'item du menu.

Voici le contenu du code:

```

1  this.menuItem2.Index = 0;
2  this.menuItem2.Text = "&Open";
3  this.menuItem2.Click += new System.EventHandler(this.menuItem2_Click);
4
5  private void menuItem2_Click(object sender, System.EventArgs e)
6  {
7      Form2 ChildForm = new Form2();
8      ChildForm.MdiParent = this;
9      ChildForm.Show();
10 }

```

Commentaires:

La ligne 3 permet d'assurer la gestion de l'événement du simple clic sur l'item du menu dont le code figure aux lignes 5 à 10. La ligne 7 comprend l'instanciation de la classe Form2 associée au deuxième formulaire et la ligne 8 permet de définir que c'est un formulaire enfant MDI ayant comme parent le formulaire Form1 (this).

Pour organiser les fenêtres enfant sur le formulaire parent, nous pouvons utiliser la méthode *LayoutMdi* avec une énumération *MdiLayout* comprenant les quatre valeurs suivantes:

Membres	Descriptions
ArrangeIcons	Toutes les fenêtres enfant sont organisées dans la région cliente du formulaire MDI parent.
Cascade	Toutes les fenêtres enfant sont mises en cascade dans la région cliente du formulaire MDI parent.
TileHorizontal	Toutes les fenêtres enfant sont placées l'une en dessous de l'autre dans la région cliente du formulaire MDI parent.
TileVertical	Toutes les fenêtres enfant sont placées l'une à côté de l'autre dans la région cliente du formulaire MDI parent.

Exercice:

Veillez définir dans le menu de l'exemple précédent des options permettant de changer le mode d'organisation des fenêtres enfant suivant les trois derniers modes présentés.

Les formulaires parent et enfant peuvent également interagir en utilisant les techniques suivantes:

- Pour lister les fenêtres enfant qui sont propriété du parent, on peut agir sur la propriété *MdiList* des options du menu du parent. La liste des enfants sera liée à l'option du menu.
- Pour déterminer la fenêtre enfant active, nous pouvons utiliser la propriété *ActiveMdiChild* de la façon suivante:
Form `activeChild = this.ActiveMdiChild;`

2. Utilisation des contrôles.

2.1. Utilisation du contrôle *StatusBar*.

Une barre d'état est une fenêtre horizontale située dans le bas d'une fenêtre parent dans laquelle une application peut afficher des catégories variées d'informations d'état. La barre d'état peut être divisées en parties pour afficher plus d'un type d'information. Les contrôles *StatusBar* peuvent contenir des panneaux (*status bar panels*) qui affichent du texte ou des icônes pour indiquer un état ou une série d'icônes dans une animation qui indique l'état d'avancement d'un travail.

Pour placer une barre d'état dans le bas d'un formulaire, il suffit de double cliquer sur le contrôle *StatusBar* que l'on retrouve dans la boîte d'outils.

Exemple 1:

Pour mettre en pratique l'utilisation des barres d'état, nous allons prendre un formulaire dans lequel nous ajouterons une barre d'état comprenant un panneau indiquant l'heure courante lors du démarrage de l'application avec un rafraîchissement toutes les secondes.

En utilisant la barre d'outils, vous pouvez double cliquer sur *StatusBar* vous permettant ainsi d'insérer une barre d'état dans votre formulaire courant. Dans les propriétés de cette barre d'état, nous retrouvons la propriété *Panels* (Collection) permettant d'ajouter des panneaux dans la barre d'état. Une fois un nouveau panneau ajouté, nous retrouvons pour ce dernier les propriétés les plus importantes:

Propriétés	Description
<i>AutoSize</i>	Détermine le mode de fonctionnement du redimensionnement
<i>Alignment</i>	Permet de définir l'alignement du panneau dans la barre d'état
<i>BorderStyle</i>	Le type de bord affiché pour les côtés du panneau
<i>Icon</i>	L'icône (*.ico) qui doit être affichée dans le panneau
<i>Style</i>	Définit le style du panneau en utilisant une des valeurs reprises dans le type énuméré <i>StatusBarPanelStyle</i> .
<i>Text</i>	La chaîne de caractères affichée dans le panneau
<i>MinWidth</i>	La largeur minimale du panneau dans la barre d'état.

Dans notre exemple, nous choisirons les propriétés suivantes pour le panneau:

Autosize Contents
Alignment Center
BorderStyle Sunken
Text 22:22:10
Name Clock

Nous devons également changer une des propriétés de la barre d'état pour valider l'affichage des différents panneaux:

ShowPanels True

Si nous désirons mettre à jour le contenu du panneau avec l'heure courante, nous devons inclure la ligne de code suivante dans la fonction *InitializeComponent*:

Ligne de code modifiée: `this.Clock.Text=System.DateTime.Now.ToLongTimeString();`

Si nous désirons que cette heure soit mise à jour toutes les secondes, nous devons ajouter un timer. Ce contrôle est disponible dans la boîte d'outils sous le nom de contrôle *Timer*.

Si nous double cliquons sur ce contrôle, celui-ci s'ajoute dans notre projet. Nous pouvons éditer les propriétés de ce contrôle pour en modifier deux en particulier:

Interval permettant de définir l'intervalle en milli secondes: 1000

Enable permettant d'activer ce contrôle: True.

Si nous accédons à la gestion des événements, nous retrouvons la propriété Tick à laquelle nous donnons le nom de la fonction devant être appelée toutes les n milli secondes. Nous choisirons comme nom de méthode: miseajour. Nous retrouvons les lignes de code suivant:

```
1        this.timer1.Tick += new System.EventHandler(this.miseajour);  
2  
3        private void miseajour(object sender, System.EventArgs e)  
4        {  
5                this.Clock.Text=System.DateTime.Now.ToLongTimeString();  
6        }
```

Comentaires:

La ligne 1 permet la mise en place de la gestion de l'événement Tick associé à l'objet timer1. Les lignes 3 à 6 comprennent la méthode devant être appelée lors de l'événement Tick. Nous retrouvons la ligne de code permettant la mise à jour du contenu du panneau avec l'heure courante.

Exemple 2: comment insérer une barre de progression dans la barre de status. Nous reprenons notre exercice précédent et nous ajoutons une nouvelle barre de progression dans la barre d'état. Pour ce, nous allons créer notre contrôle de façon dynamique sans utiliser l'interface graphique.

Voici les lignes de code que nous devons ajouter:

Dans la classe associée à notre formulaire, nous allons ajouter une référence à la classe *ProgressBar* sous la forme suivante:

```
private System.Windows.Forms.ProgressBar progressBar2;
```

Nous allons ensuite définir les propriétés de cette barre de progression et l'ajouter parmi les contrôles de la barre d'état.

```

1   this.progressBar2 = new System.Windows.Forms.ProgressBar();
2   this.progressBar2.Location = new System.Drawing.Point(0, 0);
3   this.progressBar2.Name = "progressBar1";
4   this.progressBar2.TabIndex = 2;
5   statusBar1.Controls.Add(progressBar2);

```

Commentaires:

La ligne 1 permet la création de l'objet, la ligne 2 permet de définir le positionnement de la barre de progression par rapport à sa fenêtre parent et la ligne 5 permet d'ajouter la barre de progression comme contrôle supplémentaire à la barre d'état. Nous pouvons supprimer ou tester l'appartenance d'un contrôle à une collection en utilisant les méthodes suivantes:

<code>statusBar1.Controls.Remove(progressBar2)</code>	Cette méthode permet de supprimer un contrôle de la collection renvoyée par la propriété <i>Controls</i> .
<code>statusBar1.Controls.Contains(progressBar2)</code>	Cette méthode permet de déterminer si un contrôle fait partie de la collection.

Voici la liste des différentes méthodes accessibles:

Propriété	Description
Add	Permet d'ajouter un contrôle spécifique à une collection
AddRange	Permet d'ajouter un tableau de contrôles à une collection
Clear	Permet de supprimer tous les contrôles d'une collection
Contains	Permet de déterminer si un contrôle fait partie ou pas d'une collection.
Remove	Supprime un contrôle spécifique d'une collection
RemoveAt	Supprime un contrôle correspondant à un index de position donné dans une collection.
ToString	Retourne une chaîne de caractères représentant l'objet courant.
IndexOf	Récupère l'index qu'occupe un contrôle dans une collection.
GetEnumerator	Retourne une énumération.

2.2. Utilisation du contrôle *ToolBar*.

Une barre d'outils est une interface graphique utilisateur alternative aux menus. Une barre d'outils comprend un ensemble de boutons qui sont représentés par la classe *ToolBarButton* dans le Framework .NET.

Pour utiliser une barre d'outils dans une application, il suffit de choisir l'objet correspondant dans la barre d'outils du visual studio .NET et ensuite d'y ajouter des boutons en utilisant la propriété *Buttons*.

Exemple 1: nous allons mettre en place une barre d'outils comprenant un bouton permettant l'affichage d'une boîte de dialogue. Cet affichage doit être rendu possible également par la sélection d'un des items du menu. En utilisant la barre d'outils du .NET, vous pouvez double cliquer sur *ToolBar* vous permettant ainsi d'insérer une barre d'outils dans votre formulaire courant. Dans les propriétés de cette barre d'outils, nous retrouvons la propriété *Buttons* (Collection) permettant d'ajouter des boutons dans la barre d'outils. Les boutons peuvent apparaître sous des styles différents en choisissant la valeur à associer à la propriété *ToolBarButtonStyle*:

Valeur	Description
DropDownButton	Liste déroulante qui affiche un menu ou une autre fenêtre lorsque l'on clique dessus.
PushButton	Un bouton standard avec effet tri dimensionnel
Separator	Un espace ou une ligne entre des boutons de la barre d'outils
ToggleButton	Bouton apparaissant comme étant enfoncé lorsque l'on clique dessus et ce jusqu'au prochain clic de la souris sur ce même bouton.

Lorsque un bouton a été ajouté, on peut choisir l'image qui lui est affectée. Cette image se choisit dans une liste d'images en spécifiant son index. Cette liste est créée en utilisant un contrôle ImageList. Pour l'ajouter dans notre application, il suffit de la sélectionner dans la barre d'outils du .NET en double cliquant sur ImageList. L'objet ainsi créé dans notre application comprend la propriété Images correspondant à une collection qui nous permettra d'ajouter des images sous des formats divers.

Lorsque cette liste est créée, il est alors possible de la sélectionner pour notre barre d'outils grâce à la propriété ImageList. Nous pouvons maintenant choisir dans cette liste une des images pour l'affecter à chacun des boutons grâce à la propriété ImageIndex.

Dès que le bouton est créé, nous pouvons gérer l'événement du simple clic. Nous retrouvons alors le code suivant:

```

1  this.toolBar1.ButtonClick += new
    System.Windows.Forms.ToolBarButtonClickEventHandler(this.toolBar1_ButtonClick);

2  private void toolBar1_ButtonClick(object sender,
    System.Windows.Forms.ToolBarButtonEventArgs e)
3  {
4      Form2 test = new Form2();
5      test.ShowDialog(this);
6  }
```

Commentaires:

La ligne 1 permet de mettre en place la gestion de l'événement et de l'associer à l'exécution de la méthode toolBar1_ButtonClick. Dans cette méthode, nous retrouvons l'instanciation de la classe Form2 qui correspond à un nouveau formulaire que nous avons créé précédemment. La ligne 5 permet de faire apparaître le formulaire sous forme modale.

Nous ne disposons que d'un seul événement pour l'ensemble des boutons que nous voudrions par la suite ajouter à notre barre d'outils. Pour déterminer lequel des boutons est à l'origine de l'événement, nous pouvons utiliser le deuxième paramètre de la méthode:

```
System.Windows.Forms.ToolBarButtonEventArgs e
```

Celui-ci nous permettra de récupérer soit l'index du bouton, soit l'objet correspondant à ce bouton. Voici le code correspondant:

```

1  if (e.Button == toolBarButton1)
2      MessageBox.Show("Premier bouton");
3  if (e.Button == toolBarButton2)
4      MessageBox.Show("Deuxième bouton");
5  MessageBox.Show("Index du bouton:" + this.toolBar1.Buttons.IndexOf(e.Button) );
```

Commentaires:

Les lignes 1 et 2 utilisent la propriété *Button* de la référence *e* utilisée comme deuxième argument et pointant vers l'objet ayant provoqué l'événement.

Nous pouvons tester si l'objet correspond au premier ou au deuxième bouton, ceux-ci étant référencés par `toolBarButton1` et `toolBarButton2`.

A la ligne 5, nous affichons l'index correspondant à la position du bouton dans la barre d'outils. Il est préférable malgré tout d'utiliser les références du fait que les séparateurs que l'on utilise pour la mise en forme sont également indexés et que l'on pourrait à tout moment insérer de nouveaux boutons nécessitant de remanier notre code pour mettre à jour les indexes.

Exemple 2: insertion d'une liste déroulante dans la barre d'outils. Cette utilisation est assez fréquente: il suffit pour s'en convaincre de regarder les barres d'outils de Microsoft Word permettant de sélectionner la police de caractères, la taille etc.

Cet exemple est similaire à celui repris dans le paragraphe précédent comprenant l'insertion d'une barre de progression dans la barre d'état. Nous allons ajouter après le dernier bouton un séparateur et nous utiliserons cet emplacement pour y placer notre liste.

```
1 private System.Windows.Forms.ComboBox ComboMenu;
2 ComboMenu = new System.Windows.Forms.ComboBox()
3 ComboMenu.Location = new System.Drawing.Point(toolBarButton3.Rectangle.X,
4         toolBarButton3.Rectangle.Y)
5 ComboMenu.Size = new System.Drawing.Size(121, 21)
6 ComboMenu.Items.Add("test1")
7 ComboMenu.Items.Add("test2")
8 this.toolBar1.Controls.Add(ComboMenu)
```

Commentaires:

La ligne 1 et 2 permettent la création dynamique de l'objet sans utiliser l'interface graphique. Nous allons définir la position de la liste déroulante en tenant compte de la position du séparateur placé dans notre barre d'outils.

Aux lignes 5 et 6 nous ajoutons deux items dans la liste.

A la ligne 7, nous ajoutons la liste déroulante comme contrôle à la barre d'outils.

Exercice: dans les boîtes d'outils, il est parfois intéressant de changer l'image associée à un bouton. Pensons par exemple à un cadenas ouvert et à un cadenas fermé pour indiquer à l'utilisateur qu'une ressource est verrouillée ou pas. Mettons en place ce dispositif en faisant en sorte que l'image change suite au clic de souris sur le bouton correspondant.

2.3. Sélection des boîtes de dialogue pré configurées.

Les boîtes de dialogue sont utilisées pour interagir avec l'utilisateur. Studio .NET fournit quelques boîtes de dialogue pré configurées qui peuvent être utilisées dans des applications Windows Forms pour interagir avec l'utilisateur. Tout comme les boîtes de dialogues personnalisées, nous pouvons tester la valeur retournée en utilisant la propriété `DialogResult`. Voici la liste des différentes boîtes:

Boîte de dialogue	Description
<code>OpenFileDialog</code>	Permet aux utilisateurs de parcourir l'arborescence de l'ordinateur et

	de choisir un fichier.
SaveFileDialog	Sélectionne les fichiers qui doivent être sauves et l'emplacement ou ils doivent l'être.
ColorDialog	Permet à l'utilisateur de sélectionner une couleur dans une palette.
FontDialog	Propose les différentes polices de caractères qui sont installées sur votre système.
PrintDialog	Permet la sélection d'une imprimante et la modification de ses paramètres.
PageSetupDialog	Permet de choisir les détails de mise en page.
PrintPreviewDialog	Affiche un document comme si il devrait apparaître lors de l'impression.

Exemple 1: prenons le cas d'un formulaire dans lequel nous retrouvons un contrôle permettant l'édition de texte au format rtf . Nous devons mettre en place une option du menu permettant de choisir un fichier dont le contenu s'affichera dans le contrôle. La boîte de dialogue pré configurée avec laquelle nous allons travailler est *OpenFileDialog*. Une première démarche est de créer l'option dans le menu et d'afficher simplement le nom du fichier sélectionné.

```

1 private void menuItem2_Click(object sender, System.EventArgs e)
2 {
3     OpenFileDialog test = new OpenFileDialog();
4     if (test.ShowDialog(this)==DialogResult.OK)
5         MessageBox.Show(test.FileName);
6 }

```

Commentaires:

Nous retrouvons la méthode associée à l'événement du simple clic sur l'item du menu. Nous retrouvons à la ligne 3 l'instanciation de la classe associée à la boîte de dialogue pré configurée *OpenFileDialog*. La ligne 5 permet l'affichage du nom du fichier sélectionné si nous avons fermer la boîte de dialogue en cliquant sur le bouton Ouvrir.

Notre deuxième démarche sera d'utiliser une méthode du contrôle *RichTextBox* pour ouvrir le fichier et en afficher le contenu. Voici le code correspondant:

```

1 private void menuItem2_Click(object sender, System.EventArgs e)
2 {
3     OpenFileDialog test = new OpenFileDialog();
4     if (test.ShowDialog(this)==DialogResult.OK)
5     {
6         richTextBox1.LoadFile(test.FileName.ToString());
7     }
8 }

```

Exercice: mettre en place l'option du menu ainsi que l'utilisation de la boîte de dialogue *SaveFileDialog* pour enregistrer le contenu du contrôle dans le fichier sélectionné à l'emplacement voulu.

2.4. Validation des entrées utilisateur.

La validation des données introduites par l'utilisateur peut s'effectuer au moyen de l'événement *Validating*. Nous pourrions le cas échéant envoyer un message d'erreur à l'utilisateur pour lui notifier le contrôle qui est en erreur ainsi qu'un message à afficher sous la forme d'une info bulle.

Exemple 1: Nous considérerons une zone de texte dans laquelle un utilisateur doit impérativement introduire un nombre entier. La validation s'effectue automatiquement juste avant que le contrôle ne perde le focus.

Nous retrouvons dans notre zone de texte l'événement *Validating* que nous pouvons gérer. Nous allons lui associer une méthode dont voici le code:

1	<code>private void test(object sender,</code>
	<code>System.ComponentModel.CancelEventArgs e)</code>
2	<code>{</code>
3	<code>try</code>
4	<code>{</code>
5	<code>int x=Int32.Parse(this.textBox1.Text);</code>
6	<code>e.Cancel=false;</code>
7	<code>}</code>
8	<code>catch</code>
9	<code>{</code>
10	<code>e.Cancel=true;</code>
11	<code>MessageBox.Show("Necessite un entier");</code>
12	<code>}</code>
13	<code>}</code>

Commentaires:

A la ligne 5, nous essayons de convertir le contenu de la zone de texte en un entier en utilisant la méthode statique *Parse* de la classe *Int32*. L'appel à cette méthode est intégré dans un bloc *try catch* pour récupérer les erreurs liées à la conversion et dans le cas où une erreur s'est produite, nous plaçons la valeur *true* dans la propriété *Cancel*. L'événement ne sera provoqué que si le contrôle qui doit recevoir le focus a la propriété *CausesValidation* à *true*. Le fait de forcer la valeur de la propriété *Cancel* permettra donc après l'évaluation des données entrées par l'utilisateur d'autoriser ou pas le déplacement du focus sur cet autre contrôle.

Dans ce cas précis, nous pouvons penser que l'utilisateur aura des facilités à déterminer le contenu étant à la source de l'erreur puisque il est évalué immédiatement avant que le changement de focus ne soit effectif. Nous pouvons envisager le cas de figure où ce n'est que lorsque l'utilisateur provoque un événement tel que le clic de la souris sur un bouton que l'ensemble des contrôles présents sur le formulaire est validé. Dans ce cas il serait intéressant qu'un message apparaisse en vis-à-vis de chacun des contrôles ayant un contenu erroné. Pour ce, nous allons utiliser le contrôle *ErrorProvider* présent dans la boîte d'outils.

Voici le code associé à la méthode liée à l'événement du clic sur le bouton du formulaire.

```
1 private void button1_Click(object sender, System.EventArgs e)
2 {
3     try
4     {
5         int x=Int32.Parse(this.textBox1.Text);
6         errorHandler1.SetError(textBox1, "");
7     }
8     catch
9     {
10        errorHandler1.SetError(textBox1,"Necessite un entier");
11    }
12 }
```

Commentaires:

Nous retrouvons dans les lignes 6 et 10, l'utilisation de la référence `errorProvider1` dont voici la déclaration et l'initialisation que nous retrouvons dans notre code:

```
private System.Windows.Forms.ErrorProvider errorProvider1;  
this.errorProvider1 = new System.Windows.Forms.ErrorProvider();
```

La méthode `SetError` permet de définir le contrôle nécessitant la gestion d'une erreur. Si la chaîne de caractère passée en deuxième paramètre est vide, l'icône d'erreur qui a été choisie par la propriété `Icon`, n'apparaîtra à côté du contrôle en erreur. Si la chaîne n'est pas vide, l'icône apparaît et le texte d'erreur est géré sous forme d'info bulle.

3. Construction de contrôles.

Nous disposons pour les formulaires d'un choix important de contrôles répondant à un grand nombre des besoins que nous retrouvons dans nos applications. Nous pouvons combiner ces contrôles avec des contrôles existants étendus ou être les auteurs de nos propres contrôles personnalisés. Nous aborderons dans le cadre de cette section trois aspects de la construction de contrôles: les contrôles étendus, les contrôles composites et les contrôles personnalisés.

3.1. Les contrôles étendus.

Nous pouvons personnaliser n'importe quel contrôle de formulaire Windows en créant une classe dérivée de la classe de base correspondant. Nous pouvons dès lors surcharger les propriétés, méthodes et événements ou créer nos propres membres. Nous allons pour mettre en lumière cette possibilité prendre comme exemple une zone d'édition contenant une valeur numérique et prévoir l'ajout de deux membres permettant de définir une valeur minimale et maximale autorisées pour le nombre introduit par l'utilisateur.

Exemple 1:

Vous pouvez créer la classe dérivée manuellement ou utiliser la fenêtre de visualisation des classes. Si cette fenêtre n'est pas visible, il suffit de la choisir dans la liste proposée dans l'option affichage du menu. Dans cette fenêtre, nous choisissons le projet dans lequel doit être ajoutée la classe, nous cliquons sur le bouton droit de la souris et nous choisissons 'Ajouter une classe' parmi les options proposées.

Nom de la classe: `TextBoxInt`

Espace de noms de la classe de base: `System.Windows.Forms`

Nom de la classe de base: `TextBox`.

Nous obtenons alors un nouveau fichier dans notre solution contenant la déclaration de la classe.

```
1 public class TextBoxInt : System.Windows.Forms.TextBox  
2 {  
3     public TextBoxInt()  
4     {  
5  
6     }  
7 }
```

Nous allons ajouter deux nouveaux membres à cette classe: le membre MinValue et le membre MaxValue permettant de tester si le nombre introduit dans la zone de texte se situe bien dans une fourchette donnée.

```
public int MinValue;  
public int MaxValue;
```

Comme ce nouveau contrôle ne figure pas dans la boîte d'outils, nous devons le créer dynamiquement dans notre formulaire. Pour mettre en évidence la différence de syntaxe, nous avons ajouté un contrôle de la classe de base. Voici le code :

```
private System.Windows.Forms.TextBox textBox1;  
private TextBoxInt textBox2;  
  
this.textBox1 = new System.Windows.Forms.TextBox();  
this.textBox2 = new TextBoxInt();  
  
this.textBox2.Location = new System.Drawing.Point(96, 88)  
this.textBox2.Name = "textBox2"  
this.textBox2.Text = "textBoxInt"  
this.Controls.Add(textBox2)
```

Nous souhaitons dans notre exemple que la validation du nombre entré se fasse automatiquement sans devoir créer nous même une délégation comme nous l'avons fait jusqu'à présent pour gérer nos événements. Lorsque nous parcourons les différentes méthodes des classes de base associées aux différents contrôles, nous retrouvons certaines méthodes déjà associées à des événements. C'est le cas par exemple de la méthode OnValidating que l'on a dans la classe TextBoxBase. Nous allons surcharger cette méthode dans notre classe pour valider automatiquement notre nombre entier. Toutes les méthodes commençant par 'On...' sont généralement des méthodes associées à l'événement dont le nom suit 'Validating'.

```
1     protected override void OnValidating(System.ComponentModel.CancelEventArgs e)  
2     {  
3         int x;  
4         try  
5         {  
6             x=Int32.Parse(this.Text);  
7             if ((x<this.MinValue) || (x>this.MaxValue))  
8             {  
9                 e.Cancel=true;  
10                System.Windows.Forms.MessageBox.Show("Vous devez introduire  
11                un  
12                nombre entre"+this.MinValue+" et "+this.MaxValue);  
13                return;  
14            }  
15        }  
16        catch  
17        {  
18            e.Cancel=true;  
19            return;  
20        }  
21    }
```

Commentaires:

Dans un premier temps, nous convertissons le contenu de la zone de texte en un entier. Cette conversion s'effectue dans un bloc try catch pour gérer l'exception qui pourrait apparaître du

fait que l'utilisateur pourrait entrer autre chose qu'une chaîne de caractères convertible en un entier. Dans le cas où cette conversion s'est bien déroulée, il ne reste plus qu'à tester la validité de l'entier qui doit être compris entre la valeur `MinValue` et `MaxValue`.

L'événement `Validating` apparaît lorsque le contrôle va perdre son focus. Nous pouvons bloquer ce mécanisme en jouant sur la propriété `Cancel` accessible par la référence `e` passée en paramètre à la méthode `OnValidating`.

Ligne 9: `e.Cancel=true` permet de valider le changement de focus

Ligne 13: `e.Cancel=false` bloque le changement du focus du fait que l'entier ne satisfait pas aux conditions fixées par les membres `MinValue` et `MaxValue`.

Exemple 2: Nous pouvons reprendre l'exemple précédent et l'adapter de sorte que l'utilisateur reçoive un message d'erreur si il essaie de taper le moindre caractère dans la zone d'édition qui ne serait pas un chiffre.

Nous allons utiliser à nouveau une surcharge d'une des méthodes de la classe de base liées aux événements. Dans notre exemple, nous prendrons la méthode `OnKeyPress` qui comme son nom l'indique, est appelée lorsque l'on appuie sur la moindre touche lorsque notre contrôle a le focus.

```
1     protected override void OnKeyPress(System.Windows.Forms.KeyPressEventArgs e)
2     {
3         if (char.IsDigit(e.KeyChar)==true)
4             {
5                 e.Handled=false;
6                 return;
7             }
8         if (char.IsControl(e.KeyChar))
9             {
10                e.Handled=false;
11                return;
12            }
13        e.Handled=true;
14    }
```

Commentaires:

Dans la ligne 3, nous faisons appel à la méthode statique `IsDigit` de la classe `char` pour vérifier si le caractère introduit au clavier est bien un chiffre décimal. Dans le cas où nous voulons passer l'événement à Windows du fait que la condition est bien remplie, nous l'indiquerons en positionnant la propriété `e.Handled` à `false`. Dans le cas d'une erreur d'introduction, nous empêchons l'événement de se propager en positionnant la propriété `e.Handled` à `true`.

Dans la ligne 8, nous faisons appel à la méthode statique `IsControl` parce que tout caractère qui ne serait pas numérique mais correspondrait à un contrôle doit être propagé (ex: le back slash).

3.2. Les contrôles composites.

Une autre façon de créer des contrôles est de combiner des contrôles existants. Tous les contrôles composites dérivent de la classe `System.Windows.Forms.UserControl`.

Reprenons notre exemple précédent et adaptons le pour ajouter automatiquement un contrôle d'erreur qui soit lié à notre zone d'édition

Exemple1:

Pour créer un contrôle composite, il suffit, dans notre projet, de sélectionner l'option 'Ajouter un contrôle utilisateur' dans le menu 'projet'. Nous choisirons comme nom de contrôle TextBoxInt2.

Dans ce contrôle utilisateur, nous plaçons un objet de la classe TextBoxInt développé dans l'exemple précédent et nous ajoutons un objet de la classe ErrorProvider.

Nous modifions le code présent dans la surcharge de la méthode OnValidating:

```
1     public bool ErreurNombre;
2     protected override void OnValidating(System.ComponentModel.CancelEventArgs e)
3     {
4         int x;
5         try
6         {
7             x=Int32.Parse(this.Text);
8             if ((x<this.MinValue)|| (x>this.MaxValue))
9                 {
10                    ErreurNombre=true;
11                }
12            else ErreurNombre=false;
13        }
14    catch
15    {
16        ErreurNombre=true;
17    }
18    e.Cancel=false;
19 }
```

Commentaires:

Nous utiliserons dans notre classe TextBoxInt, un membre public que nous appellerons ErreurNombre. Nous en retrouvons la déclaration à la ligne 1.

Notre objectif est de faire en sorte que ce soit le contrôle conteneur qui assure la gestion finale de l'événement validating pour faire apparaître ou pas le message d'erreur. La propagation des erreurs s'effectue du contrôle sur lequel est apparu l'erreur pour remonter ensuite vers l'ensemble des différents parents. Si nous bloquons la propagation des erreurs avec la propriété e.Cancel, notre contrôle conteneur ne recevra jamais cette erreur.

Nous allons donc communiquer la condition d'erreur au travers d'un membre dont on pourra récupérer la valeur dans le contrôle conteneur.

Nous mettons maintenant en place la gestion de la validation pour le conteneur. Voici le code:

```
1     this.Validating += new
2     System.ComponentModel.CancelEventHandler(this.TextBoxInt2_Validating);
3     private void TextBoxInt2_Validating(object sender,
4     System.ComponentModel.CancelEventArgs e)
5     {
6         if (this.textBox1.ErreurNombre==true)
7             this.errorProvider1.SetError(this.textBox1,"nombre en dehors des limites");
8         else this.errorProvider1.SetError(this.textBox1,"");
9     }
```

3.3. Les contrôles personnalisés.

Nous pouvons utiliser des contrôles personnalisés en dessinant soi même les différents éléments de l'interface utilisateur en faisant appel à l'objet Graphics GDI+ dans la méthode surchargée OnPaint.

Pour créer un contrôle personnalisé, nous dérivons une classe de la classe de base Control qui dessine un rectangle sur notre formulaire.

Exemple1: nous allons nous limiter à créer un contrôle personnalisé dans lequel nous écrivons un simple texte.

Pour ajouter un contrôle personnalisé à notre projet, nous devons prendre l'option Ajouter un item à partir du menu Projet et sélectionner le contrôle personnalisé. Nous retrouverons dans le code une surcharge de la méthode OnPaint dans laquelle nous ajouterons notre code:

```
1  protected override void OnPaint(PaintEventArgs pe)
2  {
3      pe.Graphics.DrawString("Bonjour", new Font("Arial", 12), new
        SolidBrush(Color.Red), 0, 0);
4      base.OnPaint(pe);
5  }
```

Commentaire:

Le contenu de la ligne 3 nous permet de dessiner du texte dans le contrôle en utilisant l'objet Graphics.

4. Utilisation de données dans les formulaires.

4.1. Utilisation de ADO.NET

ADO.NET est un ensemble de classes qui permettent à des applications basées .NET de pouvoir lire et mettre à jour des informations dans des bases de données et autres données stockées. Nous pouvons accéder à ces classes au travers de l'espace de nom System.Data. ADO.NET procure un large variété d'accès à des sources de données multiples incluant les bases de données Microsoft SQL Serveur, les bases de données compatibles OLE-DB, des sources non relationnelles comme Microsoft Exchange serveur et les documents XML. L'accès à une base de données ou à tout autre support de données nécessite à partir d'une application d'établir une connexion. Dans ADO.NET, nous pouvons créer et gérer une connexion en utilisant l'objet Connection. Toute application utilise un objet Connection pour communiquer avec la base de données.

Il existe en fait deux sortes de connexions: les connexions sur un serveur Microsoft SQL 7.0 ou ultérieur grâce à l'objet SqlConnection et les connexions pour l'accès aux bases de données au travers d'OLE DB grâce à l'objet OleDbConnection.

Exemple1: création d'une connexion avec une base de données de type Access.

Nous allons devoir utiliser dans notre projet un objet de type OleDbConnection. Pour placer un tel objet, nous pouvons utiliser la boîte d'outils, Data et OleDbConnection. Une des propriétés importante de cet objet est ConnectionString permettant de paramétrer les caractéristiques de notre base de données: fournisseur OLE DB, connexion, propriétés avancées (paramètres réseau, délai d'attente de connexion et autorisations d'accès). Ces choix

peuvent s'opérer au travers d'une interface présentant les trois onglets fournisseur, connexion, et propriétés avancées.

```
private System.Data.OleDb.OleDbConnection oleDbConnection1;  
this.oleDbConnection1 = new System.Data.OleDb.OleDbConnection();  
this.oleDbConnection1.ConnectionString=@"Provider=Microsoft.Jet.OLEDB.4.0;  
Data Source=D:\bar.mdb;"
```

Une fois connecté à une base de données, nous allons pouvoir utiliser des objets Command pour accéder directement aux données dans un environnement connecté. Ces objets utilisent des requêtes SQL ou des procédures stockées pour récupérer les données. Les commandes sont envoyées à travers la connexion et les résultats sont retournés sous forme d'un flux qui peut être lu par les objets DataReaders ou envoyé dans un objet de type DataSet.

```
1     this.oleDbCommand1=new System.Data.OleDb.OleDbCommand()  
2     this.oleDbCommand1.Connection=oleDbConnection1  
3     //this.oleDbCommand1=oleDbConnection1.CreateCommand()  
4     oleDbCommand1.CommandText="SELECT * FROM ADMETUD"  
5     this.oleDbConnection1.Open()
```

Commentaires:

Les deux premières lignes peuvent être remplacées par la ligne 3 apparaissant dans le code en commentaire. Nous plaçons dans la propriété oleDbCommand1.CommandText la requête SQL devant envoyer le jeu d'enregistrement. Il ne reste plus qu'à accéder au flux par un objet DataReader.

```
1     private System.Data.OleDb.OleDbDataReader test  
2     this.test=this.oleDbCommand1.ExecuteReader()  
3     test.Read()  
4     this.textBox1.Text=test.GetString(1)
```

Commentaires:

Le code de la ligne 3 permet de forcer la lecture du premier enregistrement et le positionnement sur l'enregistrement suivant. Le code de la ligne 4 permet d'effectuer la lecture du contenu de la colonne 1 qui doit être convertie sous forme d'une chaîne de caractère.

Attention: le temps que l'on accède aux données par un objet DataReader force la connexion associée à rester occupée jusqu'à ce que l'on ferme cet objet. Pour cette raison, on doit fermer le plus rapidement possible le DataReader dès que l'on a terminé de consulter le jeu de résultats.

L'objet Command comprend une collection Parameters qui permet de fournir de arguments à la requête SQL. Comme toutes les commandes, même la commande de suppression d'un enregistrement, nous devons fournir des requêtes SQL. La suppression d'un enregistrement en fonction du numéro d'identification se fera de la sorte:

delete * from ADMETUD where NADM=xxx ou xxx doit être le numéro de l'étudiant que l'on doit supprimer de la table. Nous allons donc insérer dans notre code une commande de suppression d'un enregistrement basé sur cette requête en sachant que le numéro xxx doit être passé en argument par l'emploi de la collection Parameters.

```

1  this.oleDbDeleteCommand1 = new System.Data.OleDb.OleDbCommand()
2  this.oleDbDeleteCommand1.CommandText = "DELETE FROM ADMETUD WHERE (NADM = ?)"
3  this.oleDbDeleteCommand1.Connection = this.oleDbConnection1
4  this.oleDbDeleteCommand1.Parameters.Add("NADM",
      System.Data.OleDb.OleDbType.Integer, 0, "NADM")

```

Commentaires:

La ligne 2 comprend dans la requête SQL un '?' qui d'un point de vue syntaxique correspond à un paramètre portant le même nom que le champ sur lequel le filtrage porte.

4.2. Utilisation des objets *DataAdapter* et *DataSet*.

Un objet *DataSet* représente une copie locale des données à partir de la source. Cet objet doit être capable d'interagir avec la source au travers d'un objet *DataAdapter* qui agit comme un pont entre le *DataSet* et la source de données. La classe *DataAdapter* représente un jeu de commandes et une connexion qui vous permet de remplir un *DataSet* et mettre à jour la source de données. Nous retrouverons notamment une commande de sélection d'enregistrement, une commande de suppression, une commande d'insertion et de mise à jour.

Pour mettre en évidence les différentes étapes de création de notre *DataAdapter*, nous allons envisager comme exercice la gestion des ardoises au cercle des étudiants. Nous utiliserons une base de données au format Access comprenant dans une première phase, une table reprenant les différents clients. Nous nous limiterons aux champs suivants:

Reference	de type numérique auto incrémenté servant d'index
Nom	de type caractère (50 caractères)
Prenom	de type caractère (50 caractères)

Une fois cette table créée, nous pouvons créer notre projet.

Exercice 1: création d'un objet de type *DataAdapter*.

Etape 1: ajouter dans notre projet une connexion de type *OleDbConnection*.

- 1- A partir de la boîte d'outils, double cliquer sur le contrôle *OleDbConnection*.
- 2- Dans la fenêtre des propriétés de l'objet ainsi créé, cliquer sur la propriété *ConnectionString*, cliquer ensuite sur la flèche et cliquer alors sur *New Connection*.
- 3- Dans l'onglet fournisseur, nous choisirons *Microsoft Jet 4.0 OLE DB Provider*.
- 4- Dans l'onglet connexion, nous pourrions nous déplacer dans l'arborescence du disque dur et choisir la base de données associée à la gestion du cercle des étudiants.

Etape 2: Ajouter dans notre projet un objet *OleDbDataAdapter*.

- 1- A partir de la boîte d'outils, double cliquer sur le contrôle *OleDbDataAdapter*.
- 2- Un assistant va alors apparaître. Après avoir appuyé une première fois sur le bouton suivant, nous devons choisir la connexion par laquelle l'objet devra passer pour accéder à la base de données. Il suffira de choisir la connexion créée au point précédent et apparaissant dans notre exemple sous la forme:
ACCESS.C:\DataBaseSample\BAR.mdb.Admin

- 3- Dans l'écran suivant, nous choisirons le type de requêtes pour pouvoir accéder aux données mais aussi pour pouvoir les insérer, les supprimer ou les modifier. L'écran suivant nous proposera de créer les différentes requêtes. Le bouton de construction de requêtes nous permettra sous une interface graphique de pouvoir sélectionner les différents champs dans les tables ad hoc et de créer ainsi notre requête SQL de sélection. Nous choisissons tous les champs de notre table et nous obtenons une requête semblable à celle-ci: `select Bar.* from Bar`. Le bouton des paramètres avancés de la requête nous permet de choisir la création des requêtes d'ajout, d'insertion et de suppression mais également la méthode d'accès concurrente de type optimiste.

Etape 3: Nous retrouverons dans le code ainsi généré, les points suivants:

- 1- Création des différents objets Command et initialisation de la propriété CommandText de chacun d'entre eux pour leur affecter la requête SQL permettant la sélection, l'insertion, la suppression et la modification. La propriété Connection de ces objets doit correspondre avec l'objet connection créé au point précédent et permettant l'accès à la base de données.
- 2- Initialisation des propriétés SelectCommand, DeleteCommand, InsertCommand et UpdateCommand de l'objet OleDbDataAdapter en fonction des différents objets Command créés précédemment.

Exercice 2: création d'un objet de type DataSet

Pour générer notre DataSet, nous allons cliquer sur le bouton droit de la souris sur l'objet de type OleDbDataAdapter. Nous choisissons alors dans le menu contextuel l'option Générer un DataSet. En acceptant les valeurs proposées par défaut, nous retrouvons alors un objet de type DataSet sous la référence DataSet11 ajouté à notre projet

Si nous ouvrons l'explorateur de solution, nous retrouverons un fichier de type DataSet1.xsd qui nous donne une représentation graphique de notre DataSet. Ce fichier est en fait au format XML et nous pouvons en visualiser le contenu en cliquant sur XML dans le bas à gauche de la fenêtre.

4.3. Lier des données à des contrôles.

L'idée est maintenant de faire en sorte que l'on puisse faire apparaître dans notre formulaire des contrôles dont le contenu serait lié à un champ particulier de notre table. Nous choisirons dans un premier temps une zone d'édition associée à chaque champ.

Exercice 1: Liaison entre des contrôles simples et des données.

Après avoir ajouté deux zones de textes, nous allons visualiser les propriétés de chacun de ces objets et étendre dans l'arborescence l'option DataBindings. Nous y éditerons alors la propriété Text et nous choisirons les champs associés à ces zones. Avant toute compilation, il est important de remplir le DataSet et ce par la commande suivante qui sera ajoutée

manuellement: `this.oleDbDataAdapter1.Fill(this.dataSet11);`

En analysant notre code, nous retrouvons les lignes suivantes qui ont été ajoutées:

```
this.textBox1.DataBindings.Add(new System.Windows.Forms.Binding("Text",  
this.dataSet11, "Clients.Nom"));
```

```
this.textBox2.DataBindings.Add(new System.Windows.Forms.Binding("Text",
this.dataSet11, "Clients.Prenom"));
```

Lorsque nous allons exécuter notre programme, les nom et prénom du premier étudiant encodé apparaîtront dans notre formulaire. Nous souhaitons maintenant pouvoir ajouter des boutons dans notre formulaire permettant de nous positionner sur le premier ou le dernier enregistrement ou de se déplacer sur l'enregistrement précédent ou suivant.

Nous pouvons nous déplacer dans le DataSet en provoquant la mise à jour de l'affichage en utilisant un objet CurrencyManager. Lorsque une source de données est liée au contrôle d'un formulaire, un objet CurrencyManager sera automatiquement associé. Le CurrencyManager est un objet qui permet la synchronisation entre les contrôles et la source de données. Pour chaque source différente utilisée, nous retrouverons un objet CurrencyManager différent mais cela signifie que si des zones de texte différentes sont liées aux champs d'une même table, nous n'aurons qu'un seul objet utilisé.

Chaque formulaire Windows possède un objet de type BindingsContext qui garde une trace de tous les objets CurrencyManager utilisés.

Pour pouvoir mettre en place nos boutons permettant le déplacement, nous devons lier les zones de texte non plus au DataSet directement mais à une table. Nous modifierons donc notre code de la façon suivante:

```
this.textBox1.DataBindings.Add(new System.Windows.Forms.Binding("Text",
this.dataSet11.Tables["Clients"], "Nom"));
this.textBox2.DataBindings.Add(new System.Windows.Forms.Binding("Text",
this.dataSet11.Tables["Clients"], "Prenom"));
```

Nous commencerons par ajouter le bouton permettant de se déplacer sur l'enregistrement suivant dans notre DataTable. Nous retrouverons dans la méthode associée à l'événement du simple clic sur le bouton, le code suivant:

```
1 private void NextRecord_Click(object sender, System.EventArgs e)
2 {
3     CurrencyManager CM =
4         (CurrencyManager)this.BindingContext[dataSet11.Tables["Clients"]];
5     if (CM.Position < CM.Count-1)
6         CM.Position+=1;
7     else MessageBox.Show("Le dernier enregistrement est atteint");
8 }
```

Commentaires:

Le code de la ligne 3 permet de récupérer une référence sur l'objet CurrencyManager grâce à l'objet BindingContext de notre formulaire.

Le code de la ligne 4 nous permet de tester nous ne sommes pas déjà arrivé sur le dernier enregistrement. Dans ce cas, un message en averti l'utilisateur et dans le cas contraire, nous nous déplaçons sur l'enregistrement suivant.

La propriété CM.position permet de nous déplacer dans la table tandis que la propriété CM.Count nous donne le nombre total d'enregistrements. La position qu'occupe un enregistrement avec la propriété Position est basée zéro.

Exercice 2: modifier le contenu d'un enregistrement.

Maintenant que nous avons pu visualiser le contenu de notre DataSet et nous y déplacer, nous allons envisager la mise en place d'un bouton permettant de valider l'enregistrement des modifications apportées dans la DataSet vers la base de données.

Nous pouvons remarquer que le fait de se déplacer dans le DataSet force au niveau du code sa mise à jour lorsque nous modifions une des zones de texte liée à un champ de notre table.

Nous modifions le code de sorte d'interdire cette mise à jour par le simple déplacement et de ne l'appliquer que si l'on clique sur le bouton enregistrer. Pour les différents boutons permettant le déplacement dans les enregistrements, nous ajouterons la ligne de code suivante:

```
CM.CancelCurrentEdit();
```

Pour la méthode liée à l'événement de cliquer sur le bouton d'enregistrement, nous retrouverons le code suivant:

```
1     private void Record_Click(object sender, System.EventArgs e)
2     {
3         CurrencyManager CM =
4         (CurrencyManager)this.BindingContext[dataSet11.Tables["Clients"]];
5         CM.EndCurrentEdit();
6         this.oleDbDataAdapter1.Update(this.dataSet11,"Clients");
7     }
```

Commentaires:

Le code de la ligne 4 permet de forcer la mise à jour de l'objet DataSet en terminant la phase d'édition. Une fois cette mise à jour effectuée en mémoire, nous pouvons forcer la mise à jour de la base de données à la ligne 5.

Maintenant que les mises à jour sont fonctionnelles, il ne reste plus qu'à envisager l'insertion d'un nouvel étudiant dans la base de données. Cette insertion s'effectue de la même façon: on insère premier dans l'objet DataSet et ensuite on force la mise à jour de la base de données.

Exercice 3: insertion d'un nouvel enregistrement.

Nous partons du principe que l'ajout d'un nouvel enregistrement se fera par l'intermédiaire d'un second formulaire comprenant les champs à compléter ainsi qu'un bouton d'abandon ou de validation.

```
1     private void button4_Click(object sender, System.EventArgs e)
2     {
3         Ajout test = new Ajout();
4         CurrencyManager CM =
5         (CurrencyManager)this.BindingContext[dataSet21.Tables["Clients"]];
6         CM.CancelCurrentEdit();
7         if (test.ShowDialog()==System.Windows.Forms.DialogResult.OK)
8         {
9             System.Data.DataRow tmp =
10            this.dataSet21.Tables["Clients"].NewRow();
11            tmp["Nom"]=test.textNom.Text;
12            tmp["Prenom"]=test.textPrenom.Text;
13            this.dataSet21.Tables["Clients"].Rows.Add(tmp);
14            CM.Position=this.dataSet21.Tables["Clients"].Rows.Count-1;
15            this.oleDbDataAdapter1.Update(this.dataSet21.Tables["Clients"]);
16        }
```

Commentaires :

Le code de la ligne 9 nous permet de créer une nouvelle ligne possédant le même schéma que la table, c-à-d possédant les mêmes champs. A ce stade, la ligne n'est pas encore ajoutée dans la table.

Le code des lignes 10 et 11 permet d'initialiser les champs *Nom* et *Prenom* de la ligne créée à la ligne 9 en utilisant le contenu des deux zones de texte du formulaire.

Le code de la ligne 12 permet d'ajouter la nouvelle ligne ainsi créée et initialisée dans notre table.

Exercice 4 : utilisation d'une ressource de type DataGridView

On peut envisager dans notre exercice de gestion du cercle des étudiants de pouvoir inclure l'ensemble des différentes 'ardoises' pour chacun des clients du bar. Pour ce, nous allons ajouter une table supplémentaire dans notre base de données que nous appellerons *Dettes*. Nous incluons dans cette table les champs suivants :

Reference	de type numérique servant de lien avec la table Clients
Date	de type Date/Heure
Montant	de type numérique
Ident	de type numérique auto incrémenté servant d'index

Une fois cette table créée, nous pouvons ajouter un contrôle de type *DataAdapter* et un *DataSet* dans lequel nous placerons les deux tables. Nous allons aborder l'exercice sous deux aspects : dans une première étape l'utilisation de paramètres liés à la requête SQL de sélection d'enregistrements et dans une deuxième étape la mise en relation de tables. Nous ajouterons un contrôle supplémentaire dans notre formulaire qui sera un *datagrid*.

Exercice 4 a)

Après avoir créé notre nouvel objet *OleDbDataAdapter*, nous pouvons accéder à ses propriétés et étendre dans l'arborescence l'entrée *selectcommand*. Nous retrouvons les propriétés *Parameters* et *CommandText*. La propriété *CommandText* nous permet de retrouver la requête SQL associée à la commande de sélection des enregistrements dans notre table *Dettes*. La propriété *Parameters* nous permet d'inclure des critères de sélections dans notre requête SQL.

Nous éditerons la requête SQL dans laquelle nous ajouterons un critère pour le champ référence en y plaçant la syntaxe suivante : = ?. Nous retrouvons alors la requête SQL suivante :

```
SELECT [Date], Ident, Montant, Reference
FROM Dettes
WHERE (Reference = ?)
```

Le caractère que l'on retrouve dans notre requête SQL doit correspondre à un des paramètres possédant le même nom.

Nous choisissons la propriété *parameters* qui correspond à une collection dans laquelle nous ajouterons *Reference* comme nom d'objet. Nous devons ajouter dans le constructeur du formulaire le code qui permettra d'initialiser ce paramètre permettant de limiter la sélection des enregistrements dans la table *Dettes* à ceux correspondant au client affiché.

```
1      this.oleDbDataAdapter1.Fill(this.dataSet21.Tables["Clients"]);
```

```

2      this.oleDbDataAdapter2.SelectCommand.Parameters["Reference"].Value=this.dataSet21.
      Tables["Clients"].Rows[0]["Reference"].ToString();
3      this.oleDbDataAdapter2.Fill(this.dataSet21.Tables["Dettes"]);
4      this.dataGrid1.DataSource=this.dataSet21.Tables["Dettes"];

```

Commentaires:

Le code de la ligne 2 permet d'initialiser le paramètre appelé *Reference* avec le contenu du champ *Reference* de la première ligne de la table *Clients*.

Le code des lignes 1 et 3 permet de remplir le dataset comprenant les deux tables *Dettes* et *Clients*.

Le code de la ligne 4 permet de définir la source de données devant être utilisée pour remplir le contrôle datagrid.

Lorsque nous passons dans notre formulaire au client suivant, nous devons rafraîchir le contenu du contrôle datagrid pour qu'il corresponde aux dettes du client visualisé. Voici le code inséré dans la méthode permettant de passer au client précédent et mettant le contenu du contrôle datagrid à jour.

```

1      if (CM.Position > 0)
2      {
3      CM.Position--;
4      this.oleDbDataAdapter2.SelectCommand.Parameters["Reference"].Value=
      this.dataSet21.Tables["Clients"].Rows[CM.Position]["Reference"].ToString();
5      this.dataSet21.Tables["Dettes"].Clear();
6      this.oleDbDataAdapter2.Fill(this.dataSet21.Tables["Dettes"]);
7      }

```

Commentaires :

Le code de la ligne 4 permet de modifier la valeur associée au paramètre nommé *Reference*.

Le code de la ligne 5 permet de purger le contenu de la copie de la table *Dettes* présente dans le dataset.

Le code de la ligne 6 permet de remplir le dataset sur base de la modification du paramètre affectant la requête SQL de sélection des enregistrements.

Exercice 4 b)

Nous pouvons définir des relations entre les différentes copies de tables présentes dans un contrôle DataSet. Voici le code permettant dans le constructeur du formulaire de définir cette relation :

```

1      this.dataSet21.Relations.Add("tmp",this.dataSet21.Tables["Clients"].Columns["Reference"],
      this.dataSet21.Tables["Dettes"].Columns["Reference"],false);
2      System.Data.DataView view = new DataView(this.dataSet21.Tables["Clients"]);
3      DataRowView drv = view[0];
4      DataView detailsView = drv.CreateChildView("tmp");
5      this.dataGrid1.DataSource = detailsView ;

```

Commentaires :

Le code de la ligne 1 nous permet de définir une relation entre une table parent et une table enfant en définissant les champs sur lesquels portent cette relation. Nous donnons à cette relation faisant partie d'une collection, un nom : "tmp".

La table parent et son champ: `this.dataSet21.Tables["Clients"].Columns["Reference"]`

La table enfant et son champ: `this.dataSet21.Tables["Dettes"].Columns["Reference"]`.
 Le code de la ligne 3 permet de renseigner la ligne de la table parent devant être utilisée pour assurer la construction de la vue enfant se limitant à comprendre les enregistrements qui correspondent à la ligne de la table parent renseignée à la ligne 3.

Nous retrouverons le même code dans les différentes méthodes nous permettant de nous déplacer dans les différentes "fiches" client.

```
if (CM.Position > 0)
{
    CM.Position--;
    System.Data.DataView view = new DataView(this.dataSet21.Tables["Clients"]);
    DataRowView drv = view[CM.Position];
    DataView detailsView = drv.CreateChildView("tmp");
    this.dataGrid1.DataSource = detailsView;
}
```

La relation peut être créée au moyen de l'éditeur de schéma XML. Dans le menu contextuel lié à notre objet DataSet, nous pouvons choisir l'option 'afficher le schéma' et obtenir une représentation graphique de nos deux tables. Nous pouvons alors choisir dans la boîte d'outils une relation à ajouter à notre table Clients dont nous pouvons paramétrer les différentes propriétés. Dans la liste de l'élément parent, nous choisirons la table Clients tandis que dans la liste de l'élément enfant, nous choisirons la table Dettes. Les deux champs mis en relation sont 'Reference'.

4.4. Lire et écrire des données XML dans un DataSet.

Nous pouvons utiliser la méthode ReadXml de l'objet DataSet pour charger des données à partir d'un fichier XML dans un DataSet. Quand nous utilisons cette méthode, nous pouvons charger des données à partir de fichiers XML qui contiennent seulement des données ou qui contiennent des données ainsi qu'un schéma en ligne apparaissant au début du fichier. Ce schéma décrit les informations XML qui apparaissent après le schéma dans le fichier XML. La table suivante reprend les différentes valeurs pouvant être choisies pour le second paramètre de la fonction XmlReadMode.

Valeur XmlReadMode	Description
ReadSchema	Lit le schéma inclus et le charge ainsi que les données. Si le DataSet contient déjà un schéma, toute nouvelle table qui est définie sera ajoutée. Si le schéma définit une table déjà présente dans le DataSet, une exception sera générée Si le DataSet ne contient pas de schéma et qu'il n'y a pas de schéma inclus, aucune donnée ne sera lue.
IgnoreSchema	Ignore le schéma inclus et charge les données dans le DataSet existant. Toute donnée ne correspondant pas au schéma existant sera rejetée.
InferSchema	Ignore tout schéma inclus et déduit un nouveau schéma basé sur la structure des données XML. Si le DataSet définit déjà un schéma, les tables sont ajoutées.
DiffGram	Lit un DiffGram et ajoute les données au schéma courant dans le DataSet. Un DiffGram est un format XML qui est utilisé pour

	identifier les versions courantes et originales des éléments de données.
Fragment	Lit des fragments XML et ajoute les données aux tables appropriées du DataSet. Ce paramètre est typiquement utilisé pour lire des données XML générées directement par un serveur SQL.
Auto	Examine le fichier XML et choisit l'option la plus appropriée. Si le DataSet contient un schéma ou que le DataSet contient un schéma inclus, la méthode ReadSchema est utilisée. Si le DataSet ne contient pas un schéma et que le fichier XML ne contient pas de schéma inclus, la méthode InferSchema sera utilisée.

Pour mettre en évidence l'utilisation de ces deux méthodes, nous allons dans notre exercice précédent ajouter un bouton permettant d'écrire le contenu du DataSet dans un fichier sur le disque dur. Voici le code:

```
private void button5_Click(object sender, System.EventArgs e)
{
    this.dataSet21.WriteXml("c:\\test.xml", System.Data.XmlWriteMode.WriteSchema);
}
```

Nous pouvons maintenant recréer un nouveau projet dont l'objet DataSet que nous y ajouterons sera initialisé sur base du contenu de notre fichier 'c:\\test.xml'.

5. Programmation multi-threading.

5.1. Introduction.

Lorsque l'on développe une application pour un ordinateur équipé de un ou plusieurs processeurs, nous désirons que notre application présente une réponse la plus rapide possible avec les interactions de l'utilisateur et ce même si l'application est occupée à exécuter un autre travail. L'utilisation de threads multiples est un moyen puissant de garder une application répondant rapidement à l'utilisateur et en même temps entre et même durant des événements. Les systèmes d'exploitation utilisent les processus pour séparer les différentes applications qui sont en cours d'exécution. Le thread est l'unité de base à laquelle un système d'exploitation alloue du temps processeur et l'on peut retrouver plus d'un thread pouvant exécuter du code dans un processus.

Si une application ne comprend qu'un seul thread d'exécution, nous pouvons malgré tout combiner les techniques de programmation asynchrone telles que .NET remoting et XML Web Services pour utiliser la puissance de calcul d'autres ordinateurs en plus du notre pour augmenter le temps de réponse de notre application. La technique de programmation asynchrone sera abordée dans un paragraphe ultérieur.

Un des avantages de la programmation multi-threading est que sans modification de notre code, nous trouverons un avantage certain à faire exécuter notre application sur un ordinateur équipé de plusieurs processeurs du fait que chacun des threads pourra être pris en charge par un processeur donné. La programmation multi-threading présentera quelques désavantages:

La commutation d'un thread à l'autre oblige le système d'exploitation à mémoriser le contexte du microprocesseur lié à chacun des threads (registres du microprocesseur, la pile...). Le nombre maximum de threads sera donc limité par la mémoire disponible.

Lorsque l'on considère le partage des ressources du microprocesseur entre l'ensemble des différents threads, plus il y en aura de threads exécutés simultanément, moins de temps le microprocesseur pourra consacrer à chacun d'eux.

Contrôler correctement le code d'exécution avec beaucoup de threads est complexe et peut être source de beaucoup de bugs.

Habituellement, les threads d'un même processus doivent s'échanger des données via des zones mémoire communes et utiliser des ressources communes. Nous devons également synchroniser les threads dans le sens qu'un thread devra attendre le résultat fourni par un autre thread pour poursuivre son exécution ou attendre qu'un thread ait terminé d'accéder à une zone mémoire commune avant qu'un autre thread n'y accède.

Réduire le nombre de threads dans un processus rend plus facile la synchronisation des ressources. Les ressources qui nécessitent une synchronisation comprennent:

- Les ressources système tel que un port de communication par exemple,
- Les ressources partagées par des processus multiples tels que les fichiers

5.2. Création d'un thread.

Créer une nouvelle instance de l'objet Thread va créer un nouveau thread managé. Nous retrouverons comme unique paramètre dans le constructeur un délégué ThreadStart qui permet de renseigner la méthode qui sera appelée par le nouveau thread lorsque la méthode Thread.Start sera appelée. Appeler cette méthode plus d'une fois pour le même objet provoquerait la levée d'une exception ThreadStateException.

La méthode Thread.Start envoie une requête asynchrone vers le système et retourne immédiatement même éventuellement avant que le nouveau thread ne soit encore démarré. Les méthodes Thread.ThreadState et Thread.IsAlive peuvent être utilisées pour déterminer l'état du thread à n'importe quel moment. Thread.Abort permettra d'interrompre le thread et de la placer en tant que ressource dans le ramasse miettes.

Exemple: soit un projet comprenant dans le formulaire une zone d'édition, un bouton permettant de démarrer un thread, un bouton permettant d'arrêter le thread ainsi qu'une deuxième zone d'édition indiquant l'état du thread. La première zone d'édition comprendra un entier dont la valeur s'incrémentera dans le code du thread lorsque celui-ci sera démarré.

Soit la méthode compteur ajoutée dans notre classe dont voici le contenu:

```
1 public void Compteur()
2 {
3     this.textBox2.Text="Thread exécuté";
4     try
5     {
6         while (true)
7         {
8             compte++;
9             if (compte==100) compte=0;
10            this.textBox1.Text=compte.ToString();
11            Thread.Sleep(1000);
12        }
13    }
14    catch(ThreadAbortException e)
15    {
```

```

16         this.textBox2.Text="Thread arrêté";
17     }
18 }

```

Commentaires:

Cette méthode sera liée au thread que l'on désire exécuter. Elle comprend une boucle sans fin intégrant l'incréméntation de la variable membre `i` ainsi qu'une mise en attente du thread. Lorsque l'on envoie vers le thread une requête d'arrêt via la méthode `Thread.Abort()`, une exception `ThreadAbortException` sera levée, celle-ci étant interceptée par le bloc catch de notre méthode permettant alors d'indiquer au travers de la deuxième zone d'édition que le thread est arrêté.

Voici le code des différentes méthodes associées au bouton de démarrage et d'arrêt du thread.

```

1  private Thread InstanceCaller;
2  public int compte;
3
4  private void Start_Click(object sender, System.EventArgs e)
5  {
6      InstanceCaller = new Thread(new ThreadStart(this.Compteur));
7      InstanceCaller.Name="Compteur";
8      compte=0;
9      this.textBox2.Text="Demande de demarrage";
10     InstanceCaller.Start();
11 }

```

```

1  private void Stop_Click(object sender, System.EventArgs e)
2  {
3      if (this.InstanceCaller==null) return;
4      if (this.InstanceCaller.IsAlive)
5      {
6          this.textBox2.Text="demande d'arrêt";
7          this.InstanceCaller.Abort();
8      }
9  }

```

Commentaires:

Le code de la ligne 6 nous permet d'instancier l'objet `Thread`. Nous retrouvons comme paramètre du constructeur de cet objet un délégué de type `ThreadStart` qui correspondra à notre méthode `Compteur`.

Pour démarrer le thread, nous devons appeler la méthode `Start` sous la forme `InstanceCaller.Start();`

Si nous souhaitons arrêter notre thread, nous allons tester si l'instanciation de l'objet thread a eu lieu et si le thread est en vie. Dans ce cas précis, nous pourrions envisager de l'arrêter en utilisant la méthode `Abort()` sous la forme `InstanceCaller.Abort();`

5.3. Passage de paramètres à un thread.

Nous avons pu constater lors du paragraphe précédent qu'il n'était pas possible d'envoyer des paramètres à la méthode appelée lors de la demande d'exécution du thread. Une des solutions est de placer la méthode dans une classe dont l'instanciation nous permettra de faire passer des paramètres par le constructeur et ensuite de pouvoir renseigner soit une méthode statique liée à la classe ou une méthode instanciée liée à l'objet.

Exemple: Soit un exemple identique au précédent mais pour lequel on désire une sortie automatique du thread lorsque l'on atteint une valeur maximale définie par le thread principale.

Contenu de la classe correspondant au thread secondaire

```
1  public class ClassCompteur
2  {
3      public int ComptMax;
4      public int Compt=0;
5      public ClassCompteur(int ComptMax)
6      {
7          this.ComptMax=ComptMax;
8      }
9
10     public void Compteur()
11     {
12         while (Compt<ComptMax)
13         {
14             Compt++;
15             Thread.Sleep(1000);
16         }
17     }
18 }
```

Contenu du thread principal:

```
1  public Form1()
2  {
3      InitializeComponent();
4      ComptInstance = new ClassCompteur(20);
5  }
6  private void button1_Click(object sender, System.EventArgs e)
7  {
8      MyThread=new Thread(new
9      ThreadStart(ComptInstance.Compteur));
10     MyThread.Start();
11 }
12 private void timer1_Tick(object sender, System.EventArgs e)
13 {
14     if (ComptInstance != null)
15         this.textBox1.Text=ComptInstance.Compt.ToString();
16 }
17 private void button2_Click(object sender, System.EventArgs e)
18 {
19     MyThread.Abort();
20 }
```

Commentaires:

Pour le thread principal:

Le code de la ligne 4 permet d'instancier la classe ClassCompteur en faisant passer un paramètre au constructeur correspondant à la valeur maximale jusque laquelle le compteur du thread secondaire doit compter.

Le code des lignes 8 et 9 permet de créer un thread en renseignant la méthode instanciée ComptInstance.Compteur comme méthode liée au thread.

Pour le thread secondaire:

Lorsque `Compt<ComptMax` renvoie une valeur fausse, nous quittons la boucle while et en même temps la méthode Compteur, cette fin de méthode correspondant à la fin du thread.

5.4. Récupérer les données d'un thread par une méthode.

Nous voudrions que le thread secondaire puisse automatiquement exécuter une méthode du thread principale lorsqu'il aura terminé son travail demandé. Pour ce, nous utiliserons la technique des délégués tels que vu dans le cours de C# en faisant passer la signature de la fonction qui doit être appelée comme paramètre du constructeur de la classe associée au thread secondaire. Cette technique permet d'éviter au thread principal d'attendre la fin d'un thread secondaire pour savoir lorsque un traitement donné est terminé.

Exemple: nous allons reprendre l'exemple précédent et ajouter une zone d'édition dans laquelle la méthode appelée lors de la fin du thread affichera le contenu du compteur envoyé en paramètre.

Contenu du thread secondaire:

```
1  public class ClassCompteur
2  {
3      public int ComptMax;
4      public int Compt=0;
5      private CompteurCallback callback;
6      public ClassCompteur(int ComptMax, CompteurCallback MainCallBack)
7      {
8          this.callback=MainCallBack;
9          this.ComptMax=ComptMax;
10     }
11     public void Compteur()
12     {
13         try
14         {
15             while (Compt<ComptMax)
16             {
17                 Compt++;
18                 Thread.Sleep(1000);
19             }
20         }
21         finally
22         {
23             callback(Compt);
24         }
25     }
26 }
```

Contenu du thread principal:

```
1  public delegate void CompteurCallback(int lineCount);
2
3  public class Form1 : System.Windows.Forms.Form
4  {
5      public Form1()
6      {
7          InitializeComponent();
8          ComptInstance = new ClassCompteur(20 , new
9              CompteurCallback(ResultCompteur));
10     }
11     public void ResultCompteur(int lineCount)
12     {
13         this.textBox2.Text=lineCount.ToString();
14     }
15 }
```

Commentaires:

Pour le thread principal:

Nous retrouvons à la ligne 1 le code permettant de déclarer notre délégué: celui-ci doit correspondre à une méthode ne devant rien retourner et devant recevoir comme paramètre un entier.

Le code de la ligne 8 comprend l'instanciation de la classe ClassCompteur avec comme passage de paramètre l'instanciation de notre délégué comprenant lui-même le nom de la fonction devant être appelée comme paramètre.

Pour le thread secondaire:

Nous utilisons dans la méthode Compteur un bloc try-finally intégrant l'appel à la méthode passée en paramètre dans le constructeur. Etant donné que cet appel s'effectue dans le bloc finally, la méthode sera appelée que le thread se termine normalement ou de façon forcée.

6. Programmation asynchrone.

Le .NET Framework autorise l'appel à n'importe quelle méthode de façon asynchrone. Pour ce, il suffit de créer un délégué avec la même signature que la méthode que l'on désire appeler; le CLR définit automatiquement les méthodes BeginInvoke et EndInvoke pour ce délégué.

La méthode BeginInvoke est utilisée pour initialiser l'appel asynchrone. Elle a les mêmes paramètres que la méthode que l'on désire exécuter de façon asynchrone avec en plus deux paramètres supplémentaires qui seront décrits plus tard. BeginInvoke retourne immédiatement et n'attend pas que l'appel soit terminé. BeginInvoke retourne un objet de type IAsyncResult permettant de suivre l'évolution de l'appel.

La méthode EndInvoke est utilisée pour récupérer les résultats d'un appel asynchrone. Cette méthode peut être appelée à n'importe quel moment après l'appel à BeginInvoke; si la méthode asynchrone n'est pas terminée, EndInvoke provoquera le blocage jusqu'à ce que l'appel soit terminé. Les paramètres de cette méthode peuvent inclure des paramètres out et ref de la méthode que l'on aura exécuté de façon asynchrone et en plus un objet IAsyncResult. Après avoir appelé la méthode BeginInvoke, nous pouvons effectuer les actions suivantes:

- Faire un autre travail et ensuite appeler EndInvoke en attente de la fin de l'exécution de la méthode.
- Obtenir un objet WaitHandle en utilisant IAsyncResult.AsyncWaitHandle et utiliser sa méthode WaitOne pour bloquer l'exécution jusqu'à ce que WaitHandle soit signalé et seulement alors appeler EndInvoke
- Passer un délégué comme paramètre de la méthode BeginInvoke. Cette méthode sera exécutée sur un thread quand l'exécution de la méthode asynchrone sera arrivée à terme et alors appeler EndInvoke.
- Scruter l'objet IAsyncResult retourné par la méthode BeginInvoke pour déterminer lorsque la méthode asynchrone est terminée et appeler alors EndInvoke.

Exemple 1: Nous allons envisager un exercice similaire à celui envisagé dans le paragraphe des thread à savoir le comptage jusqu'à une valeur maximale passée en paramètre. Nous aborderons plusieurs variantes en fonction de la technique choisie pour être informé de la fin de l'exécution de la méthode asynchrone.

1a) Appel de la méthode EndInvoke.

Nous retrouvons un bouton dont le clic provoquera l'exécution de la méthode Compteur de façon asynchrone. Le décomptage apparaîtra dans la première zone d'édition tandis que la deuxième zone d'édition contiendra l'état d'exécution de la méthode.

1	<code>public delegate void DeleCompt(int MaxVal);</code>
2	
3	<code>public class Form1 : System.Windows.Forms.Form</code>
4	<code>{</code>
5	<code>public Form1()</code>
6	<code>{</code>
7	<code>InitializeComponent();</code>
8	<code>AsyncCompteur = new DeleCompt(this.Compteur);</code>
9	<code>}</code>
10	
11	<code>public void Compteur(int MaxVal)</code>
12	<code>{</code>
13	<code>int i=0;</code>
14	<code>while (i<MaxVal)</code>
15	<code>{</code>
16	<code>i++;</code>
17	<code>Thread.Sleep(1000);</code>
18	<code>this.textBox1.Text=i.ToString();</code>
19	<code>this.textBox1.Refresh();</code>
20	<code>}</code>
21	<code>}</code>
22	<code>Private void Start_Click(object sender, EventArgs e)</code>
23	<code>{</code>
24	<code>System.IAsyncResult result=AsyncCompteur.BeginInvoke(20,null,null);</code>
25	<code>this.textBox2.Text="en attente de la fin";</code>
26	<code>this.textBox2.Refresh();</code>
27	<code>AsyncCompteur.EndInvoke(result);</code>
28	<code>this.textBox2.Text="Fonction asynchrone terminée";</code>
29	<code>}</code>
30	<code>}</code>

Commentaires:

Lorsque l'on clique sur le bouton, un événement est levé et la méthode Start_Click est exécutée. Cette méthode comprend l'appel asynchrone de la méthode Compteur par l'intermédiaire d'un délégué qui se trouve déclaré dans le code de la ligne 1 et dont l'instanciation s'effectue dans le code de la ligne 8.

Nous voyons apparaître le message "en attente de fin " dans la deuxième zone de texte et ensuite l'application apparaît comme figée, la preuve que nous avons un effet de blocage provoqué par la méthode EndInvoke en attente de la fin de l'exécution de la méthode asynchrone (ligne 27).

Lorsque la méthode asynchrone se termine, nous voyons apparaître le texte "Fonction asynchrone terminée" dans la deuxième zone de texte.

1b) utilisation d'une méthode de CallBack.

Il est évident que dans le premier exemple l'intérêt d'un fonctionnement asynchrone perd son sens puisque la méthode EndInvoke est appelée immédiatement et que le thread appelant reste bloqué dans l'attente d'une fin d'exécution bien que nous aurions pu placer du code à exécuter entre l'appel des deux méthodes. Nous pouvons imaginer que le thread appelant est à un moment donné besoin des résultats fournis par la méthode asynchrone.

```

1     public void Compteur(int MaxVal)
2     {
3         int i=0;
4         while (i<MaxVal)
5         {
6             i++;
7             Thread.Sleep(1000);
8             this.textBox1.Text=i.ToString();
9             this.textBox1.Refresh();
10        }
11    }
12    private void button1_Click(object sender, System.EventArgs e)
13    {
14        System.IAsyncResult result=AsyncCompteur.BeginInvoke(20,new
15        AsyncCallback(FinComptage),null);
16        this.textBox2.Text="en attente de la fin";
17    }
18    public void FinComptage(System.IAsyncResult result)
19    {
20        AsyncCompteur.EndInvoke(result);
21        this.textBox2.Text="Fonction asynchrone terminée";
22    }

```

Commentaires:

Nous retrouvons à la ligne 20 le code permettant de démarrer notre méthode asynchrone avec en passage de paramètre pour BeginInvoke, la méthode devant être appelée lorsque l'exécution de la méthode asynchrone se termine.

A la différence de l'exercice précédent, malgré le message "en attente de la fin", notre thread principal ne reste pas bloqué et ce n'est que lorsque la méthode asynchrone se termine que notre méthode de callback est appelée et que l'on voit alors apparaître le message "Fonction asynchrone terminée".

1c) Utilisation de la méthode WaitOne.

La différence entre la méthode EndInvoke et WaitOne est minime. Dans le premier cas, il est impératif que la méthode asynchrone termine son exécution pour que l'on puisse être débloqué tandis que dans le second cas, nous pouvons renseigner un délai de dépassement (Timeout) en paramètre. Cette dernière méthode ne nous dispense pas d'appeler la méthode EndInvoke pour que nous puissions récupérer des données en provenance de la méthode asynchrone.

7. Utilisation de services Web.

7.1. Introduction.

Les services Web sont accessibles à partir de n'importe quelle sorte d'applications pouvant comprendre les applications Windows, les applications Web, d'autres services Web mais aussi les applications en mode console. Un service Web est un ensemble de fonctions qui sont exposées sur le réseau au travers d' adresses URL, l'ensemble des données échangées étant transmis par le biais de messages XML. Ces services peuvent être accédées de façon interne par une simple application ou être exposées de façon externe à travers Internet pour être utilisés par n'importe quelle autre application en utilisant un 'service broker' ce dernier étant

un nœud qui héberge un registre UDDI (Universal Description, Discovery and Integration). Cette technologie est donc basée sur HTTP, XML et SOAP, SOAP (Single Object Access Protocol) étant un protocole léger, basé XML, pour l'échange d'informations dans un environnement décentralisé et distribué.

Dans une implémentation du modèle de services XML, nous retrouvons les éléments de base suivants: le broker, le provider qui est le nœud sur le réseau hébergeant le service Web et le client.

WSDL est le langage de description de services. Un document WSDL définira les messages XML qui seront acceptés ou générés par le service; on y retrouvera l'élément racine appelé 'definitions' contenant cinq éléments enfants primaires décrits dans le tableau ci après:

Eléments	Description
types	Définit les types de données utilisés pour échanger les messages
message	Décrit les messages devant être communiqués
portType	Identifie un ensemble d'opérations et les messages invoqués pour chacune de ces opérations.
binding	Spécifie les détails du protocole pour les opérations liées au service et décrit comment lier le contenu abstrait des messages avec un format concret.
service	Regroupe un jeu de ports.

7.2. Comment accéder à un service Web XML.

Lorsque nous avons créé notre projet, nous pouvons générer ce que l'on appelle un proxy: c'est une classe qui encapsule le code permettant de parser les messages qui sont reçus d'un service Web. Cette génération peut s'effectuer au moyen d'un utilitaire fourni avec le Framework SDK qui est WSDL.exe ou d'utiliser une des options du menu projet 'Générer une référence Web'.

Nous prendrons un service Web créé au préalable comprenant les méthodes suivantes:

```
[WebMethod (Description="Ceci est mon premier service")]
public string HelloWorld()
{
    return "Mon premier service web";
}
[WebMethod (Description="Introduction d'un nom")]
public string Introduction(string Nom)
{
    return "Nom introduit:"+Nom;
}
```

Une fois la référence ajoutée à notre projet, nous pouvons instancier la classe proxy et appeler une des méthodes de notre service. Pour ce, nous placerons dans le formulaire de notre projet deux zones d'édition et un bouton: la première zone d'édition permet d'introduire un nom tandis que le bouton permet d'appeler la méthode 'Introduction' de notre service en lui faisant passer le contenu de la première zone d'édition en paramètre. La deuxième zone d'édition permettra l'affichage de la chaîne de caractères renvoyée par cette méthode.

```
private void button1_Click(object sender, System.EventArgs e)
{
    WebService1.Service1 test = new WebService1.Service1();
    this.textBox2.Text=test.Introduction(this.textBox1.Text);
}
```

