Concepteur Développeur en Informatique

Développer des composants d'interface

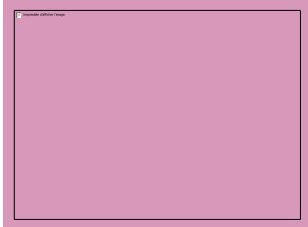
Formulaires Windows – Partie 3

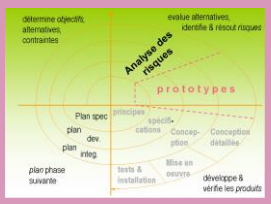
Accueil

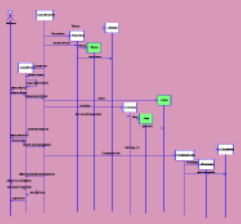
Apprentissage

PAE

Evaluation







Localisation : U03-E03-S02

SOMMAIRE

1 Introduction

Ce document permet de revenir sur quelques principes à prendre en compte pour la mise en place d'application Windows :

- Un approfondissement de la classe **Control** et quelques techniques associées pour automatiser certaines tâches.
- La validation des données acquises via les contrôles et la communication d'informations sur les erreurs à l'utilisateur.

Et de poursuivre la présentation des contrôles avec ceux permettant de gérer des listes d'objets.

2 Approfondissement de la classe Control

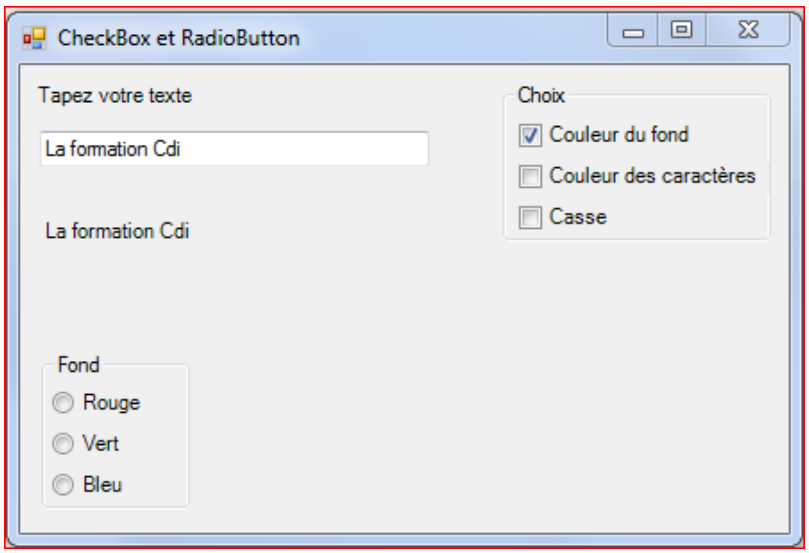
2.1 La propriété Tag.

Cette propriété existe sur tous les contrôles et permet d'associer à chaque contrôle :

- Une référence d'objet lorsque cette propriété est assignée par programme lors de l'exécution (RunTime)
- Une référence de type String en mode conception (DesignTime)

Cette propriété rend de nombreux services et facilité l'automatisation de certaines tâches.
Elle permet notamment de traiter des relations de dépendances entre deux contrôles ou entre un contrôle et un objet.

Je reprends ici un exemple traité lors de la manipulation des groupes de contrôles.



Dans ce contexte, le choix d'un des boutons radio du groupe Fond va permettre d'assigner la couleur définie au niveau de la propriété Tag du bouton radio :

(Name)	**rbBCRouge**

Tag	**Red**

Il ne reste plus qu'à programmer l'opération dans le gestionnaire d'événement.

```
private void rb_CheckedChanged(object sender, EventArgs e)
{
    switch (((RadioButton)sender).Name)
    {
        case "rbBCRouge":
        case "rbBCVert":
        case "rbBCBleu":
            this.lblAffiche.BackColor = Color.FromName(((Control)sender).Tag.ToString());
            break;
```

Nous pourrions aussi envisager d'associer un contrôle image à un objet de type Salarie.

Sur le survol de l'image, nous pourrions alors afficher son nom et son prénom.

L'exemple programmé


```
public partial class FrmTag : Form
{
    Salarie sal = new Salarie("Bost", "Vincent", "96GBC11");
    public FrmTag()
    {
        InitializeComponent();
        this.pbBost.Tag = sal;
    }

    private void pbBost_MouseHover(object sender, EventArgs e)
    {
        Label lbSal = new Label();
        lbSal.AutoSize = true;
        lbSal.Text = string.Format("Son nom {0} et son prénom {1}", sal.Nom, sal.Prenom);
        lbSal.Left = this.pbBost.Right + 10;
        lbSal.Top = this.pbBost.Top + 10;
        this.Controls.Add(lbSal);
    }
}
```

2.2 La recherche d'un contrôle existant

Une méthode qu'il faut absolument connaître pour envisager, conjointement avec la propriété Tag, certains automatismes, la méthode Find().
Celle-ci, implémentée sur la collection Controls des conteneurs, permet de récupérer un tableau des contrôles dont la propriété Name répond à la chaîne passée en argument de la méthode.

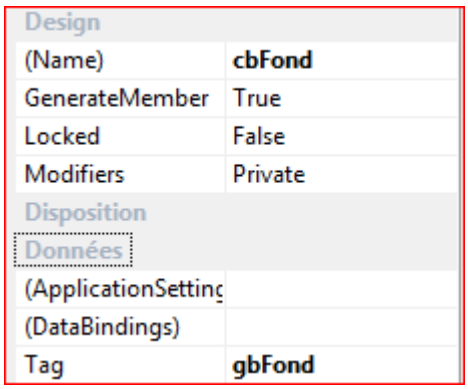
Cette méthode renvoie donc un tableau des références des contrôles.

La méthode utilise un deuxième argument qui précise si la recherche s'effectue parmi les descendants directs (false) ou parmi tous les descendants (true).

L'exemple suivant illustre son usage dans la recherche d'un descendant direct.

Toujours dans l'exemple de gestion des groupes de contrôles, cette méthode me permet de récupérer le contrôle dont le nom est assigné à la propriété Tag d'un contrôle dépendant.

A chaque case à cocher est associé au niveau de la propriété Tag le nom d'un groupe de contrôle qui devra être affiché ou non en fonction de la valeur de la coche.



Ainsi, lorsque l'utilisateur coche ou décoche une case à cocher, le gestionnaire affiche ou masque le groupe de contrôle correspondant en permutant l'état visible.

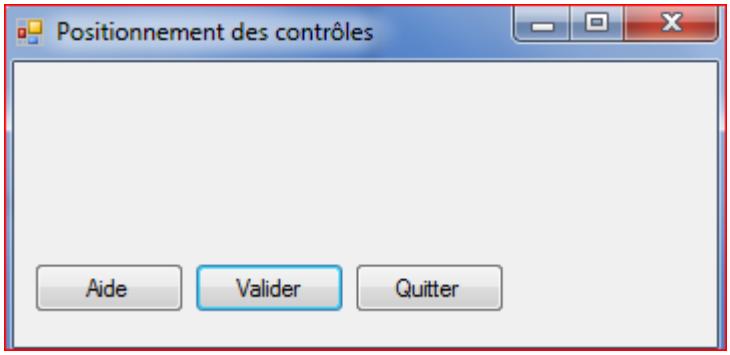
```
private void cb_CheckedChanged(object sender, EventArgs e)
{
    this.Controls.Find(((CheckBox)sender).Tag.ToString(), false)[0].Visible =
        !this.Controls.Find(((CheckBox)sender).Tag.ToString(), false)[0].Visible;
}
```

2.3 Le positionnement des contrôles

Il existe différentes possibilités pour définir la taille et la position des contrôles.
Vous pouvez :

- Tailler et positionner les contrôles en définissant les propriétés Width et Height puis Top, Left, Bottom, Right avec des valeurs absolues.
- Exprimer cette taille et ce positionnement en fonction de valeurs relatives à un autre contrôle.
- Enfin définir les frontières du rectangle occupé par le contrôle.

Le résultat suivant :



Est obtenu en exécutant la méthode ci-dessous :

```
private void AjouterBoutons()
{
    // Valeurs absolues
    Button buttonOK = new Button();
    buttonOK.Location = new Point(90,100);
    buttonOK.Size = new Size(75, 25);
    buttonOK.Text = "&Valider";
    this.AcceptButton = buttonOK;

    // Valeurs relatives
    Button buttonCancel = new Button();
    buttonCancel.Top = buttonOK.Top;
    buttonCancel.Left = buttonOK.Right + 5;
    buttonCancel.Width = buttonOK.Width;
    buttonCancel.Height = buttonOK.Height;
    buttonCancel.Text = "&Quitter";
    this.CancelButton = buttonCancel;

   // Frontières du rectangle : mélange des genres
    Button buttonHelp = new Button();
    buttonHelp.Bounds = new Rectangle(10, buttonCancel.Top, 75, 25);
    buttonHelp.Text = "&Aide";

    this.Controls.AddRange(new Control[] { buttonOK, buttonCancel, buttonHelp });
}
```

3 La validation des données

Il est toujours nécessaire de vérifier la validité des données communiquées par un utilisateur dans un contrôle, avant d'en traiter le contenu.

Cette approche ne vous dispense pas pour autant de gérer les exceptions au sein des classes des entités métier que vous serez amené à concevoir.

Les objectifs ne sont pas les mêmes :

- Au sein de votre type, il s'agit de préserver l'intégrité de vos objets : Ils doivent respecter les règles définies.
- Au niveau de l'interface : recherchez la performance et éviter d'effectuer des opérations inutiles et couteuses comme la levée d'une exception.

La propriété **Text** d'une zone de texte est toujours une chaîne de caractères. Elle peut donc accueillir n'importe quel caractère frappé au clavier, numérique comme alphabétique.

Pour effectuer des calculs arithmétiques elle devra être convertie en un type C# numérique (int, long, float, double …) et pour être convertie sans générer d'erreurs (voir le support de cours Initiation C#), le développeur devra s'assurer que les caractères entrés sont bien des chiffres ou des séparateurs décimaux. Pour cela, il vous faudra vous familiariser avec les méthodes Parse et TryParse

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

Sur quels événements réaliser les contrôles ?

3.1 Les évènements à utiliser

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

Sur le bouton **Valider** du formulaire. Il est alors nécessaire de réaliser l'exécution de l'ensemble des contrôles sur la feuille.

Mais il est aussi possible de les envisager sur :

- L'événement de demande de validation du contrôle
- La pression d'une touche du clavier.

3.1.1 L'événement Validating

Il se produit sur l'évènement de la zone de texte nécessitant un contrôle.

L'événement **Validating** d'un contrôle se produit lorsque le focus quitte ce contrôle et passe à un contrôle dont la propriété **CausesValidation** est true.

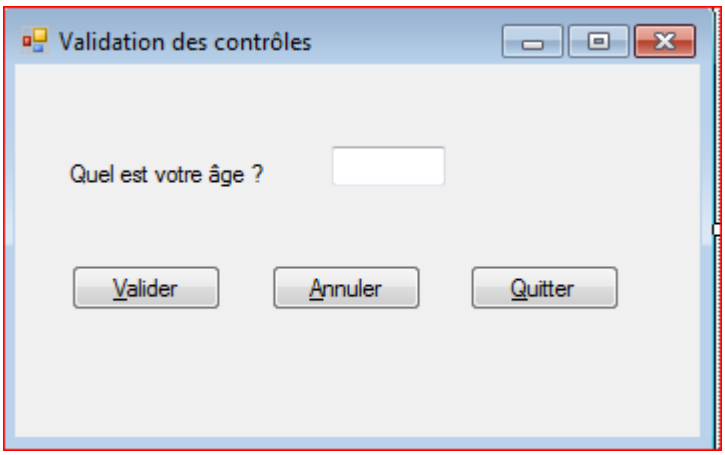
Si les données saisies sont erronées la demande de validation est interrompue en définissant la propriété **Cancel** du paramètre **CancelEventArgs** à **true**. Le contrôle conserve alors le focus et ne permet pas le passage du focus au contrôle suivant.

(Name)	**btnQuitter**
GenerateMember	True
Locked	False
Modifiers	Private
Disposition	
Données	
(ApplicationSettinç	
(DataBindings)	
Tag	
Focus	
CausesValidation	**False**

Pour éviter que l'évènement **Validating** se produise lorsque l'utilisateur souhaite quitter la feuille (bouton Quitter) ou annuler sa saisie (bouton Annuler), **false** sera assigné à la propriété **CausesValidation** de ces deux boutons.

La structure d'évènement **CancelEventArgs** propose une donnée, **Cancel,** qui, positionnée à **true,** annule l'évènement.

Si la demande de validation n'a pas été annulée, l'événement **Validated** se déroule après l'événement **Validating**. Le focus passe alors au contrôle suivant.



Dans l'exemple d'illustration, il est nécessaire de vérifier que l'âge donné soit supérieur à 18.

Tant que la donnée n'est pas valide (ou l'un des boutons Quitter ou Annuler n'ait été choisi) le contrôle txtAge conserve le focus et l'utilisateur ne peut quitter ce dernier.

```
private void txtAge_Validating(object sender, CancelEventArgs e)
{
    int age;
    if (!int.TryParse(txtAge.Text, out age)) e.Cancel = true;
    if (age < 18) e.Cancel = true;
}
```

3.1.2 L'événement KeyPress

Il est aussi possible de faire un contrôle caractère par caractère.

C'est ce type de contrôle qui est mis en œuvre avec les masques d'édition de texte et le contrôle **MaskedTextBox**.

Le contrôle sera exécuté lors de la pression de chaque touche dans la zone de texte dans un gestionnaire d'événement associé à K**eyPress**.

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

L'événement **KeyPress** n'est pas déclenché par les touches qui ne sont pas de type caractère. Ces touches déclenchent uniquement les événements **KeyDown** et **KeyUp**.

Il convient de laisser le droit à **corriger une erreur** de saisie et donc autoriser la touche de correction **BackSpace**.

L'événement **KeyPress** transporte des informations qui sont accessibles via la donnée de type **KeyEVentArgs** :

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

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

}
```

3.2 Informer l'utilisateur

Toute information non valide doit être signalée à l'utilisateur. Sous quelle forme ? Plusieurs voies existent, la dernière étant celle à privilégier :

- Envoi d'une boite de dialogue modal par le biais de la méthode **MessageBox**.
- Positionnement du focus et mise en inversion vidéo du champ en erreur (voir les propriétés spécifiques des **TextBox**)
- Utilisation d'un fournisseur d'erreurs, le contrôle dédié **ErrorProvider**.

Il convient de recourir de manière parcimonieuse aux messages en mode modal qui interrompent les actions de l'utilisateur. En tout cas, cette approche est tout à fait inopportune pour mettre l'accent sur un ensemble d'erreurs. L'utilisateur ne peut se souvenir de plus de tous les messages.

3.2.1 Mise en œuvre du contrôle ErrorProvider

Si plusieurs messages d'erreur risquent d'apparaître à la suite, l'utilisateur devra se souvenir de tous ces messages : une meilleure solution consiste à utiliser le contrôle ErrorProvider. Il se trouve dans le menu des composants qui regroupe l'ensemble des contrôles fournissant un service sans disposer de représentation graphique propre :

Déposez ce contrôle depuis le menu des compoosants sur le formulaire où il doit être mis en œuvre.

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

Amendons l'exemple précédent pour fournir un message à l'utilisateur en cas d'erreur.

```
private void txtAge_Validating(object sender, CancelEventArgs e)
{
    errorProvider1.SetError(txtAge, string.Empty);
    int age;
    if (!int.TryParse(txtAge.Text, out age))
    {
        errorProvider1.SetError(txtAge, "Doit être une valeur numérique entière");
        e.Cancel = true;
        return;
    }
        else  errorProvider1.SetError(txtAge, string.Empty);

    if (age < 18)
    {
        e.Cancel = true;
        errorProvider1.SetError(txtAge, "Age ne peut être inférieur à 18 ans");
    }
    else errorProvider1.SetError(txtAge, string.Empty);
}
```

Vous pouvez utiliser un **seul fournisseur d'erreurs** par formulaire.

La méthode SetError est utilisée pour :

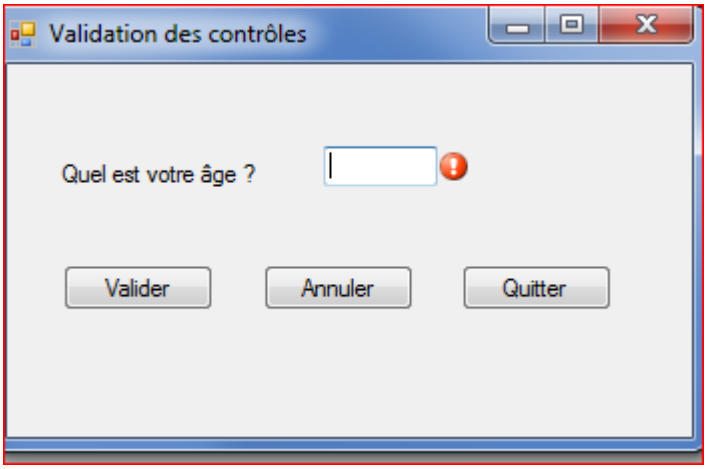
- Positionner le contrôle en erreur, premier argument, et le message de l'erreur, en deuxième argument
- Effacer l'erreur sur le contrôle, en premier argument et une chaine vide en deuxième argument de la méthode.

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

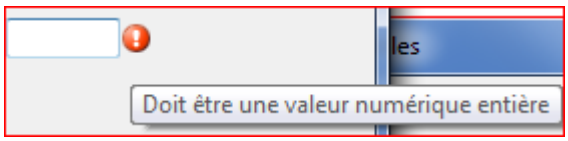
La propriété **BlinkRate** permet de ralentir ou accélérer ce clignotement.
Lorsqu'une autre erreur est détectée, l'icône ne recommence à clignoter que s'il s'agit d'une erreur différente :

En attribuant à la propriété **BlinkStyle**, la valeur **AlwaysBlink**, l'icône clignotera à chaque erreur.

Le résultat en image :



Le message est affiché à l'utilisateur lorsque celui-ci survole l'icône signalant l'erreur.



4 Les boites de liste

Les boites de liste affichent un ensemble d'éléments (**items**). Ces éléments peuvent être le plus souvent du texte mais aussi faire référence à des objets.
L'utilisateur peut sélectionner un ou plusieurs éléments.
Le contrôle affiche automatiquement des barres de défilement si le contenu dépasse le cadre du contrôle.

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

Les boites de liste sont représentées par les contrôles suivants :

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

Il est possible d'ajouter à la fin de la liste ou à une position donnée avec la méthode **Insert**() :

```
{
    // Ajout à la fin;
    lbNoms.Items.Add("Bost");
    // Ajout en position
    lbNoms.Items.Insert(0, "Morillon");
}
```

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

```
// Récupération de la valeur en position 0
String s = (string)lbNoms.Items[0];

// Récupération de l'élément sélectionné
s = (string)lbNoms.SelectedItem;

// Récupération de l'élément sélectionné

 s = (string)lbNoms.Items[lbNoms.SelectedIndex];
```

4.1 Les boites de liste standard (ListBox)

Propriétés	
MultiColumn	Positionnée à true, la ListBox affiche ses éléments en plusieurs colonnes et une scrollBar horizontale apparaît.
ScrollAlwaysVisible	Positionnée à true, une ScrollBar apparaît selon le nombre d'éléments.
SelectionMode	Détermine le nombre d'éléments pouvant être sélectionnés à la fois.
Items	Collection des libellés des éléments de la liste (dont la propriété Count détermine le nombre d'éléments de la liste)
Propriétés Run-Time	
SelectedIndex	Index de l'élément sélectionné (0 pour le premier de la liste, -1 si aucune sélection)
SelectedIndices	Collection des indices des éléments sélectionnés
SelectedItem	Elément sélectionné (Etant de type Object, il peut s'appliquer à n'importe quel type ou classe- string par exemple-mais un casting est toujours nécessaire.)
SelectedItems	Collection des éléments sélectionnés
Sorted	Permet de trier les éléments automatiquement

La propriété **SelectedItem** donne l'élément sélectionné dont l'indice dans la liste est **SelectedIndex**.

En cas de sélection multiple, on utilisera les collections **SelectedItems** et **SelectedIndices**.

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

```
foreach (string item in lbNoms.SelectedItems)
{

}
```

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

4.1.1 Les événements

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

L'évènement **SelectedIndexChanged** est déclenché à chaque nouvelle sélection dans la liste.7

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

4.2 Les boites de liste avec cases à cocher

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

Etant dérivée de **ListBox**, les propriétés et méthodes de **ListBox** sont applicables à la classe **CheckedListBox.**

La classe présente toutefois des propriétés et méthodes propres comme dans cet exemple où un nom est ajouté puis coché.

```
// Ajout à la fin;
clbNoms.Items.Add("Bost");

// Cocher un élément en position
clbNoms.SetItemChecked(0, true);
```

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

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

4.2.1 Gérer les évènements particuliers des zones de liste à cocher

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

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

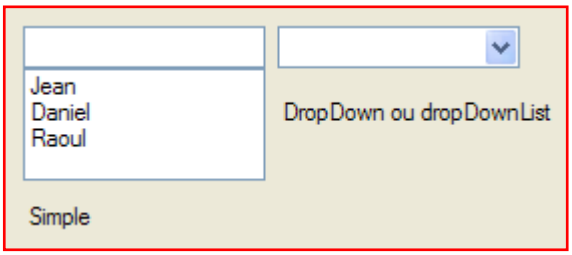
- Index Indice de l'élément qui a généré l'événement
- CurrentValue état courant de la case à cocher (valeur CheckState)
- NewValue nouvel état de la case à cocher (valeur CheckState)

4.3 Les boites de liste déroulantes (comboBox)

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

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

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



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

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

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

Etant dérivée de **ListBox**, les propriétés et méthodes de **ListBox** sont applicables à la classe **ComboBox**.

La classe présente toutefois des propriétés et méthodes propres.

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

4.3.1 Les évènements

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

La classe ComboBox expose des évènements particuliers

Evénements	
DropDown	Affichage de la liste déroulante
SelectionChangeCommitted	Modification d'un élément de la liste déroulante