Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Les Contrôles WPF

Julien DOLLON

Microsoft Student Partners

Sommaire

1 Introduction

Après avoir vu comment disposer nos contrôles dans une fenêtre en WPF et les différents panels qui existent, nous allons voir quels sont les contrôles proposé par WPF, comment les utiliser et ce qu'ils nous permettent de faire ?

Mais pour commencer qu'est-ce qu'un contrôle ?

Un contrôle c'est tout simplement un objet du Framework .NET. Il est donc fournit par le Framework, ce qui va nous permettre de gagner énormément de temps. Vous vous rendrez compte que la plupart de ces contrôles étaient déjà présent avec les WinForms.

En revanche il faut garder à l'esprit qu'un contrôle est avant toute chose un objet, un élément d'une fenêtre qui va permettre à l'utilisateur de cette dernière d'interagir avec la couche métier de notre application.

Ce chapitre est extrêmement important car il en découlera les chapitres sur les Templates, très important en WPF, et la création de nos propres contrôles.

Pour finir, de nombreux développeurs ont tendance à faire de plus en plus abstraction de l'IHM et à se concentrer sur la couche métier. En revanche il faut garder à l'esprit que moins l'IHM sera facile d'utilisation moins les fonctionnalités métier de votre application seront facile d'accès !

C'est pourquoi je vous invite à être attentif et à connaître l'utilité de chacun des contrôles qui vous seront présenté ici de façon à pouvoir varier et choisir le bon contrôle le moment venu.

2 Les contrôles de base

2.1 Gestion du texte en WPF

Nous verrons dans cette partie les contrôles qui permettent la manipulation de texte en WPF.

Vous noterez en revanche que nous ne parlerons pas du label, tout simplement car il n'a pas changé avec WPF.

2.1.1 TextBlock

Un TextBlock ou `System.Windows.Controls.TextBlock` est un contrôle destiné à être léger et par conséquent à afficher de petites quantités de texte.

Un TextBlock peut contenir toute les propriétés de contenu de type Inlines, c'est-à-dire Bold, Italic, Hyperlink etc.

Pour ce première exemple nous verrons comment déclarer un TextBlock en XAML et en C# mais par la suite nous conserverons seulement le XAML.

```
<!--XAML-->
<TextBlock Name="DFTextBlock" Margin="10,10,10,10" TextWrapping="Wrap"
Height="170" Width="215">
      <Bold>Dotnet-France</Bold> formez vous pour passer vos
      <Italic>certifications</Italic>
</TextBlock>
```

Comme vous pouvez le voir rien de sorcier cela reste similaire à ce qu'on a vu précédemment dans le cours sur le layout WPF au niveau de la déclaration en XAML.

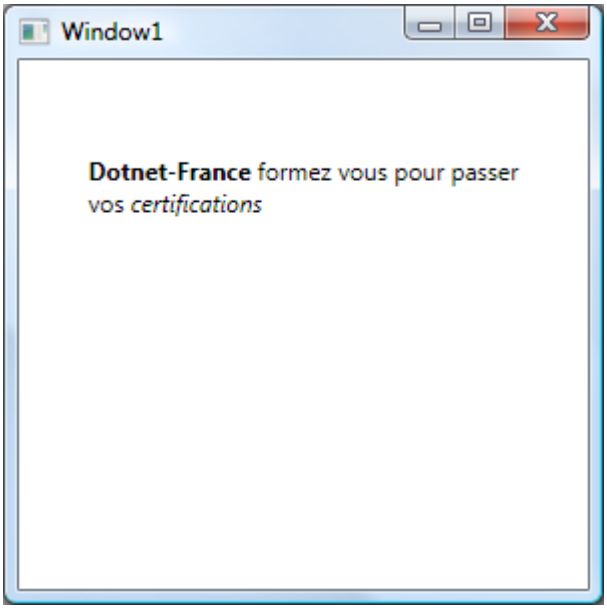
Comme vous pouvez le voir à l'intérieur de notre TextBlock on a utilisé deux propriétés de type Inlines *Bold* et *Italic*.

Pour ce qui est de la déclaration en C# :

```
//C#
TextBlock textBlock1 = new TextBlock();

textBlock1.TextWrapping = TextWrapping.Wrap;
textBlock1.Name = "DFTextBlock";

Thickness myThickness = new Thickness();
myThickness.Bottom = 10;
myThickness.Left = 10;
myThickness.Right = 10;
myThickness.Top = 10;

textBlock1.Margin = myThickness;
textBlock1.Height = 170;
textBlock1.Width = 215;
textBlock1.FontWeight = FontWeights.UltraBold;

textBlock1.Text = "Dotnet-France formez vous pour passer vos
certifications";

this.Content = textBlock1;
```

```
' VB
Dim textBlock1 As New TextBlock()

textBlock1.TextWrapping = TextWrapping.Wrap
textBlock1.Name = "DFTextBlock"

Dim myThickness As New Thickness()
myThickness.Bottom = 10
myThickness.Left = 10
myThickness.Right = 10
myThickness.Top = 10

textBlock1.Margin = myThickness
textBlock1.Height = 170
textBlock1.Width = 215
textBlock1.FontWeight = FontWeights.UltraBold

textBlock1.Text = "Dotnet-France formez vous pour passer vos
certifications"

Me.Content = textBlock1
```

On arrive au résultat suivant :



2.1.2 TextBox

Un TextBox ou `System.Windows.Controls.TextBox` est un contrôle qui offre la possibilité d'entrer du texte simple dans vos applications WPF. Il fonctionne pratiquement de la même manière qu'avec les WinForms, le dépaysement ne sera donc pas total : nous allons juste rajouter une notion d'XAML.

Voici comment on déclare une textbox en XAML :

```
<!--XAML-->
<TextBox Height="23" Name="DFTextBox" MaxLength="12" Width="120">
      Démo
</TextBox>
```

Dans notre exemple vous pouvez voir qu'on peut limiter le nombre de caractères que peut contenir la Textbox, très pratique dans un formulaire ou on a besoin par exemple d'une référence avec un nombre de caractères bien définis.

Comme on le verra tout à l'heure une Textbox requiert moins de ressources qu'une RichTextBox en revanche elle ne permet pas autant de choses. La Textbox permet tout de même d'avoir accès aux fonctionnalités de vérification orthographique en temps réel et à un menu contextuel.

2.1.3 RichTextBox

Une RichTextBox ou `System.Windows.Controls.RichTextBox` est un contrôle d'édition avancé qui propose plus de fonctionnalité que la TextBox.

En effet, comme on a pu le voir, les fonctionnalités proposées par une Texbox restent tout de même limitées. La RichTextBox va posséder les mêmes fonctionnalités de base qu'une Textbox avec en plus la possibilité de mettre en forme notre texte ou encore avoir du contenu tel que des images des tableaux etc., tout cela fait un peu penser aux WYSIWYG.

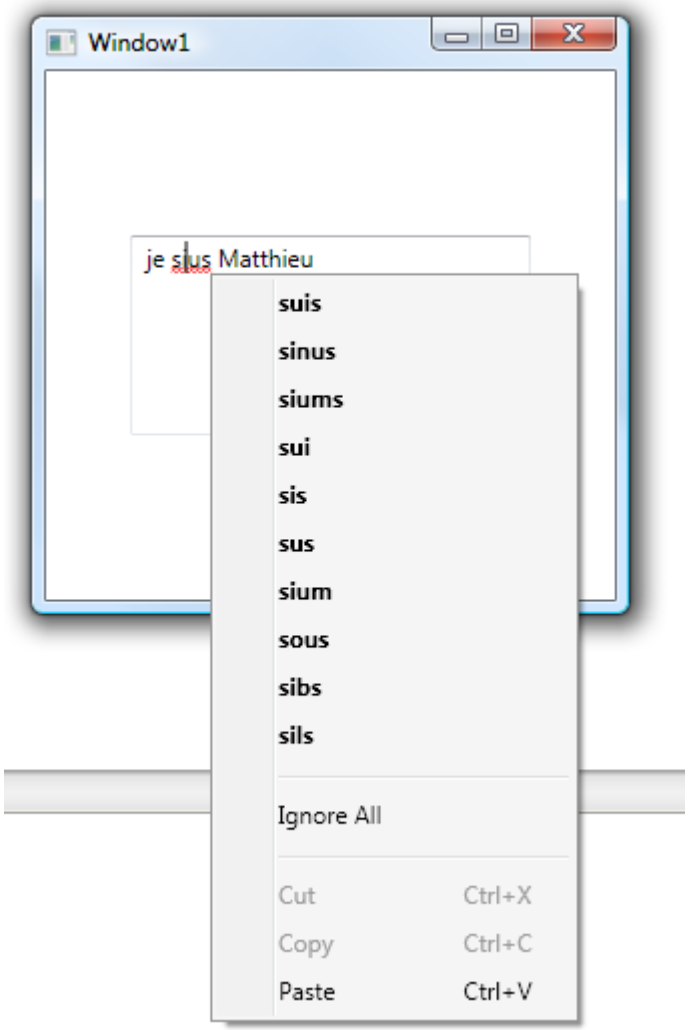
Voici comment déclarer une RichTextBox en XAML :

```
<!--XAML-->
<RichTextBox SpellCheck.IsEnabled="True" Language="en-us" Height="100"
Name="DFRichTextBox" Width="200" />
```

Comme vous pouvez le voir ici, nous avons décidé d'implémenter à notre RichTextBox l'option de correction orthographique qui peut s'avérer être très pratique.

Pour ce faire, nous avons utilisé la propriété SpellCheck.IsEnabled qu'on a défini à True puis on a spécifié la langue grâce à la propriété Language. Ainsi en deux temps trois mouvements on a une entrée texte avec correction orthographique dans notre programme.

Et voici le résultat :



2.2 Boutons

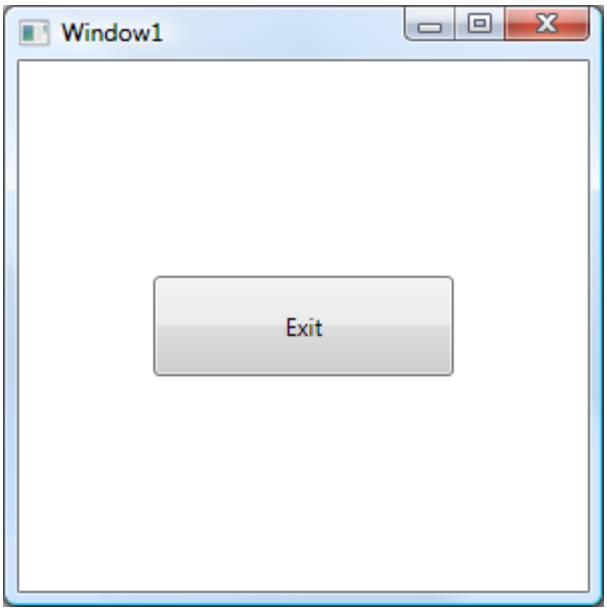
Nous allons maintenant baser notre étude sur un contrôle de base mais très important dans notre application, ce contrôle est bien sûr le bouton en WPF `System.Windows.Controls.Button`.

Concrètement, le bouton va permettre à votre utilisateur d'exécuter l'action en cours de façon explicite.

Pour créer un bouton, rien de très difficile, tout en continuant dans notre lancé sur du code XAML, voici comment se déroule la déclaration d'un bouton en XAML :

```
<!--XAML-->
<Button Height="50" Name="DFButton" Width="150">Exit</Button>
```

Et voici le Résultat :



On ne se penchera pas ici sur la gestion des événements liés à ce bouton car comme énoncé précédemment les événements sont le sujet des chapitres suivants que je vous invite donc à consulter.

2.3 Gestion de formulaires

Nous allons continuer notre découverte des contrôles de WPF par celle des contrôles de formulaires. Un formulaire est clairement ce qui va proposer à l'utilisateur un espace de saisi dans l'interface de votre programme, cet espace de saisi n'est bien sûr pas limité à des contrôles comme les Textbox ou Richtextbox mais peut aussi être des CheckBox, RadioButton ou encore ComboBox comme on pourra le voir dans cette partie.

2.3.1 CheckBox

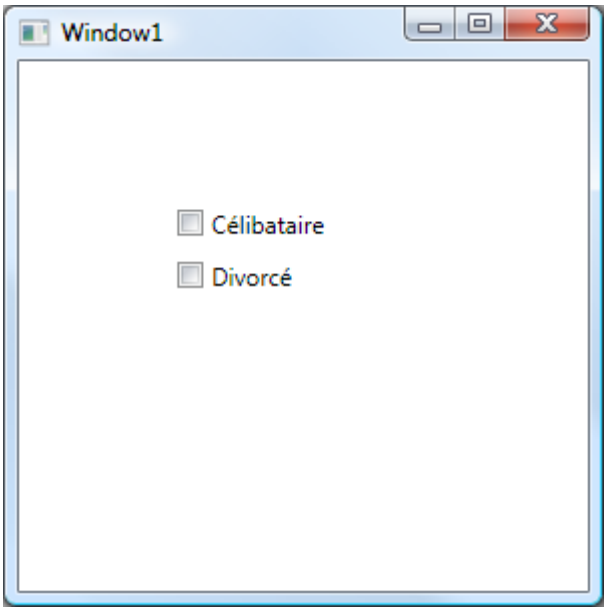
Les CheckBox ou `System.Windows.Controls.Checkbox` sont utilisé dans une application généralement pour présenter une option à l'utilisateur qu'il peut activer ou désactiver. Les CheckBox peuvent avoir par contre 3 états : activé, désactivé et indéterminé.

Vous pouvez utiliser une CheckBox seule, comme par exemple pour que l'utilisateur décide de se loger automatiquement… Ou plusieurs pour la même question comme par exemple pour un moteur de recherche immobilier ou on choisirait de rechercher une villa avec 2 OU 3 chambres.

Voici comment en XAML on déclare une CheckBox :

```
<!--XAML-->
<CheckBox Height="16" Name="DFCheckBox1" Margin="79,0,79,102">
      Célibataire
</CheckBox>
<CheckBox Height="16" Name="DFCheckBox2" Margin="79,0,79,50">
      Divorcé
</CheckBox>
```

Comme on peut l'observer rien de très compliqué encore une fois, voici le résultat que nous donne ce code :



Pour exploiter en C# une TextBox il suffit d'utiliser la méthode `IsChecked` qui renvoie un booléen, true si coché false dans le cas contraire.

2.3.2 RadioButton

Pour continuer sur les contrôles de formulaires on va étudier maintenant le RadioButton ou `System.Windows.Controls.RadioButton`. Le but d'un radio bouton est de donner le choix et un seul et unique choix parmi un ensemble de propositions/options. En effet, on ne peut sélectionner qu'un seul RadioButton à la fois.

C'est la seule convergence qu'il y a entre la CheckBox et le Radiobutton. En effet on va continuer d'utiliser la méthode `IsChecked` pour vérifier si le RadioButton est sélectionné ou pas.

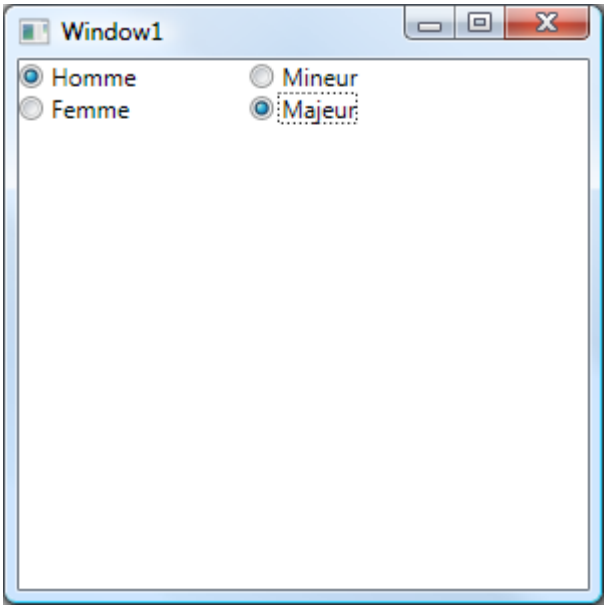
Généralement pour ce type de sélection nous apprécions utiliser un StackPanel, et ce sera l'occasion de mettre en pratique concrètement le chapitre précédent sur le layout WPF.

Voici comment on déclare des RadioButtons. En XAML :

```
<!--XAML-->
<Window x:Class="WPFDotnetFrance.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">

    <Grid>
        <StackPanel>
            <RadioButton GroupName="sexe">Homme</RadioButton>
            <RadioButton GroupName="sexe">Femme</RadioButton>
        </StackPanel>
        <StackPanel HorizontalAlignment="Center">
            <RadioButton GroupName="age">Mineur</RadioButton>
            <RadioButton GroupName="age">Majeur</RadioButton>
        </StackPanel>
    </Grid>
</Window>
```

Comme vous pouvez le voir on a séparer deux groupes de Textbox grâce à la propriété GroupName qui va nous permettre donc d'avoir deux groupes différent de RadioButton donc ici on va pouvoir dire si on est une Femme ou un Homme ET si on est mineur ou majeur.

Voici le résultat :



2.3.3 ComboBox

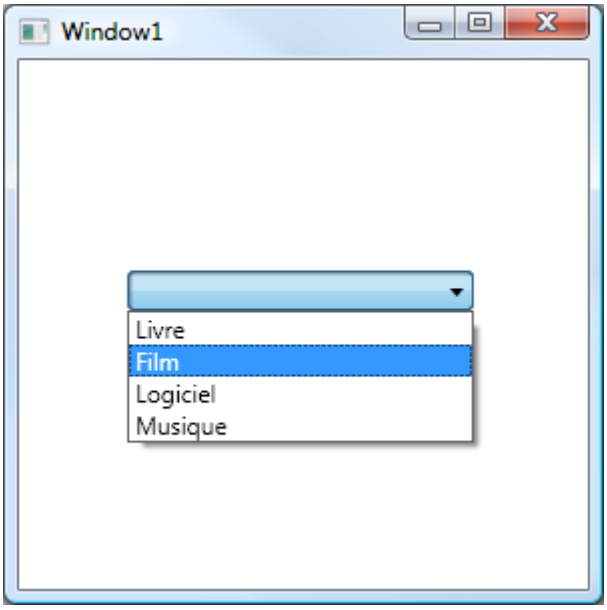
Nous allons poursuivre en vous présentant le ComboBox ou System.Windows.Controls.Combobox. Un ComboBox est un contrôle qui va nous permettre d'afficher une liste déroulante d'un clic ou de la masquer de la même façon.

Voici en XAML comment on peut définir une ComboBox :

```
<!--XAML-->
<Window x:Class="WPFDotnetFrance.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">

    <Grid>
        <ComboBox Height="20" Margin="54,105,57,0"
VerticalAlignment="Top">
            <ComboBoxItem>Livre</ComboBoxItem>
            <ComboBoxItem>Film</ComboBoxItem>
            <ComboBoxItem>Logiciel</ComboBoxItem>
            <ComboBoxItem>Musique</ComboBoxItem>
        </ComboBox>
    </Grid>
</Window>
```

Comme vous pouvez le voir c'est assez facile, nous allons utiliser une balise principal ComboBox à laquelle nous allons passer les propriétés habituelles c'est-à-dire Height, Width Margin etc. Et dans cette dernière nous allons ajouter les balises ComboBoxItem qui vont nous permettre de définir un item dans notre combobox tout simplement.

Voici le résultat final :



2.4 Listes et hiérarchies de données

2.4.1 ListView

La ListView va vous permettre d'ordonner et classer des données dans une liste à la manière de ce que l'on peut trouver dans l'explorateur de répertoire de Windows.

On utilise généralement le contrôle ListView avec le Data Binding qui permet de lier des données à un contrôle très simplement. Comme nous n'avons pas vu comment utiliser le Data Binding pour l'instant, nous allons nous contenter d'un exemple sommaire.

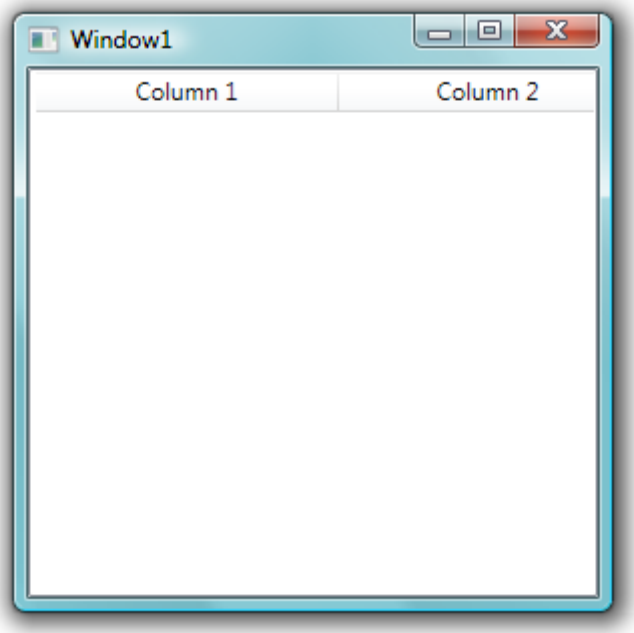
Afin d'utiliser une ListView, il faut préalablement la déclarer dans votre fichier XAML, puis ensuite modeler son patron, à base de GridView. Voici un exemple de ListView :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
   <ListView Name="MyListView">
        <ListView.View>
            <GridView>
                <GridViewColumn Width="150" Header="Column 1" />
                <GridViewColumn Width="150" Header="Column 2" />
            </GridView>
        </ListView.View>
    </ListView>
</Window>
```

Nous venons donc de créer une ListView que nous avons déclaré avec les balises ListView, et nous lui avons donné un nom. Ensuite, nous indiquons entre les balises ListView.View le patron de nos ListView.

Les balises GridView permettent de délimiter la grille que nous allons dessiner dans la ListView. A l'intérieur nous trouvons deux balises GridViewColumn qui vont nous permettre d'à la fois nommer chaque colonne, mais aussi de remplir chaque ligne grâce au Data Binding. Nous aurons l'occasion de vous montrer comment remplir une ListView ou tout autre contrôle dans le chapitre dédié au Data Binding.

Si nous compilons notre exemple, nous avons bien à faire à une ListView vide :



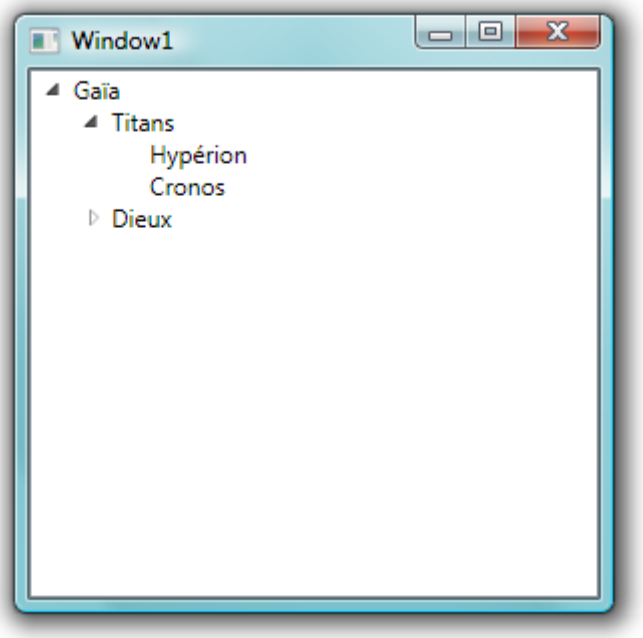
2.4.2 TreeView

TreeView va vous permettre un peu à la manière de ListView d'ordonner des données, mais cette fois ci non plus dans une liste, mais plutôt dans une hiérarchie.

L'utilisation de TreeView ressemble un peu à celui de ListView, mais il à l'avantage de proposer d'imbriquer plusieurs items afin de créer vos arbres. Un TreeView se compose donc tout simplement de TreeViewItem auquel on définit une valeur à l'attribut Header. Voici un exemple de TreeView :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <TreeView>
        <TreeViewItem Header="Gaïa">
            <TreeViewItem Header="Titans">
                <TreeViewItem Header="Hypérion" />
                <TreeViewItem Header="Cronos" />
            </TreeViewItem>
            <TreeViewItem Header="Dieux">
                <TreeViewItem Header="Zeus" />
                <TreeViewItem Header="Déméter" />
            </TreeViewItem>
        </TreeViewItem>
    </TreeView>
</Window>
```

Ce qui donne :



3 Les contrôles complexes

3.1 Les éléments principaux

3.1.1 Menu

Si on ne devait retenir qu'un seul contrôle complexe, on choisirait à coup sûr Menu. En effet, celui-ci va nous permettre de créer les barres de menus traditionnelles que l'on retrouve dans toutes les applications que vous connaissez (Fichier, Edition, Affichage…) ainsi que des menus contextuels, le tout grâce à quelques balises XAML.

Pour créer un menu, nous allons utiliser principalement deux classes, Menu et MenuItem.

La classe Menu nous permet de grouper un ensemble de MenuItem afin de composer notre menu.

La Classe MenuItem dérive du type `Windows.Controls.HeaderedItemsControl` et nous permet ainsi d'utiliser deux propriétés indispensable : `Header` et `Items`

`Header` va nous permettre d'indiquer le titre du menu, par exemple, le traditionnel « Fichier », `Item` elle permet d'indiquer tous les enfants de ce menu, on va y retrouver donc « Ouvrir » , « Enregistrer » etc.

Note : Pour donner une valeur à Item, vous pouvez par défaut l'indiquer dans les balises ouvrante et fermante de votre MenuItem. Généralement, en XAML, on écrit donc jamais la propriété Item. En C# ou VB.NET, vous devrez par contre l'utiliser.

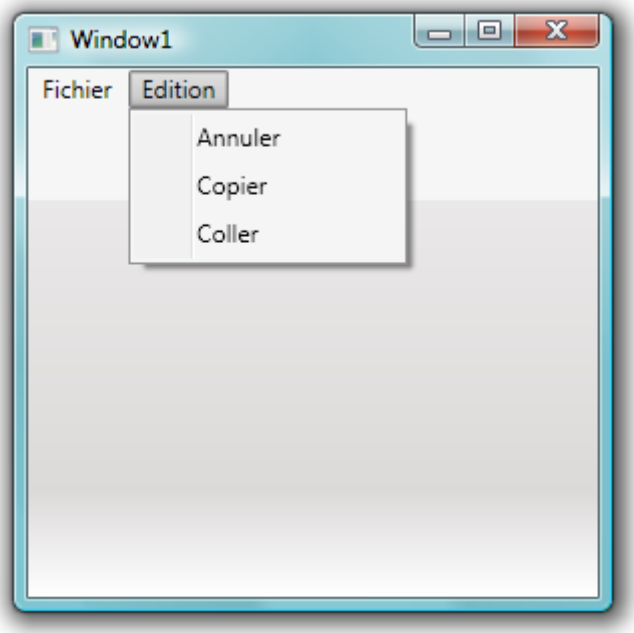
Voyons maintenant un premier exemple d'utilisation de MenuItem, celui d'une petite barre de Menu :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Grid x:Name="LayoutRoot">
        <MenuItem Header="Fichier">
            <MenuItem Header="Nouveau" />
            <MenuItem Header="Ouvrir" />
            <MenuItem Header="Enregistrer" />
        </MenuItem>
    </Grid>
</Window>
```

A ce stade là, si vous compilez, vous verrez s'afficher un énorme bouton Fichier avec une petite flèche à son extrémité droite. De plus, si vous cliquez sur le bouton, rien ne se passera. Nous allons rajouter maintenant les balises Menu dont nous avons parlé plus haut afin d'englober tous nos MenuItem en une entité unique. Nous en profitons également pour rajouter un second MenuItem : « Edition » :

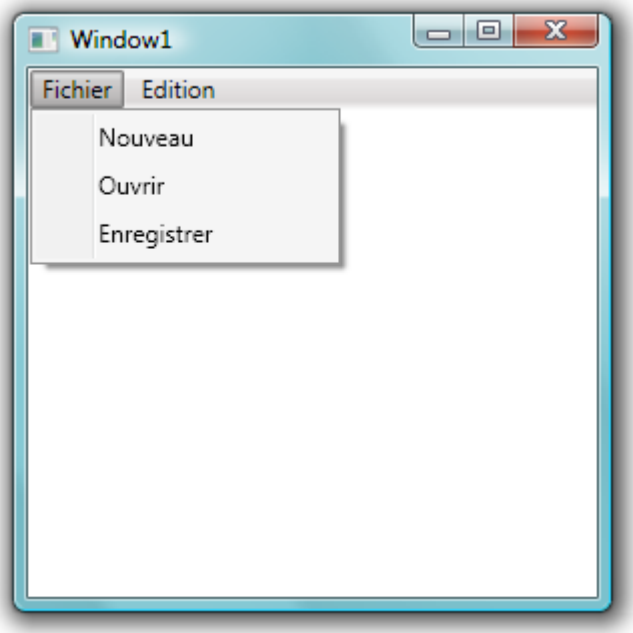
```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Grid x:Name="LayoutRoot">
        <Menu>
            <MenuItem Header="Fichier">
                <MenuItem Header="Nouveau" />
                <MenuItem Header="Ouvrir" />
                <MenuItem Header="Enregistrer" />
            </MenuItem>
            <MenuItem Header="Edition">
                <MenuItem Header="Annuler" />
                <MenuItem Header="Copier" />
                <MenuItem Header="Coller" />
            </MenuItem>
        </Menu>
    </Grid>
</Window>
```

A ce stade, notre exemple commence à ressembler à une barre de menu :



Nous ne sommes cela dit pas tout à fait satisfait de son apparence, nous allons y remédier en utilisant un layout vu au chapitre précédent : DockPanel. Voici l'apparence que va prendre notre menu si nous le collons tout en haut de notre fenêtre, et que nous rajoutons une zone de texte en dessous :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Grid x:Name="LayoutRoot">
        <DockPanel>
            <Menu DockPanel.Dock="Top">
                <MenuItem Header="Fichier">
                    <MenuItem Header="Nouveau" />
                    <MenuItem Header="Ouvrir" />
                    <MenuItem Header="Enregistrer" />
                </MenuItem>
                <MenuItem Header="Edition">
                    <MenuItem Header="Annuler" />
                    <MenuItem Header="Copier" />
                    <MenuItem Header="Coller" />
                </MenuItem>
            </Menu>
            <TextBlock />
        </DockPanel>
    </Grid>
</Window>
```



Nous avons finalement un menu correctement dessiné.

Maintenant que nous avons vu le cas de la barre de menu, nous allons examiner un exemple de menu contextuel. Nous allons rester très succinct l'exemple reprenant dans l'ensemble les notions vu précédemment.

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Grid x:Name="LayoutRoot">
        <DockPanel>
            <Menu DockPanel.Dock="Top"></Menu>
            <TextBlock x:Name="TextContent"
ContextMenuOpening="MAJ_Delete_Item">
                <TextBlock.ContextMenu>
                    <ContextMenu>
                        <MenuItem Header="Ajouter">
                            <MenuItem x:Name="AjouterMot" Header="Mot"
Click="AjouterMot_Click" />
                            <MenuItem x:Name="AjouterPhrase"
Header="Phrase" Click="AjouterPhrase_Click" />
                        </MenuItem>
                        <MenuItem x:Name="Delete" Header="Effacer"
Click="Delete_Click"/>
                    </ContextMenu>
                </TextBlock.ContextMenu>
            </TextBlock>
        </DockPanel>
    </Grid>
</Window>
```

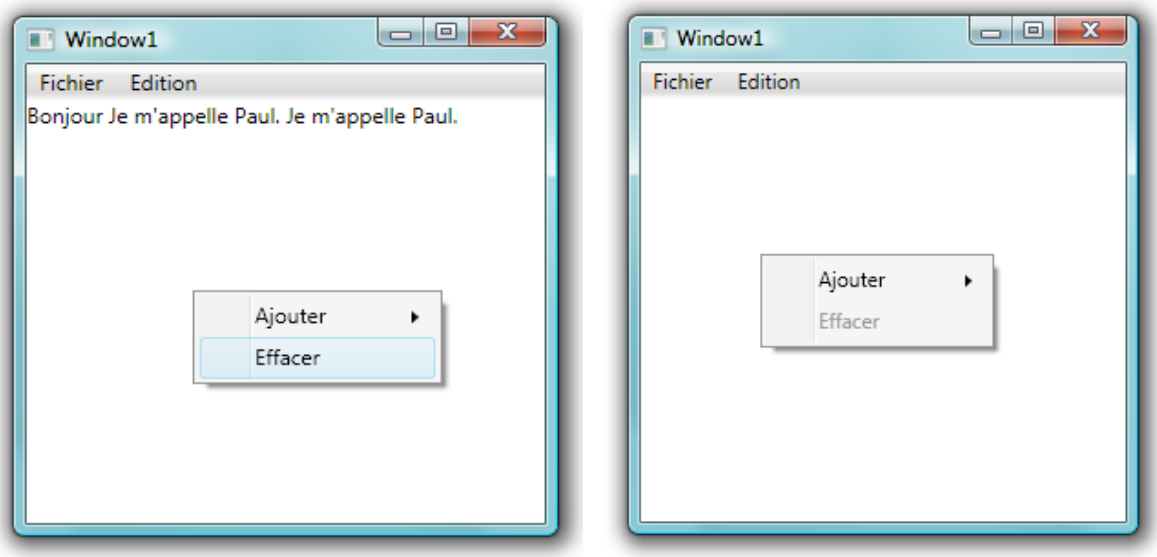
Nous avons donc sous notre précédent menu ajouté un TextBlock. Nous lui donnons un nom, et lui faisons souscrire à l'évènement ContextMenuOpening qui va nous permettre d'exécuter une méthode à l'ouverture du menu contextuel.

A l'intérieur de ce TextBlock, nous avons les balises TextBlock.ContextMenu qui permettent d'associer un menu contextuel à ce TextBlock. Lorsque l'on cliquera sur le bouton droit à l'intérieur, notre menu s'ouvrira. Enfin nous encapsulons nos MenuItem dans les balises ContextMenu, de la même manière que pour la barre de Menu.

Vous remarquerez que nous avons donné des noms et souscrit aux évènements Click de nos MenuItem afin d'ajouter un peu d'interactivité à ce menu. Vous devrez définir le comportement de toutes les méthodes appelées dans le fichier C# ou VB.NET associé à votre XAML. Voici un exemple en C# :

```
//C#
public partial class Window1 : Window
{
    public Window1()
    {
        this.InitializeComponent();
    }

    private void Delete_Click(object sender, RoutedEventArgs e)
    {
        if (TextContent.Text != "")
            TextContent.Text = "";
    }

    private void AjouterMot_Click(object sender, RoutedEventArgs e)
    {
        TextContent.Text += "Bonjour ";
    }

    private void AjouterPhrase_Click(object sender, RoutedEventArgs e)
    {
        TextContent.Text += "Je m'appelle Paul. ";
    }

    private void MAJ_Delete_Item(object sender, RoutedEventArgs e)
    {
        Delete.IsEnabled = (TextContent.Text != "");
    }
}
```

```
' VB
Partial Public Class Window1
    Inherits Window

    Public Sub New()
        Me.InitializeComponent()
    End Sub

    Private Sub Delete_Click(ByVal sender As Object, ByVal e As
RoutedEventArgs)
        If TextContent.Text <> "" Then
            TextContent.Text = ""
        End If
    End Sub

    Private Sub AjouterMot_Click(ByVal sender As Object, ByVal e As
RoutedEventArgs)
        TextContent.Text += "Bonjour "
    End Sub

    Private Sub AjouterPhrase_Click(ByVal sender As Object, ByVal e As
RoutedEventArgs)
        TextContent.Text += "Je m'appelle Paul. "
    End Sub

    Private Sub MAJ_Delete_Item(ByVal sender As Object, ByVal e As
RoutedEventArgs)
        Delete.IsEnabled = (TextContent.Text <> "")
    End Sub

End Class
```



Le code n'est pas bien compliqué, remarquez juste la méthode MAJ_Delete_Item qui rend disponible ou non le bouton Effacer selon le contenu de notre TextBlock.

3.1.2 ToolBar et StatusBar

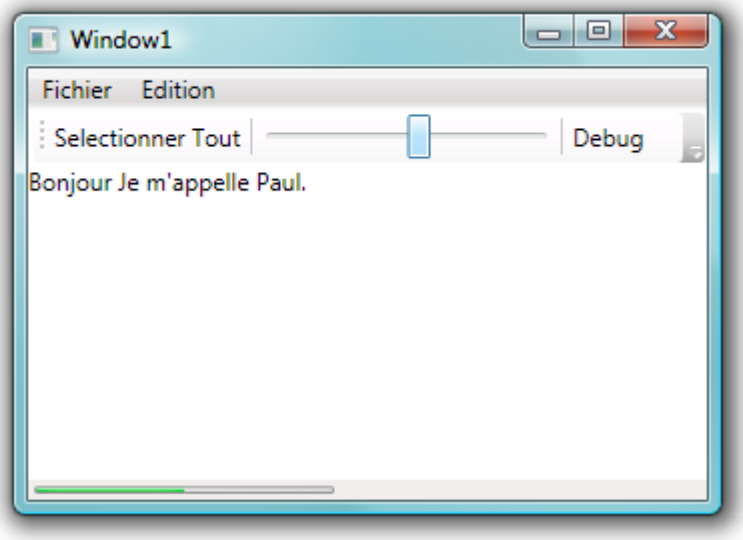
La création d'une barre d'outils et d'une barre de statut est vraiment simple en WPF. Vous disposez pour cela de deux classes : `Toolbar` et `StatusBar` dérivant toutes les deux de `System.Windows.Controls.ItemsControl` dont elles récupèrent la propriété Items.

Item est une collection d'objets de type **`object`**, vous pouvez donc englober dans vos barres n'importe quel objet, que ce soit un bouton, une image ou un slider.

Voyons tout de suite un exemple d'utilisation, nous allons créer une fenêtre contenant nos deux barres dockée en haut et en bas.

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Grid x:Name="LayoutRoot">
        <DockPanel>
            <Menu DockPanel.Dock="Top"></Menu>
            <ToolBar DockPanel.Dock="Top">
                <Button Content="Selectionner Tout" />
                <Separator />
                <Slider Width="150" />
                <Separator />
                <CheckBox>Debug</CheckBox>
            </ToolBar>
            <StatusBar DockPanel.Dock="Bottom">
                <ProgressBar Width="150" Value="50" />
            </StatusBar>
            <TextBlock />
        </DockPanel>
    </Grid>
</Window>
```

Ce qui nous donne simplement :



Le code XAML ne représente, comme vous le voyez, aucune difficulté, on englobe simplement les contrôles entre les balises <StatusBar /> ou <ToolBar />. Difficile de faire plus simple.

3.2 Les éléments secondaires

3.2.1 TabControl

TabControl va nous permettre de créer des onglets que vous pouvez retrouver couramment dans Windows et divers autres applications.

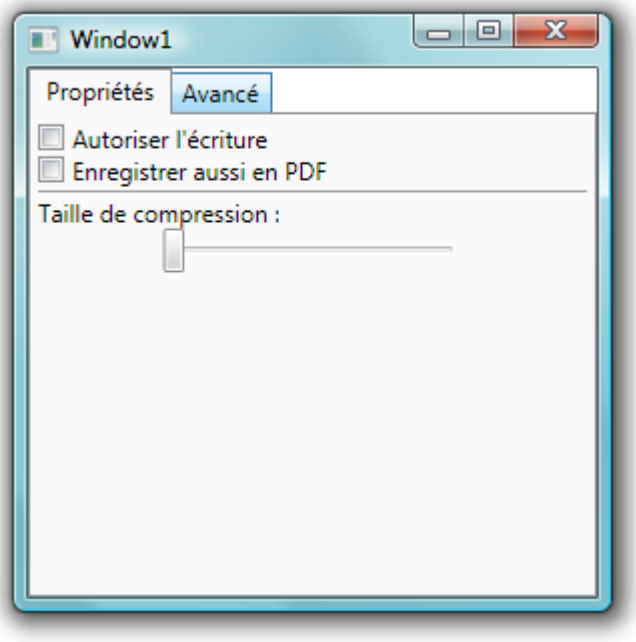
Tout comme les barres vu précédemment, TabControl dérive de ItemsControl, vous pouvez donc utiliser tous les objets que vous souhaitez afin de remplir vos onglets. Généralement, on remplit nos TabControl avec des TabItem.

TabItem dérive de HeaderedContentControl, dont il reprend les propriétés Header (pour le titre de l'onglet) et Content (pour le contenu de l'onglet). Ces deux propriétés acceptent tout type d'objet mais ne sont pas des collections ! Vous devrez donc utiliser un Layout si vous voulez disposer plusieurs contrôles dans le Content ou le Header.

Voyons tout de suite un exemple simple de l'utilisation des TabControl.

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Grid x:Name="LayoutRoot">
        <TabControl>
            <TabItem Header="Propriétés">
                <StackPanel>
                    <CheckBox>Autoriser l'écriture</CheckBox>
                    <CheckBox>Enregistrer aussi en PDF</CheckBox>
                    <Separator />
                    <TextBlock>Taille de compression :</TextBlock>
                    <Slider Width="150" />
                </StackPanel>
            </TabItem>
            <TabItem Header="Avancé">
                <CheckBox>Activer le mode Avancé ?</CheckBox>
            </TabItem>
        </TabControl>
    </Grid>
</Window>
```

Ce qui donne :



Là encore le code ne représente pas une énorme difficulté, l'exemple ressemble beaucoup au premier Menu que nous avons produit plus haut. Attention de ne pas confondre TabControl et TabItem cela dit.

Remarque : Par défaut, le premier TabItem est toujours affiché en premier. Il est possible de changer cela en mettant à True l'attribut IsSelected du TabItem à afficher en premier.

3.2.2 Expander

L'Expander est un contrôle complexe permettant de créer des zones enroulable au sein de votre programme. Il dérive tout comme TabItem de HeaderedContentControl, il en récupère donc les propriétés Header et Content.

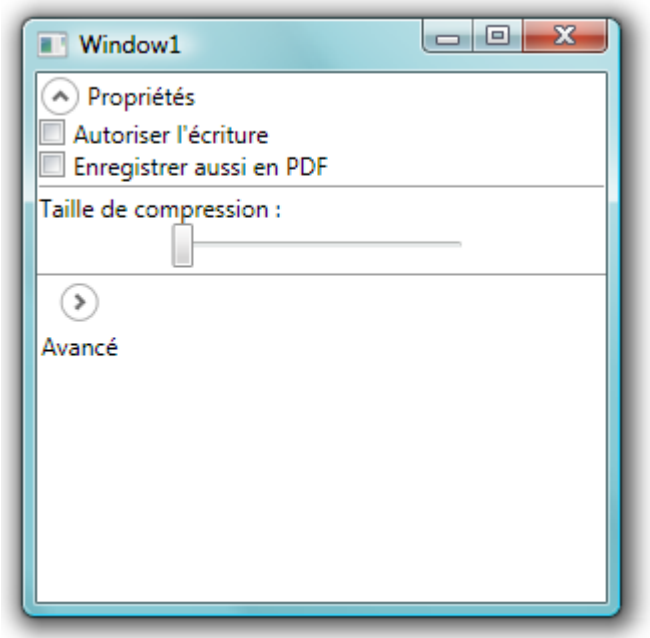
Plus difficile à décrire qu'à inclure dans vos programmes, nous allons tout de suite passer à l'exemple.

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <StackPanel>
        <Expander Header="Propriétés" IsExpanded="True">
            <StackPanel>
                <CheckBox>Autoriser l'écriture</CheckBox>
                <CheckBox>Enregistrer aussi en PDF</CheckBox>
                <Separator />
                <TextBlock>Taille de compression :</TextBlock>
                <Slider Width="150" />
            </StackPanel>
        </Expander>
        <Separator />
        <Expander Header="Avancé" ExpandDirection="Right">
            <CheckBox>Activer le mode Avancé ?</CheckBox>
        </Expander>
    </StackPanel>
</Window>
```

Nous avons donc créé deux Expander sur le même modèle que l'exemple sur TabControl :

- Le premier Expander contient un StackPanel contenant lui-même divers contrôles. Il a pour en-tête « Propriétés » et possède également un attribut IsExpanded à True lui permettant d'être déroulé par défaut.
- Le Second Expander ne contient qu'un seul contrôle (une checkbox), son en-tête est « Avancé » et il possède l'attribut ExpandDirection définit à Right qui permet d'indiquer que l'Expander devra se dérouler vers la droite.

Ce qui donne :



Remarquez la flèche orientée vers la droite du second Expander.

4 Les contrôles utilisateurs

4.1 Qu'est ce qu'un UserControl ?

Si vous avez connu les WinForms, vous allez être un peu chamboulé par les UserControl en WPF. En effet, alors qu'auparavant ils servaient à étendre n'importe quel contrôle pour l'adapter, graphiquement on fonctionnellement à son application, on va dorénavant séparer la partie graphique et fonctionnelle de nos contrôles.

Un UserControl va donc consister en WPF à un regroupement de contrôles qui n'en forment plus qu'un et dont on peut modifier le comportement.

Ainsi nous allons pouvoir créer par exemple un UserControl qui va nous servir de formulaire de Login afin de pouvoir le réutiliser facilement dans plusieurs applications.

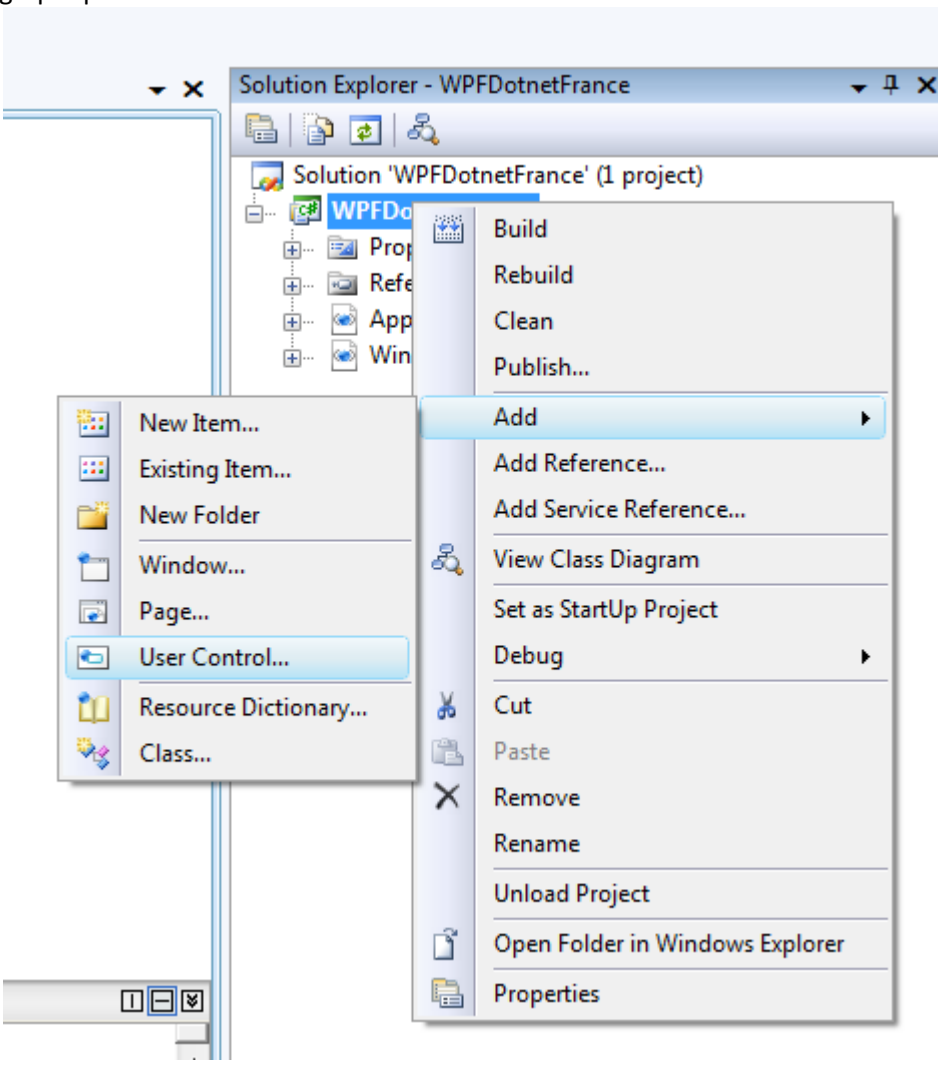
Nous n'aborderons pas dans ce chapitre comment modifier l'apparence de notre UserControl, en effet les notions requises à la création d'un Custom Control (puisque c'est son nom) seront vues dans le chapitre suivant.

4.2 Création pas à pas d'un UserControl

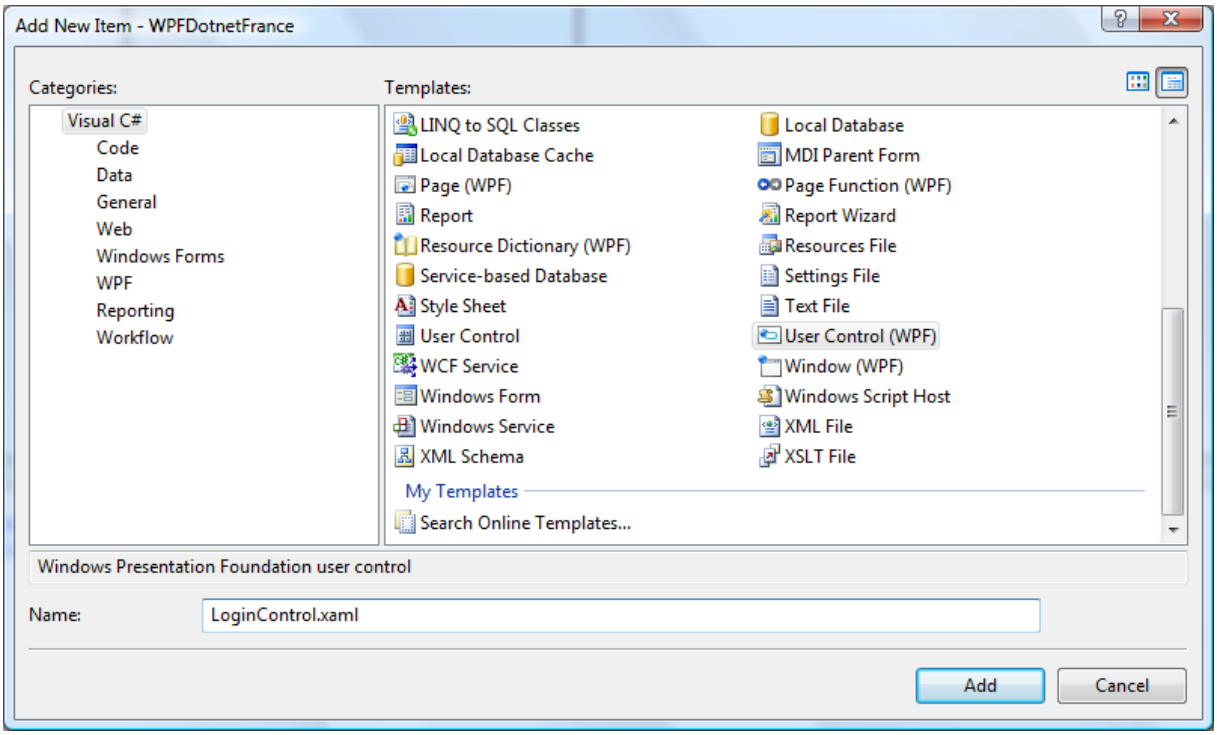
Pour la création de notre UserControl nous allons créer un projet de base WPF. Nous ne reviendrons pas ici sur la création d'un projet, si vous avez un problème à ce niveau là nous vous invitons à consulter le Chapitre 1 d'introduction à WPF.

4.2.1 Interface Graphique

Une fois le projet créé, nous allons maintenant ajouter un nouvel UserControl via l'interface graphique de Visual Studio :



Concrètement que va faire Visual Studio ? Tout simplement ouvrir la fenêtre Add ➔ New Item et pointer le bon composant. De ce fait vous n'aurez plus qu'à lui donner son nom :



Comme vous pouvez le remarquer, notre UserControl durant la durée de ce cours sera nommé LoginControl.

Durant cette première partie, nous allons nous intéresser à l'interface graphique de notre contrôle puis nous verrons par la suite le code behind.

```
<!--XAML-->
<UserControl x:Class="WPF.LoginControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="300" Width="300">
    <Grid>

    </Grid>
</UserControl>
```

Comme vous pouvez le voir, on utilise la balise <UserControl> au lieu de <Window> lorsque l'on travaille sur les UserControl.

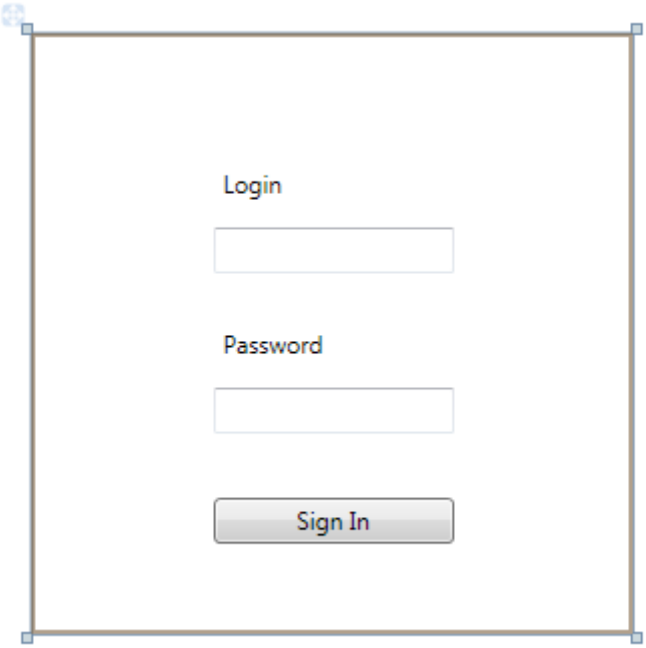
Maintenant nous allons ajouter divers contrôles à notre UserControl car comme on a pu le voir précédemment, un UserControl est tout simplement un Control contenant d'autres contrôles.

Ici le but est de créer un UserControl de Login qui contient un champ login un champ password des labels de description et un bouton de validation.

Voici ce que cela donne :

```
<!--XAML-->
<UserControl x:Class="WPF.LoginControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="300" Width="300">
    <Grid>
        <Label Height="28" Margin="91,63,89,0" Name="LoginLabel"
VerticalAlignment="Top">Login</Label>
        <Label Margin="91,143,89,129"
Name="PasswordLabel">Password</Label>
        <TextBox Height="23" Margin="91,97,89,0" Name="LoginTextBox"
VerticalAlignment="Top" />
        <PasswordBox Height="23" Margin="91,0,89,100"
Name="PasswordTextBox" VerticalAlignment="Bottom" />
        <Button Height="23" Margin="91,0,89,45" Name="SignInButton"
VerticalAlignment="Bottom">Sign In</Button>
    </Grid>
</UserControl>
```

Tous ces contrôles ayant été vu précédemment vous devriez comprendre ce code aisément.
Voici le résultat :



4.2.2 Code Behind

Nous allons maintenant passer au code behind de notre UserControl.

Dans ce cas l'essentiel était de montrer ce qu'été un UserControl, comment créer un UserControl et finalement comment l'utiliser. C'est pourquoi vous l'aurez compris le code behind que nous proposons est extrêmement simple. Vous pourrez cela dit totalement étendre le comportement de votre UserControl sans restriction.

L'exemple consiste donc simplement à gérer l'événement Click du bouton SignIn et afficher une MessageBox contenant l'utilisateur.

```
//C#
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;

namespace WPF
{
    /// <summary>
    /// Interaction logic for LoginControl.xaml
    /// </summary>
    public partial class LoginControl : UserControl
    {
        public LoginControl()
        {
            InitializeComponent();
        }

        private void SignInButton_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show(LoginTextBox.Text);
        }
    }
}
```

```
Imports System
Imports System.Collections.Generic
Imports System.Linq
Imports System.Text
Imports System.Windows
Imports System.Windows.Controls

Namespace WPF
    ''' <summary>
    ''' Interaction logic for LoginControl.xaml
    ''' </summary>
    Partial Public Class LoginControl
        Inherits UserControl

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub SignInButton_Click(ByVal sender As Object, ByVal e As
RoutedEventArgs)
            MessageBox.Show(LoginTextBox.Text)
        End Sub

    End Class

End Namespace
```

Comme vous pouvez le voir il n'y a vraiment rien de compliqué au niveau fonctionnel. Mais n'oubliez pas de lier l'évènement Click du bouton à la méthode SignInButton_Click.

4.2.3 Réutilisation d'un UserControl

Maintenant que nous avons créé notre UserControl, nous allons pouvoir le réutiliser à volonté dans nos applications. Pour cela, nous allons simplement le rajouter en XAML comme nous l'aurions fait pour n'importe quel contrôle :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      xmlns:lc="clr-namespace:WPF"
      Width="300" Height="300">
    <Grid>
        <lc:LoginControl></lc:LoginControl>
    </Grid>
</Window>
```

Attention toutefois, comme notre User Control n'appartient pas au namespace fournit par les .NET Framework, nous allons devoir indiquer à notre application de l'utiliser. Nous avons donc rajouté la ligne suivante dans notre balise <Window> :

```
<!--XAML-->
xmlns:lc="clr-namespace:WPF"
```

Ainsi, nous faisons l'équivalent d'un using en C# ou Imports en VB.NET.

Vous devrez finalement préfixer le nom de votre UserControl par le nom du namespace XAML. Ici nous avons choisit « lc » pour « LoginControl ».

```
<!--XAML-->
<lc:LoginControl></lc:LoginControl>
```

5 Conclusion

Pour conclure, on a pu voir tout au long de ce chapitre ce qu'été concrètement un contrôle WPF, comment se servir des contrôles de base comme les boutons les TextBlocks etc. Mais également des contrôles plus complexes comme les menus ou Expanders. Enfin nous avons vu comment créer ces UserControl.

Nous n'avons pas parlé dans ce chapitre des contrôles médias, car même si c'est une partie importante et une nouveauté de WPF, nous avons préféré leur dédier un chapitre spécial.

Nous n'avons pas décrit en détail toutes les propriétés de chaque contrôle dans ce chapitre pour une raison de place. Afin d'approfondir l'étude d'un contrôle en particulier, et connaitre toutes ses propriétés, rendez vous sur la doc MSDN : http://msdn.microsoft.com/fr-fr/library/ms752324.aspx

Pour élargir notre étude de WPF, je vous invite à suivre le chapitre suivant qui traitera des Templates, Styles et Custom Control.