Cyril Seguenot

2016

Résumé

Support d’accompagnement d’une formation à la programmation avec WPF faite en présentiel

Formation à WPF

Support de cours

Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l’auteur ou de ses ayants droit ou ayants cause est illicite selon le Code de la propriété intellectuelle (Art L 122-4) et constitue une contrefaçon réprimée par le Code pénal.

Seules sont autorisées (Art L122-5) les copies ou reproductions strictement réservées à l’usage privé du copiste et non destinées à une utilisation collective, ainsi que les analyses et courtes citations justifiées par le caractère critique, pédagogique ou d’information de l’œuvre à laquelle elles sont incorporées, sous réserve, toutefois, du respect des dispositions des articles L 122-10 à L 122-12 du même Code, relatives à la reproduction par reprographie.

©Cyril Seguenot 2016

SOMMAIRE

[1 Introduction 5](#_Toc476851810)

[2 Présentation de WPF 5](#_Toc476851811)

[2.1 Les apports de WPF 5](#_Toc476851812)

[2.2 Caractéristiques techniques 6](#_Toc476851813)

[2.3 Evolution 6](#_Toc476851814)

[3 Construire une interface visuelle 6](#_Toc476851815)

[3.1 Les contrôles WPF 6](#_Toc476851816)

[3.2 La disposition (layout) 10](#_Toc476851817)

[3.3 Le contenu 24](#_Toc476851818)

[3.4 Les collections d’éléments 27](#_Toc476851819)

[4 Les espaces de noms 32](#_Toc476851820)

[4.1 Déclaration des espaces de noms 32](#_Toc476851821)

[4.2 Accès aux types d’un espace de noms 33](#_Toc476851822)

[5 Les ressources 34](#_Toc476851823)

[5.1 Créer une ressource 35](#_Toc476851824)

[5.2 Utiliser une ressource 39](#_Toc476851825)

[6 Les styles 41](#_Toc476851826)

[6.1 Création d’un style 41](#_Toc476851827)

[6.2 Application d’un style 42](#_Toc476851828)

[6.3 Héritage 43](#_Toc476851829)

[6.4 Triggers 43](#_Toc476851830)

[7 La liaison de données (DataBinding) 44](#_Toc476851831)

[7.1 Principe du DataBinding 44](#_Toc476851832)

[7.2 Propriétés Mode et UpdateSourceTrigger 45](#_Toc476851833)

[7.3 Liaison avec un objet unique 46](#_Toc476851834)

[7.4 Liaison avec une collections d’objets 51](#_Toc476851835)

[7.5 Liaisons avec d’autres sources de données 55](#_Toc476851836)

[7.6 DataTemplate 56](#_Toc476851837)

[7.7 Navigation, tri, filtrage et regroupement 58](#_Toc476851838)

[7.8 Débogage 63](#_Toc476851839)

[8 Conversion, validation et notification des changements 65](#_Toc476851840)

[8.1 Conversion 65](#_Toc476851841)

[8.2 Validation 68](#_Toc476851842)

[8.3 Notification des changements 70](#_Toc476851843)

[9 Les évènements routés 72](#_Toc476851844)

[9.1 Présentation 72](#_Toc476851845)

[9.2 Paramètres 74](#_Toc476851846)

[9.3 Evènement attaché 74](#_Toc476851847)

[9.4 Mise en œuvre 75](#_Toc476851848)

[10 Les commandes 77](#_Toc476851849)

[10.1 Présentation 77](#_Toc476851850)

[10.2 Evènements vs commandes 78](#_Toc476851851)

[10.3 Commandes routées 79](#_Toc476851852)

[10.4 Commandes relais 82](#_Toc476851853)

[11 Le modèle MVVM 85](#_Toc476851854)

[12 Les contrôles utilisateur (UserControl) 87](#_Toc476851855)

[12.1 Créer et utiliser un UserControl 87](#_Toc476851856)

[12.2 Ajouter des propriétés 88](#_Toc476851857)

[13 Références bibliographiques 90](#_Toc476851858)

# Introduction

Le présent document sert de support pour une formation en présentiel. Il explique de façon synthétique les notions de base de la programmation avec WPF (Windows Presentation Fundation), et n’a pas vocation à remplacer un ouvrage de référence.

Les notions sont illustrées par des exemples de code, dont certains font appel à des notions qui ne sont abordées que plus loin dans le support. Il est donc tout à fait normal que vous ne compreniez pas immédiatement toutes les lignes de code. Ceci ne doit pas vous empêcher de continuer, car toutes les notions seront expliquées progressivement.

N’hésitez pas à annoter vous-même ce support tout au long de la formation, afin d’en faire le document le plus utile pour vous-même.

# Présentation de WPF

WPF (Windows Presentation Fundation) est la technologie successeuse de Windows Forms pour la création des interfaces visuelles des applications. Elle a été introduite en 2006 avec le .Net Framework 3.0.  
NB/ WCF (Windows Communication Fundation) a été introduit en même temps.

WPF est utilisée dans les applications du Windows Store (applications Modern UI), mais aussi dans les applications de bureau classiques (Win32), pour créer des interfaces plus modernes et plus riches qu’avec les anciennes technologies (Windows Forms ou autres).

## Les apports de WPF

WPF apporte des bénéfices importants sur le plan **fonctionnel** :

* Comme le rendu graphique est vectoriel, il devient possible de zoomer et d’appliquer toutes sortes d’effets et de transformations aux éléments de l’interface.
* Il est possible de créer des interfaces bien plus riches et personnalisées qu’avec les anciennes technologies (formes, couleurs, effets, animations, graphismes…)

… mais aussi dans le **processus de conception et développement** :

* Sous certaines conditions, WPF permet de découpler complètement la description de l’interface visuelle, de la logique applicative, ce qui amène 2 bénéfices :
  + Le code est plus facile à maintenir et à faire évoluer
  + Le code de la logique applicative est plus facilement testable, notamment avec des tests unitaires
* Le visuel peut être conçu et réalisé par d’autres personnes que le développeur : ergonome, graphiste, designer. La réalisation peut notamment être faite avec les outils Expression Design (création d’illustrations) et Expression Blend (création de l’interface utilisateur en xaml).

En contrepartie :

* WPF nécessite des ressources matérielles plus importantes (carte graphique, mémoire), aussi bien pour le développement que pour l’exécution des applications.
* La maîtrise des techniques de création des interfaces visuelles (langage xaml et techniques de design), demande plus d’effort qu’avec les anciennes technologies
* Le débogage est plus compliqué. En effet, l’interface étant décrite de façon déclarative, les erreurs de liaisons de données ne sont pas détectées au moment de la compilation, mais à l’exécution. Même à l’exécution, on peut facilement passer à côté de certains problèmes, car ils ne font pas planter l’application.

NB/ La plateforme UWP pour Windows 10 introduit une nouvelle syntaxe de databinding, avec des performances améliorées, et une résolution des erreurs au moment de la compilation. Cette syntaxe n’a toutefois pas été portée dans WPF.

## Caractéristiques techniques

Les principales caractéristiques techniques de WPF sont :

* Le rendu graphique vectoriel basé sur DirectX
* La description déclarative de l’interface en XAML (Extensible Application Markup Language), qui est un langage très proche du xml, pour créer des objets (éléments) WPF
* La notion de liaisons de données (databinding) pour relier l’interface visuelle au code applicatif .net, qui s’appuie sur la notion de propriété de dépendance.

[Point d’entrée](https://msdn.microsoft.com/fr-FR/library/ms754130(v=vs.110).aspx) de WPF dans la doc MSDN.

## Evolution

Les technologies de développement récentes, telles que UWP (Universal Windows Plateform) et Xamarin.Forms s’appuient sur les mêmes principes que WPF pour décrire les interfaces visuelles.

Le découplage de l’interface et de la logique applicative reste un principe majeur.

**UWP** introduit des contrôles permettant de créer des interfaces valables sur toutes les catégories d’appareils Windows 10 (PC, tablette, mobile, console…). Ces contrôles permettent la création d’interfaces qui s’adaptent aux différentes résolutions d’écran (responsive design), et aux différents modes de saisies (souris, tactile).

**Xamarin.Forms** permet de décrire les interfaces visuelles d’applications capables de fonctionner sous Android, iOS et Windows 10. Le principe est d’avoir un langage descriptif universel, interprété ensuite de façon spécifique pour chaque système d’exploitation, afin d’utiliser ses contrôles natifs.

# Construire une interface visuelle

## Les contrôles WPF

### Catégories

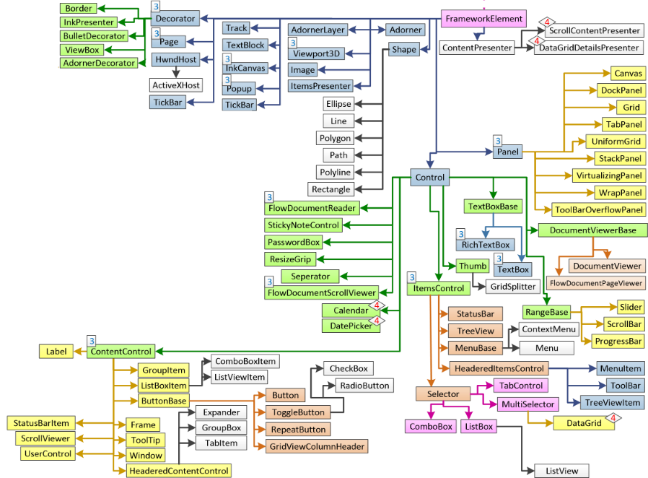
[Cette page MSDN](https://msdn.microsoft.com/fr-fr/library/ms754204(v=vs.110).aspx) présente la liste complète des contrôles WPF classés par catégories.

En voici un résumé :

|  |  |  |
| --- | --- | --- |
| **Catégorie** | **Utilisation** | **Contrôles** |
| Disposition | Gérer la position, la disposition, la taille et les dimensions des éléments enfants | Border, BulletDecorator, Canvas, DockPanel, Expander, Grid, GridSplitter, GroupBox, Panel, ResizeGrip, Separator, ScrollBar, ScrollViewer, StackPanel, Thumb, Viewbox, VirtualizingStackPanel, Window, WrapPanel |
| Boutons | Contrôles d'interface utilisateur les plus élémentaires. En général, les applications exécutent une tâche dans l'événement Click | Button  RepeatButton |
| Affichage des données | Afficher les données à partir d'une source de données. | DataGrid  ListView  TreeView |
| Affichage et sélection de date | Afficher et sélectionner les informations du calendrier | Calendar  DatePicker |
| Menus | Regrouper des actions connexes ou fournir une assistance contextuelle. | ContextMenu  Menu  ToolBar |
| Sélection | Sélectionner un ou plusieurs items parmi une collection | CheckBox  ComboBox  ListBox  RadioButton  Slider |
| Navigation | Améliorent ou étendent la navigation dans l'application avec des liens, des onglets… | Frame  Hyperlink  Page  NavigationWindow  TabControl |
| Boîtes de dialogue | Scénarios communs d'interaction utilisateur, tels que l'impression, la sélection et l’enregistrement de fichiers | OpenFileDialog  PrintDialog  SaveFileDialog |
| Informations utilisateur | Fournissent de l’aide contextuelle ou clarifient l'interface utilisateur d'une application. L'utilisateur ne peut généralement pas interagir avec ces contrôles. | AccessText  Label  Popup  ProgressBar  StatusBar  TextBlock  ToolTip |
| Saisie | Permettent de saisir des données | TextBox  RichTextBox  PasswordBox |
| Documents | WPF inclut des contrôles spécialisés pour consulter des documents. Ces contrôles optimisent l'environnement de lecture selon le scénario ciblé | DocumentViewer  FlowDocumentPageViewer  FlowDocumentReader  FlowDocumentScrollViewer  StickyNoteControl |
| Multimédia | WPF inclut la prise en charge de contenus audio et vidéo, ainsi que des codecs pour la plupart des formats d'image courants | Image  MediaElement  SoundPlayerAction |
| Encre numérique | Prennent en charge les fonctionnalités Tablet PC, telles que la saisie et l’affichage de données manuscrites | InkCanvas  InkPresenter |

### Hiérarchie d’héritage

Tous les éléments WPF dérivent de la classe [FrameworkElement](https://msdn.microsoft.com/fr-fr/library/system.windows.frameworkelement(v=vs.110).aspx), dont voici une représentation de la hiérarchie d’héritage dans .net 4.0 :



Nous nous intéresserons dans ce cours plus particulièrement aux branches des classes suivantes :

* Panel (en haut à droite) : contrôles permettent de définir la disposition (le layout) des autres contrôles dans la page, c’est-à-dire leurs tailles et positions relatives ou absolues.
* ContentControl : contrôle de contenu, pouvant contenir un seul enfant
* ItemsControl : contrôles pouvant afficher des listes d’objets

### Propriétés communes

Les dimensions

FrameworkElement fournit les propriétés suivantes pour le dimensionnement d’un élément :

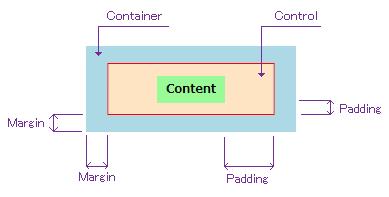
|  |  |
| --- | --- |
| **Propriété** | **Description** |
| ActualHeight / ActualWidth | Hauteur / largeur courante en exécution. En lecture seule |
| Height / Width | Hauteur / Largeur |
| MaxHeight / MaxWidth | Hauteur / Largeur maximale |
| MinHeight / MinWidth | Hauteur / Largeur minimale |

NB/ Les dimensions sont exprimées en DIP (Device Independant Pixel). Cette unité permet de définir les dimensions apparentes, indépendamment de la résolution de l’écran. Elle est reliée au nombre de pixels par la formule suivante : DIPs = pixels / (DPI/96)  
… où DPI (Dot Per Inch) est le nombre de points par pouces défini dans les réglages de l’écran.

Les marges

On distingue 2 types de marges sur un élément WPF :

* Les marges extérieures (Margin), entre les bords de l’élément et de son conteneur
* Les marges intérieures (Padding), entre les bords et le contenu de l’élément



Les valeurs des propriétés Margin et Paddings sont spécifiées par une suite de 4 entiers, séparés par des virgules, et désignant respectivement les marges gauche, supérieure, droite et inférieure. Ex :

<Button Content="Annuler" Margin="10,0,20,10"/>

Si on veut que les marges gauche et droite d’une part, et supérieure et inférieure d’autre part, soient identiques, on peut se contenter de spécifier seulement 2 nombres :

<Button Content="Annuler" Margin="10,20"/>

Si on veut que les 4 marges soient identiques, on peut spécifier un seul nombre

NB/ Padding n’est pas une propriété de FrameworkElement, mais de ContentControl. Elles ne s’appliquent qu’aux contrôles qui peuvent avoir un contenu.

L’alignement

Les propriétés HorizontalAlignment et VerticalAlignment définissent les alignements horizontal et vertical de l’élément lorsqu’il est à l’intérieur d’un élément parent. Elles peuvent prendre les valeurs suivantes :  
Center, Left, Right, Stretch

Les contrôles qui peuvent posséder un contenu ont 2 propriétés supplémentaires, qui définissent l’alignement de leur contenu : HorizontalContentAlignment et VerticalContentAlignment

/!\ Il y a des interactions entre les propriétés de dimensions et d’alignement, et le type de conteneur qui contient l’élément.

## La disposition (layout)

Les contrôles de disposition sont des conteneurs qui permettent d’agencer les autres contrôles à l’intérieur d’une fenêtre. Ils peuvent contenir plusieurs contrôles enfants.

La fenêtre contient toujours un unique conteneur principal, qui peut lui-même contenir d’autres conteneurs.

Nous allons voir dans ce chapitre :

* Les principaux conteneurs dérivés de la classe Panel (Canvas, Grid, StackPanel, DockPanel)
* D’autres conteneurs dérivés de la classe ContentControl, tels que ScrollViewer, GroupBox, Expander et TabItem
* Le contrôle Border

Il s’agit de présenter les contrôles les plus utilisés pour la disposition des éléments d’une fenêtre, et non pas la liste exhaustive.

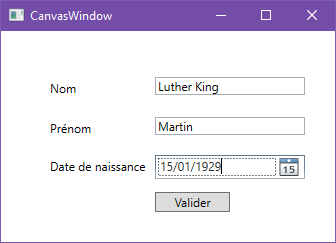
### Canvas

Dans ce type de conteneur, les contrôles sont placés et dimensionnés de façon absolue, c’est-à-dire qu’on attribue à chaque contrôle :

* Des coordonnées par rapport aux bords gauche et supérieur du canevas (propriétés attachées Canvas.Left et Canvas.Top)
* Une largeur et une hauteur (propriétés Width et Height) si nécessaire

En mode design, la barre d’outils Disposition (Layout) offre les commandes nécessaires pour aligner et dimensionner facilement les contrôles, sans avoir à saisir toutes les valeurs numériques à la main.

Exemple :



L’intérieur de cette fenêtre peut être réalisé avec le code suivant :

<Canvas>

<Label Content="Nom" Canvas.Left="44" Canvas.Top="44"/>

<TextBox TextWrapping="Wrap" Canvas.Left="154" Canvas.Top="46" Width="150"/>

<Label Content="Prénom" Canvas.Left="44" Canvas.Top="84"/>

<TextBox Canvas.Left="154" Canvas.Top="86" Width="150"/>

<Label Content="Date de naissance" Canvas.Left="44" Canvas.Top="122"/>

<DatePicker Canvas.Left="154" Canvas.Top="124" Width="150"/>

<Button Content="Valider" Canvas.Left="154" Canvas.Top="161" Width="75"/>

</Canvas>

**Propriétés attachées**

Les propriétés Canvas.Left et Canvas.Top qui sont affectées à chaque élément ne sont pas celles de l’élément, mais bien celles du canevas, c’est pourquoi on parle de propriétés **attachées**. C’est une particularité importante du xaml, qu’on ne retrouve pas dans le langage xml classique.

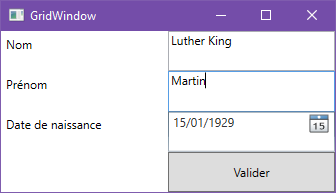
Le canevas permet donc de positionner les éléments de façon très simple à l’aide de propriétés attachées. Mais l’interface obtenue est rigide. Il n’y a par exemple pas de repositionnement, ni de redimensionnement des contrôles selon la taille de la fenêtre. Le canevas ne permet donc pas d’exploiter l’aspect dynamique de WPF, c’est pourquoi on ne l’utilise généralement pas.

### Grid

La grille est un conteneur riche et très largement utilisé en WPF. Elle permet de placer des contrôles dans des cases, repérées par des numéros de lignes et de colonnes.

La hauteur et la largeur des lignes et des colonnes peuvent s’adapter dynamiquement aux dimensions de la fenêtre. C’est pourquoi la grille est très souvent utilisée comme conteneur principal, afin d’obtenir un visuel dynamique.

Voici un exemple de visuel obtenu…



… avec le code suivant :

<Grid>

<!-- Création de 4 lignes -->

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<!-- Création de 2 colonnes -->

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<!-- Ligne 0 -->

<Label Content="Nom"/>

<TextBox Grid.Column="1"/>

<!-- Ligne 1 -->

<Label Content="Prénom" Grid.Row="1"/>

<TextBox Grid.Row="1" Grid.Column="1"/>

<!-- Ligne 2 -->

<Label Content="Date de naissance" Grid.Row="2"/>

<DatePicker Grid.Row="2" Grid.Column="1"/>

<!-- Ligne 3 -->

<Button Content="Valider" Grid.Row="3" Grid.Column="1"/>

</Grid>

Dans ce code :

* On commence par définir les lignes et colonnes de la grille
* On place ensuite les contrôles dans la grille à l’aide des propriétés attachées Grid.Row et Grid.Column. On n’a pas besoin de spécifier leurs valeurs pour la première ligne et la première colonne, car la valeur par défaut vaut 0.

Ce code et son rendu visuel illustrent plusieurs caractéristiques essentielles de la grille :

#### Caractéristiques principales :

* La hauteur des lignes et la largeur des colonnes s’adaptent dynamiquement aux dimensions de la fenêtre, sauf si on spécifie des valeurs fixes. Par défaut, toutes les lignes ont la même hauteur, et toutes les colonnes ont la même largeur.
* Un contrôle à l’intérieur d’une cellule occupe par défaut toute la place qui lui est offerte, c’est-à-dire la cellule entière, sauf si on spécifie ses dimensions.  
  NB/ Pour les Labels, ce n’est pas visible car leurs bordures ne sont pas visibles.
* La numérotation des lignes et colonnes commence à 0 (valeur par défaut)
* Un contrôle peut s’étendre sur plusieurs lignes et plusieurs colonnes si besoin (cf. plus bas)

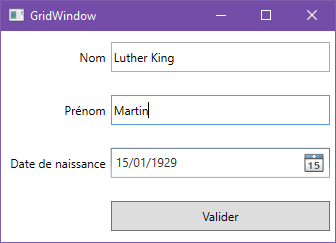
Le visuel obtenu précédemment est dynamique, c’est-à-dire qu’en exécution, les dimensions des contrôles s’adaptent à la taille de la fenêtre pour occuper tout l’espace disponible.

#### Positionnement des contrôles dans la grille

Pour améliorer le visuel précédent, on peut jouer en particulier sur les éléments suivants :

* La hauteur des contrôles
* La hauteur des lignes et la largeur des colonnes de la grille
* L’alignement horizontal et vertical de chaque contrôle à l’intérieur de sa cellule
* L’alignement horizontal et vertical du contenu de chaque contrôle
* Les marges autour de chaque contrôle.

Voici le visuel obtenu…



...en complétant le code précédent de cette façon :

<Grid Margin="0,0,5,0">

<!-- Création de 4 lignes -->

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<!-- Création de 2 colonnes -->

<Grid.ColumnDefinitions>

<ColumnDefinition Width="1\*"/>

<ColumnDefinition Width="2\*"/>

</Grid.ColumnDefinitions>

<!-- Ligne 0 -->

<Label Content="Nom" VerticalContentAlignment="Center"

HorizontalContentAlignment="Right"/>

<TextBox Grid.Column="1" Height="30" VerticalContentAlignment="Center"/>

<!-- Ligne 1 -->

<Label Content="Prénom" Grid.Row="1" VerticalContentAlignment="Center"

HorizontalContentAlignment="Right"/>

<TextBox Grid.Row="1" Grid.Column="1" Height="30" VerticalContentAlignment="Center"/>

<!-- Ligne 2 -->

<Label Content="Date de naissance" Grid.Row="2" VerticalContentAlignment="Center"

HorizontalContentAlignment="Right"/>

<DatePicker Grid.Row="2" Grid.Column="1" Height="30" VerticalContentAlignment="Center"/>

<!-- Ligne 3 -->

<Button Content="Valider" Grid.Row="3" Grid.Column="1" Height="30"/>

</Grid>

Dans ce code, on a :

* Spécifié une hauteur fixe pour les zones de saisies
* Centré verticalement le contenu des libellés et zones de texte
* Centré à droite le texte de chaque libellé, pour qu’il reste proche de la zone de saisie associée, notamment lorsqu’on élargit la fenêtre
* Spécifié des largeurs de colonnes relatives de 1 et 2, et extensibles (grâce à l’astérisque)
* Spécifié une marge droite de 5 sur le bord droit de la grille elle-même, afin qu’elle ne soit pas complètement collée au bord de la fenêtre

Le visuel obtenu est cette fois à la fois dynamique et esthétique.

**Taille des lignes et colonnes**

Revenons plus en détail sur la façon de spécifier les largeurs de colonnes et hauteur de lignes de la grille.

<ColumnDefinition Width="2\*"/> permet à la colonne d’adapter dynamiquement sa largeur (grâce à \*), tout en gardant une largeur relative double par rapport aux autres colonnes. Ce sont les valeurs relatives des largeurs des colonnes qui comptent, et non les valeurs absolues. On aurait ainsi pu obtenir le même résultat en spécifiant des largeurs de « 33\* » et « 66\* » par exemple.

Il y a en fait 4 façons de définir la largeur d'une colonne (respectivement la hauteur d'une ligne) :

* **Auto** : la largeur s'adapte automatiquement aux contrôles contenus dans la colonne
* **[Valeur]** : valeur absolue en dip
* **[Valeur]\*** : la largeur s'adapte proportionnellement à la valeur spécifiée. Lorsque plusieurs colonnes ont une largeur définie de la sorte, c'est le ratio entre les valeurs qui compte et non les valeurs absolues.
* **\*** : la colonne prend tout le reste de la place disponible en largeur. Si plusieurs colonnes ont une largeur \*, elles se partagent équitablement la largeur disponible.

**Positionnement d’un contrôle sur plusieurs lignes ou plusieurs colonnes**

Un contrôle peut être positionné sur plusieurs lignes et plusieurs colonnes à la fois. Pour cela, on utilise les propriétés attachées **Grid.RowSpan** et **Grid.ColumSpan**.

En reprenant l’exemple précédent, et en modifiant le code du bouton comme ceci :

<Button Content=**"Valider"** Grid.Row=**"3"** Grid.ColumnSpan=**"2"** Height=**"30"**/>

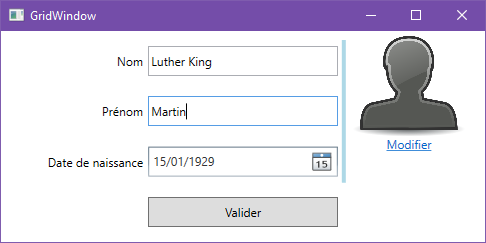
… le bouton s’étendra sur les 2 colonnes de la grille.

#### GridSplitter

Le GridSplitter (séparateur de grille) peut être associé à une grille pour permettre à l’utilisateur de modifier lui-même les largeurs relatives de 2 colonnes, ou les hauteurs relatives de 2 lignes, sans modifier les dimensions de la fenêtre.

Il se présente comme une barre que l’on fait glisser avec la souris.

Dans l’exemple ci-dessous, le GridSplitter est représenté par la barre verticale en bleu clair :



Examinons le code correspondant :

<Grid Margin="5">

<!-- Création de 4 lignes -->

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<!-- Création de 3 colonnes -->

<Grid.ColumnDefinitions>

<ColumnDefinition Width="3\*"/>

<ColumnDefinition Width="4\*"/>

<ColumnDefinition Width="3\*"/>

</Grid.ColumnDefinitions>

<GridSplitter Grid.Column="2" Grid.RowSpan="3" HorizontalAlignment="Left"

Width="4" Margin="4" Background="LightBlue"/>

<!-- Ligne 0 -->

...

<Image Grid.Column="2" Grid.RowSpan="2"

Source="https://cdn.pixabay.com/photo/2016/03/31/17/33/account-1293744\_960\_720.png" />

<!-- Ligne 1 -->

...

<!-- Ligne 2 -->

...

<TextBlock Grid.Row="2" Grid.Column="2" HorizontalAlignment="Center">

<Hyperlink>Modifier</Hyperlink>

</TextBlock>

<!-- Ligne 3 -->

...

</Grid>

Par rapport à la version précédente, nous avons ajouté une nouvelle colonne contenant :

* Une image qui s’étend sur les 2 premières lignes
* Un lien hypertexte
* Un séparateur de grille vertical, aligné à gauche, et qui s’étend sur les 3 premières lignes. Nous lui avons spécifié une largeur, une marge et une couleur afin qu’il soit plus visible :

<GridSplitter Grid.Column="2" Grid.RowSpan="3" HorizontalAlignment="Left"  
 Width="4" Margin="4" Background="LightBlue"/>

Le GridSplitter peut être ajouté indifféremment sur l’une des 2 colonnes ou lignes séparées. Il est par défaut aligné verticalement à droite et horizontalement au centre.

Pour créer un séparateur horizontal, on utilisera un code semblable à celui-ci :

<GridSplitter Grid.Row="3" Grid.ColumnSpan="2" Height="5"

ResizeDirection="Rows" HorizontalAlignment="Stretch" VerticalAlignment="Top"/>

### StackPanel

Le StackPanel (qu’on pourrait traduire par « panneau d’empilage ») est un conteneur assez basique, qui permet d’empiler des contrôles les uns à la suite des autres, soit horizontalement, soit verticalement, sans chevauchement possible.

**Caractéristiques principales :**

* La propriété **Orientation** permet de définir si le contenu du StackPanel est empilé verticalement ou horizontalement
* Si l’orientation est verticale, chaque contrôle placé à l’intérieur du StackPanel occupe par défaut :
  + Le minimum de hauteur. Celle-ci est fixée par le contenu du contrôle (ex : la hauteur du texte d’une TextBox)
  + Toute la largeur du StackPanel
* Si l’orientation du StackPanel est horizontale, c’est l’inverse : chaque contrôle occupe le minimum de place en largeur, et toute la hauteur du StackPanel
* Le contenu qui dépasse la largeur ou la hauteur du StackPanel est tronqué

Ces caractéristiques sont illustrées par l’exemple suivant :

|  |  |
| --- | --- |
|  | Les StackPanel ont un fond coloré pour mieux distinguer leur étendue.  StackPanel vertical  StackPanel horizontal contenant 3 StackPanel verticaux  StackPanel horizontal avec alignement à droite |

Code correspondant :

<Grid Margin=**"10"**>

<Grid.RowDefinitions>

<RowDefinition Height=**"20\*"**/>

<RowDefinition Height=**"10\*"**/>

<RowDefinition Height=**"50\*"**/>

<RowDefinition Height=**"10\*"**/>

<RowDefinition Height=**"10\*"**/>

</Grid.RowDefinitions>

<!-- Ligne 0 : nom et prénom -->

<StackPanel Orientation=**"Vertical"** Background=**"#FFC8C8FF"**>

<TextBox Text=**"Prénom"** Height=**"25"** Margin=**"0,5,0,5"**/>

<TextBox Text=**"Email"** Height=**"25"**/>

</StackPanel>

<!-- Ligne 1 : libellé -->

<TextBlock Grid.Row=**"1"** VerticalAlignment=**"Bottom"** Margin=**"0,0,0,5"**>**Quels sont vos centres d'intérêt ?**</TextBlock>

<!-- Ligne 2 : cases à cocher -->

<StackPanel Orientation=**"Horizontal"** Grid.Row=**"2"**>

<StackPanel Orientation=**"Vertical"** Background=**"#FFFFC8C8"**>

<CheckBox Content=**"Développement Web"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk21"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk31"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk41"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk51"**/>

</StackPanel>

<StackPanel Orientation=**"Vertical"** Background=**"#FFC8FFC8"**>

<CheckBox Content=**"cbk12"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk22"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk32"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk42"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk52"** Margin=**"0,0,5,5"**/>

</StackPanel>

<StackPanel Orientation=**"Vertical"** Background=**"#FF96C8FF"**>

<CheckBox Content=**"cbk13"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk23"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk33"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"cbk43"** Margin=**"0,0,5,5"**/>

<CheckBox Content=**"Développement mobile"** Margin=**"0,0,5,5"**/>

</StackPanel>

</StackPanel>

<!-- Ligne 3 : cases à cocher avec texte long -->

<CheckBox Grid.Row=**"3"**/>

<TextBlock Grid.Row=**"3"** Margin=**"20,0,0,0"** TextWrapping=**"Wrap"**

Text=**"J'autorise ... à m'envoyer des offres commerciales de ses partenaires"**/>

<!-- Ligne 4 : boutons OK et Annuler-->

<StackPanel Orientation=**"Horizontal"** Grid.Row=**"4"** HorizontalAlignment=**"Right"** VerticalAlignment=**"Bottom"** Background=**"#FFF3F319"**>

<Button Content=**"OK"** Width=**"100"**/>

<Button Content=**"Annuler"** Width=**"100"** Margin=**"10,0,0,0"**/>

</StackPanel>

</Grid>

La grille principale ne comporte que 4 lignes.

Les StackPanel verticaux qui contiennent les cases à cocher évitent d’avoir à créer une multitude de lignes de grille. On pourra ainsi rajouter facilement de nouvelles cases à cocher si besoin.

On voit que la largeur de ces panneaux s’adapte à leur contenu. Comme la largeur de la fenêtre est trop juste, on voit que le texte de la dernière case à cocher est tronqué.

A l’intérieur des StackPanel, l’application de marges sur les différents contrôles permet de les espacer légèrement.

Dans la ligne 3, comme le texte associé à la case à cocher est long, et que le contrôle CheckBox ne permet pas le retour à la ligne, on a utilisé un TextBlock.

**Bonnes pratiques**

Le StackPanel est beaucoup utiliser pour regrouper des contrôles qui vont ensemble, afin de les manipuler plus facilement.

Comme dans l’exemple ci-dessus, il peut être utilisé pour mettre plusieurs éléments dans une seule cellule de grille, sans qu’ils ne se superposent, et ainsi éviter d’avoir à créer une grille très fine (avec beaucoup de lignes et de colonnes). En effet, plus le nombre de lignes et de colonnes d’une grille est élevé, plus celle-ci est difficile à maintenir.

**WrapPanel**

Le WrapPanel est identique au StackPanel, hormis le fait que son contenu n’est pas tronqué, mais passe sur une nouvelle ligne, s’il excède la largeur du panneau. Cette caractéristique peut être utilisée par exemple dans une barre d’outils, pour que le boutons restent visibles si on réduit la largeur de la barre.

### DockPanel

Le DockPanel (qu’on pourrait traduire par « panneau d’ancrage ») maintient son contenu ancré sur ses bords, même si la taille du DockPanel varie.

**Caractéristiques principales :**

* La propriété attachée **Dock** permet de définir sur quel bord du panneau est ancré le contrôle
* L’ordre dans lequel sont ajoutés les contrôles est important, comme le montre l’exemple qui suit
* Par défaut, le dernier contrôle ajouté remplit le reste de l’espace disponible dans le panneau

Exemple :

|  |  |
| --- | --- |
|  | <DockPanel>  <TextBlock Text="Bloc 1"  DockPanel.Dock="Top"  Background="#FFC8C8FF" Padding="5"/>  <TextBlock Text="Bloc 2"  DockPanel.Dock="Left"  Background="#FFFFC8C8" Padding="5"/>  <TextBlock Text="Bloc 3"  DockPanel.Dock="Bottom"  Background="#FFC8FFC8" Padding="5"/>  <TextBlock Text="Bloc 4"  DockPanel.Dock="Right"  Background="#FF96C8FF" Padding="5"/>  <TextBlock Text="Bloc 5" Padding="5"/>  </DockPanel> |

**Bonnes pratiques**

Le DockPanel n’est en général pas utilisé comme seul conteneur. Typiquement, il est utilisé pour ancrer des composants tels que des menus, barres d’outils ou composants de sélection, sur les bords de la fenêtre. Au centre, on place une grille.

On peut aussi profiter du fait que le dernier composant utilise toute la place disponible, pour réaliser certaines dispositions qu'on ne peut pas faire avec un StackPanel.

L'exemple suivant affiche un libellé, une zone de saisie et une case à cocher sur la même ligne. La case à cocher est alignée à droite, ce qu'on ne peut pas faire avec un StackPanel horizontal :



<DockPanel>

<Label Content="Priorité"/>

<TextBox Text="1" Width="20"

HorizontalContentAlignment="Center"

VerticalContentAlignment="Center"/>

<CheckBox Content="Fait" HorizontalAlignment="Right"

VerticalContentAlignment="Center"/>

</DockPanel>

Cette technique est une bonne alternative à la grille pour placer 2 ou 3 composants sur une même ligne, si on cherche à occuper toute la largeur disponible, sans avoir une répartition régulière des composants. L’utilisation du DockPanel permet d’avoir un code xaml très concis.

NB/ Utiliser les propriétés Padding et Margin pour que ce soit moins tassé.

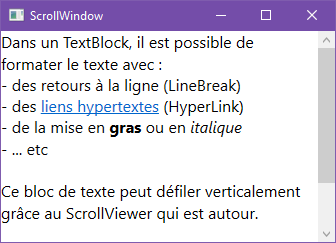
### ScrollViewer

Le fait de placer un contrôle à l’intérieur d’un ScrollViewer, permet d’afficher des barres de défilement sur ce contrôle, si celui-ci le permet. Il n’est pas nécessaire d’ajouter soi-même un contrôle ScrollBar.

La barre de défilement n’est activée que si c’est nécessaire, c’est-à-dire que si le contenu auquel elle s’applique n’est pas entièrement visible.

Si le ScrollViewer est le premier contrôle enfant de la fenêtre, la barre de défilement porte logiquement sur l’ensemble du contenu de la fenêtre.

Dans l’exemple ci-dessous, le ScrollViewer entoure un bloc de texte. Ceci permet d’afficher automatiquement une barre de défilement verticale sur le côté du bloc de texte.



<ScrollViewer>

<TextBlock TextWrapping="Wrap" FontSize="16">

Dans un TextBlock, il est possible de formater le texte avec :<LineBreak/>

- des retours à la ligne (LineBreak)<LineBreak/>

- des <Hyperlink>liens hypertextes</Hyperlink> (HyperLink)<LineBreak/>

- de la mise en <Bold>gras</Bold> ou en <Italic>italique</Italic><LineBreak/>

- ... etc

<LineBreak/><LineBreak/>

Ce bloc de texte peut défiler verticalement grâce au ScrollViewer qui est autour.

<LineBreak/><LineBreak/>

Jouez avec la taille de la fenêtre pour observer le comportement du texte et de la barre de défilement.

</TextBlock>

</ScrollViewer>

### GroupBox, Expander et TabControl/TabItem

Ces contrôles servent surtout à regrouper visuellement des éléments.

Comme ils ne peuvent contenir qu’un seul élément enfant, on y place généralement un StackPanel ou un Grid.

La contrôle **GroupBox** affiche un titre et un encadré. On y place généralement des boutons radio et des cases à cocher.

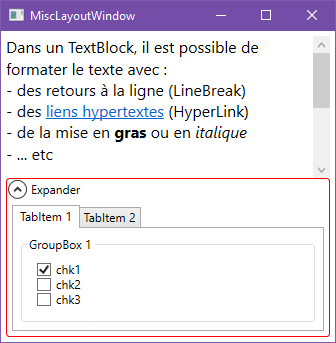
Le contrôle **Expander** est semblable au GroupBox, mais il permet en plus de se replier ou de se déplier, pour masquer ou afficher son contenu. Il a un en-tête et un contenu. Lorsqu’il est replié, seul l’en-tête est visible

Caractéristiques :

* La propriété IsExpanded définit si l’Expander est replié ou déplié
* La propriété ExpandDirection définit dans quelle direction se déplie l’Expander (bas, haut, gauche ou droite)
* Par défaut, sa bordure est invisible, mais on peut la rendre visible ne spécifiant une épaisseur et une couleur.

Le **TabControl** est un conteneur d’onglets. Les onglets partagent un même espace à l’écran, optimisant ainsi grandement cet espace. On ne peut voir le contenu que d’un seul onglet à la fois.  
Le TabControl gère le clic sur les onglets, pour afficher le contenu de l’onglet courant et masquer celui des autres onglets. Il peut contenir une collection d’objets de n’importe quel type (texte, image, panneau…), mais le type le plus approprié est le **TabItem**, qui permet d’afficher un titre d’onglet et un contenu.

Ces contrôles sont illustrés par l’exemple suivant :



Code correspondant :

<Grid Margin="5">

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<!-- 1ère ligne -->

<ScrollViewer>

<TextBlock TextWrapping="Wrap" FontSize="16">

...

</TextBlock>

</ScrollViewer>

<!-- 2de ligne -->

<Expander Header="Expander" Grid.Row="1" IsExpanded="True"

BorderThickness="1" BorderBrush="Red">

<TabControl Margin="5">

<!-- Onglet 1 -->

<TabItem Header="TabItem 1">

<GroupBox Header="GroupBox 1" Margin="5" Padding="10">

<StackPanel>

<CheckBox Content="chk1" IsChecked="True"/>

<CheckBox Content="chk2"/>

<CheckBox Content="chk3"/>

</StackPanel>

</GroupBox>

</TabItem>

<!-- Onglet 2 -->

<TabItem Header="TabItem 2">

<GroupBox Header="GroupBox 2" Margin="5" Padding="10" BorderBrush="Blue">

<StackPanel Orientation="Horizontal">

<RadioButton Content="opt1" Margin="0,0,20,0"/>

<RadioButton Content="opt2" Margin="0,0,20,0" IsChecked="True"/>

<RadioButton Content="opt3" Margin="0,0,20,0"/>

</StackPanel>

</GroupBox>

</TabItem>

</TabControl>

</Expander>

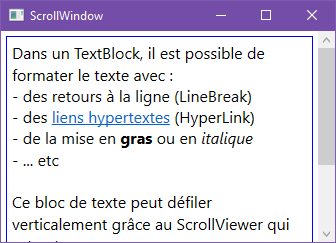
</Grid>

### Border

Le contrôle **Border** affiche simplement une bordure avec l’épaisseur et la couleur souhaitée. Il est utile pour encadrer les contrôles qui n’ont pas de bordure, comme par exemple le TextBlock.

On peut le comparer à un GroupBox sans titre.

Ci-dessous, on ajoute une bordure bleue autour du bloc de texte de l’exemple précédent :



<ScrollViewer>

<Border BorderBrush="Blue" BorderThickness="1" Margin="5" Padding="5">

<TextBlock TextWrapping="Wrap" FontSize="16">

...

</TextBlock>

</Border>

</ScrollViewer>

### Recommandations

* Il faut essayer de spécifier le moins possible de tailles fixes pour les contrôles, surtout en largeur, de façon à garder un visuel dynamique et un code facilement modifiable.
* Pour des visuels riches (avec de nombreux contrôles), il est bon de regrouper les contrôles dans des conteneurs intermédiaires (Grid ou StackPanel), avant de les placer dans la grille principale, de façon à pouvoir les traiter par paquets. C’est plus productif et plus évolutif que d’utiliser uniquement la grille principale.
* De nombreux niveaux d’imbrications de conteneurs rendent toutefois le code plus difficile à lire et peuvent dégrader les performances. Il ne faut donc pas en abuser.
* Pour obtenir un visuel harmonieux :
  + Aligner au maximum les contrôles les uns avec les autres
  + Les répartir de façon homogène (éviter les espaces vides trop importants)
  + Optimiser la place : dimensionner les fenêtres au plus juste, éviter les marges trop importantes (penser aux écrans d’ordinateurs portables qui ont parfois une résolution assez faible) …
* Mettre des commentaires dans le xaml pour s’y retrouver plus facilement, notamment avant les conteneurs principaux, les lignes de grilles, les blocs d’informations...etc.

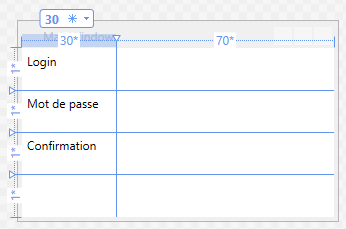
### Le designer de Visual Studio

Lorsqu’on crée une nouvelle fenêtre, Visual Studio génère automatiquement le code XAML pour la fenêtre et son conteneur principal, de type Grid.

On peut définir les lignes et les colonnes de la grille en cliquant sur les bords de celle-ci. Puis on peut ajouter les contrôles dans les cellules de la grille, en les faisant glisser à partir de la boîte à outils du designer.

Cependant, si on se contente de faire cela, les contrôles sont dimensionnés et positionnés de façon statique à l’intérieur des cellules à l’aide de leurs propriétés Width, Height et Margin, ce qui n’est généralement pas ce qu’on souhaite.

Pour créer un visuel dynamique, on précédera de la façon suivante :

* Créer les lignes et colonnes de la grille en cliquant sur sa bordure
* Définir leurs tailles relatives au moyen des petites listes déroulantes qui apparaissent au survol avec la souris :  
    
  A ce stade, le xaml contient les définitions de lignes et de colonnes
* Placer tous les contrôles dans les cellules de la grille, sans se préoccuper de leur taille
* Sélectionner tous les contrôles, et faire un clic droit sur l’un d’entre eux, puis cliquer sur le menu contextuel « **Layout \ Reset all**». Ceci a pour effet de supprimer tous les formatages inutiles, notamment les tailles et positions des contrôles. A ce stade, on a déjà un visuel dynamique et un code propre
* Peaufiner ensuite le visuel en spécifiant les alignements, tailles et marges pour les contrôles qui le nécessitent, grâce au panneau de propriétés. On peut spécifier des valeurs de propriétés sur plusieurs contrôles en même temps, pour aller plus vite.

L’utilisation du designer a ses limites ; il est généralement compliqué d’ajouter des éléments à un visuel existant, sans impacter de nombreux éléments.

L’écriture directe du code xaml permet en général d’aller plus directement au résultat souhaité, et souvent de façon bien plus rapide qu’avec le designer. Il est de toute façon nécessaire de maîtriser le code xaml pour pouvoir l’ajuster et le compléter.

De plus, le designer ne permettra pas d’afficher correctement les composants externes (non fournis par Visual Studio), si cela n’a pas été prévu par l’éditeur.

## Le contenu

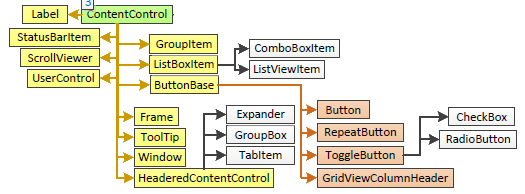
Le terme « contenu » est ici à prendre au sens large. Les contrôles de contenu sont ceux qui ne peuvent contenir qu’**un seul élément enfant**, de n’importe quel type. Cela englobe des contrôles très variés : les fenêtres, les libellés, les différents types de boutons, les contrôles représentant des items de collection…etc.

Ces contrôles dérivent de la classe ContentControl.

NB/ Attention à ne pas confondre contenu et conteneur. Les contrôles que l’on nomme « conteneurs » sont généralement les contrôles de disposition (Grid, StackPanel…etc).

### La classe ContentControl

Cette classe est l’ancêtre des contrôles de contenu. Voici sa hiérarchie d’héritage :



La particularité de la classe ContentControl est de posséder une propriété **Content** de type Object, qui représente le contenu du contrôle. Il y a 2 façons de spécifier la valeur de cette propriété en XAML :

<Button Content="Valider"/>

…équivaut à :

<Button>Valider</Button>

Un ContentControl peut contenir et afficher n’importe quel type CLR (comme string ou DateTime par exemple) ou n’importe quel type dérivé de UIElement, ce qui laisse énormément de possibilités.

Ceci est illustré par l’exemple de code suivant :

<!-- Contenu de type String -->

<Label>Texte simple</Label>

<!-- Contenu de type DateTime-->

<Label xmlns:sys="clr-namespace:System;assembly=mscorlib" ContentStringFormat="D">

<sys:DateTime>2016-12-25 13:06:55</sys:DateTime>

</Label>

<!--Contenu de type UIElement -->

<Label>

<Border BorderBrush="Red" BorderThickness="1" Padding="5" CornerRadius="10">

<StackPanel Orientation="Horizontal">

<Ellipse Height="20" Width="20" Fill="Blue" Margin="0,0,5,0"/>

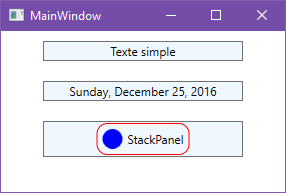
<TextBlock VerticalAlignment="Center">StackPanel</TextBlock>

</StackPanel>

</Border>

</Label>

…qui donne le visuel suivant (certains styles supplémentaires ont été appliqués) :



La propriété **ContentTemplate** est également l’une des plus intéressantes. Elle permet de spécifier le modèle (DataTemplate) à utiliser pour afficher son contenu. Cette fonctionnalité est utilisée conjointement avec la liaison de données, que nous verrons plus loin.

### Quelques exemples de ContentControl

**ContentControl** n’est pas une classe abstraite ; on peut donc utiliser cet élément directement. Il est quasiment identique au Label.

**ButtonBase** est la classe abstraite ancêtre des boutons, notamment Button et ToggleButton. Elle définit l’évènement click, ainsi que la propriété Command, qui permet d’invoquer une commande (cf. + loin) lorsque le bouton est pressé

**CheckBox et RadioButton**dérivent de ToggleButton, qui possède 2 propriétés propres : IsChecked (état coché ou non) et IsThreeState (possibilité ou non d’avoir un état intermédiaire).

**Tooltip** (info-bulle) se présente sous forme d’une popup permettant de fournir des informations à l’utilisateur.  
Le Tooltip ne peut pas avoir de parent, mais peut être assigné à la propriété Tooltip d’un autre contrôle. La popup apparaît généralement au survol de ce contrôle par la souris.

<CheckBox Content="Imprimer les dessins" IsChecked="True"

ToolTip="Désactiver cette option pour imprimer les documents sans graphiques ni zones de texte flottantes"/>

<TextBox HorizontalAlignment="Left">ToolTip contenant une image et du texte

<TextBox.ToolTip>

<ToolTip>

<DockPanel Width="50" Height="70">

<Image Source="data\flower.jpg"/>

<TextBlock>Information texte</TextBlock>

</DockPanel>

</ToolTip>

</TextBox.ToolTip>

</TextBox>

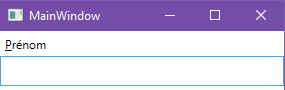
Un Tooltip peut être défini en tant que ressource (cf. plus bas) afin d’être utilisable sur plusieurs contrôles.

**Label** (étiquette) est fait pour être associé à un autre contrôle. Il a une propriété Target pour indiquer le nom de l’élément qui reçoit le focus lorsqu’on utilise l’accès rapide du Label au moyen de ALT + <touche>.

La touche d’accès rapide est définie en mettant un « \_ » devant la lettre souhaitée du texte du Label. Ex :

<Label Content="\_Prénom" Target="{Binding ElementName=txtPrenom}"/>

Ici, lorsqu’on appuie sur ALT + P, la zone de texte txtPrenom reçoit automatiquement le focus :



**Différences entre Label et TextBlock :**

Label hérite de ContentControl et donc de Control. Il a donc des propriétés HorizontalContentAlignment et VerticalContentAlignment, IsTabStop, TabIndex, Background, Foreground, BorderBrush, que ne possède pas le TextBlock.

TextBlock hérite directement de FrameworkElement, et est spécialisé dans l’affichage de texte. Il a des propriétés propres pour cela :

* IsHyphenationEnabled (coupure de mots automatique)
* LineHeight (hauteur des lignes de texte)
* LineStackingStrategy
* Text
* TextAlignment
* TextDecorations
* TextEffects
* TextTrimming
* TextWrapping (retour à la ligne)
* Typography

Il peut contenir plusieurs éléments enfants, ce qui permet notamment d’imbriquer d’autres TextBlock, et de mettre en forme le texte avec les éléments Bold, Italic, LineBreak…etc.

<TextBlock TextWrapping="Wrap" FontSize="14">

Dans un TextBlock, il est possible de formater le texte avec :<LineBreak/>

- des retours à la ligne (LineBreak)<LineBreak/>

- des <Hyperlink>liens hypertextes</Hyperlink> (HyperLink)<LineBreak/>

- de la mise en <Bold>gras</Bold> ou en <Italic>italique</Italic><LineBreak/>

- ... etc

</TextBlock>

## Les collections d’éléments

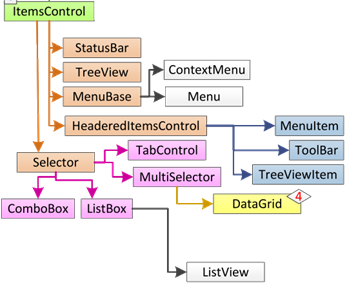
L’affichage de collections d’éléments concerne :

* Les menus (y compris contextuels)
* La barre de statut
* Les barres d’outils
* Les arbres
* Les onglets
* Les contrôles de sélection (listes, grille de données)

Ces contrôles dérivent de la classe [ItemsControl](https://msdn.microsoft.com/fr-fr/library/system.windows.controls.itemscontrol(v=vs.90).aspx)

### La classe ItemsControl

Cette classe est l’ancêtre des contrôles qui gèrent une collection d’éléments. Voici sa hiérarchie d’héritage :



**Contenu de la collection**

Pour chaque dérivé de ItemsControl, il existe une classe dédiée à la représentation de ses items :

|  |  |
| --- | --- |
| **ItemsControl** | **Classe qui représente ses items** |
| ComboBox | ComboBoxItem |
| ListBox | ListBoxItem |
| ListView | ListViewItem |
| StatusBar | StatusBarItem |
| Menu | MenuItem |
| TabControl | TabItem |
| TreeView | TreeViewItem |

…mais leur utilisation n’est pas obligatoire, car un ItemsControl peut contenir tout type d’objet (texte, image, case à cocher…), et les éléments d’une même collection ne sont pas forcément de même nature

Exemple avec le contrôle Menu :

|  |  |
| --- | --- |
|  | <Menu>  <MenuItem Header="\_Edition">  <MenuItem Command="ApplicationCommands.Copy"/>  <MenuItem Command="ApplicationCommands.Cut"/>  <MenuItem Command="ApplicationCommands.Paste"/>  </MenuItem>  <CheckBox Content="case à cocher"/>  <Button Content="Bouton"/>  <Image Source="Information.png" Height="20"/>  </Menu> |

Dans cet exemple, on utilise des MenuItem pour créer un menu déroulant avec des commandes associées. On a également ajouté d’autres items de différents types.

**Affichage des éléments**

Pour personnaliser l’affichage des éléments de la collection, on peut utiliser entre autres les propriétés suivantes de ItemsControl :

|  |  |  |
| --- | --- | --- |
| **Propriété** | **Type** | **Description** |
| ItemsPanel | ItemsPanelTemplate | Panneau qui contrôle la disposition des éléments |
| ItemStringFormat | String | Format d’affichage des éléments s'ils sont affichés en tant que chaînes. Cela peut être un format prédéfini (G, F, D, X…), ou un format personnalisé (ex : MMMM/yyyy) |

Ces deux propriétés sont illustrées dans l’exemple ci-dessous :

<ListBox SelectionMode="Multiple" ItemStringFormat="MMM">

<ListBox.ItemsPanel>

<ItemsPanelTemplate>

<StackPanel Orientation="Horizontal"

VerticalAlignment="Center" HorizontalAlignment="Center"/>

</ItemsPanelTemplate>

</ListBox.ItemsPanel>

<sys:DateTime>2017/1/31</sys:DateTime>

<sys:DateTime>2017/2/28</sys:DateTime>

<sys:DateTime>2017/3/31</sys:DateTime>

<sys:DateTime>2017/4/30</sys:DateTime>

<sys:DateTime>2017/5/31</sys:DateTime>

<sys:DateTime>2017/6/30</sys:DateTime>

<sys:DateTime>2017/7/31</sys:DateTime>

<sys:DateTime>2017/8/31</sys:DateTime>

<sys:DateTime>2017/9/30</sys:DateTime>

<sys:DateTime>2017/10/31</sys:DateTime>

<sys:DateTime>2017/11/30</sys:DateTime>

<sys:DateTime>2017/12/31</sys:DateTime>

</ListBox>



On a utilisé ici un StackPanel orienté horizontalement pour afficher les items. Ces derniers sont de type DateTime. On a défini le format « MM » pour n’afficher que le mois de façon abrégée.

**Liaison de données**

Un des principaux intérêts des ItemsControl est de pouvoir afficher des collections d’objets fournies dynamiquement par le code, au moyen de liaisons de données. Pour cela, ItemsControl possède des propriétés propres qui seront présentées dans le chapitre sur la liaison de données.

### Les contrôles de sélection

Un contrôle de sélection permet à l’utilisateur de sélectionner des éléments parmi ses éléments enfants.

Tous les contrôles de sélection dérivent de la classe abstraite **Selector** elle-même dérivée de ItemsControl.

Nous allons voir ici les propriétés principales de Selector, puis les propriétés spécifiques des contrôles ComboBox, ListBox et ListView. Ces contrôles prennent tout leur intérêt avec la liaison de données et les DataTemplate, que nous verrons dans les prochains chapitres.

NB/ Nous avons déjà vu le TabControl en tant que contrôle de disposition. Nous ne traiterons pas le DataGrid, qui est l’un des contrôles les plus complexe, et nécessiterait un chapitre entier.

#### Propriétés communes

Voyons les propriétés principales de la classe ancêtre Selector.

**Propriétés de sélection**

|  |  |  |
| --- | --- | --- |
| **Propriété** | **Type** | **Description** |
| SelectedIndex | int | Indice de l’item sélectionné |
| SelectedItem | object | Item sélectionné |
| SelectedValuePath | string | Nom de la propriété qui définit la valeur sélectionnée |
| SelectedValue | object | Valeur sélectionnée |
| DisplayMemberPath | string | Nom de la propriété qui définit ce qui est affiché par la liste |

Exemple :

<ComboBox SelectedValuePath="Tag">

<ComboBoxItem Tag="M" Content="Monsieur" IsSelected="True"/>

<ComboBoxItem Tag="F" Content="Madame"/>

</ComboBox>

Dans cet exemple :

* SélectedIndex = 0, car le premier ComboBoxItem a sa propriété IsSelected à vrai
* SelectedItem contient l’objet ComboBoxItem sélectionné
* SelectedValuePath a la valeur « Tag »
* SelectedValue a la valeur « M », qui est celle de la propriété Tag de l’item sélectionné

Un contrôle de sélection peut être lié à une collection d’objets. Supposons par exemple qu’il soit lié à des objets Personne ayant des propriétés Nom, Prénom et Sexe, et qu’on souhaite afficher uniquement le nom. Alors :

* Il faudra affecter la valeur « Nom » à SelectedValuePath.
* La propriété SelectedValue contiendra un objet de type Personne.

**Propriétés attachées d’affichage**

Lorsque les items ne sont pas du texte, on peut utiliser les propriétés attachées TextSearch.Text ou [TextSearch.TextPath](https://msdn.microsoft.com/fr-fr/library/system.windows.controls.textsearch.textpath(v=vs.90).aspx) pour les afficher sous forme de texte, comme illustré par l’exemple suivant :

|  |  |
| --- | --- |
|  | <ComboBox IsEditable="True" Width="100" FontSize="16">  <ComboBoxItem Content="&#x2660;" TextSearch.Text="Pic"/>  <ComboBoxItem Content="&#x2665;" TextSearch.Text="Coeur"/>  <ComboBoxItem Content="&#x2666;" TextSearch.Text="Carreau"/>  <ComboBoxItem Content="&#x2663;" TextSearch.Text="Trèfle"/>  </ComboBox> |

Cette ComboBox contient des items dont le contenu est un caractère Unicode, et est associé à une valeur textuelle grâce à la propriété attachée TextSearch.Text. On peut ainsi sélectionner un élément en tapant les premières lettres du texte associé dans la zone de saisie, et l’élément sélectionné s’affiche sous forme de texte dans cette zone.

#### ComboBox

La [ComboBox](https://msdn.microsoft.com/fr-fr/library/system.windows.controls.combobox(v=vs.90).aspx) permet à l'utilisateur de sélectionner un élément dans une liste déroulante ou d'entrer éventuellement le nouveau texte dans la zone de texte du contrôle.

Les propriétés **IsEditable** et **IsReadonly** déterminent ce qu’il est possible de faire en termes de saisie dans ce composant. Ceci est résumé par le tableau suivant :

|  |  |  |  |
| --- | --- | --- | --- |
|  | **IsEditable** | **IsReadonly** | |
| **True** | **False** |
| Sélectionner un élément en saisissant du texte | **True** | - | ✔ |
| **False** | ✔ | ✔ |
| Saisir du texte qui ne correspond à aucun élément | **True** | - | ✔ |
| **False** | - | - |
| Sélection d'une partie du texte dans la zone de texte | **True** | ✔ | ✔ |
| **False** | - | - |
| Copier/coller du texte | **True** | Copie | ✔ |
| **False** | - | - |

#### ListBox

La ListBox est une liste qui permet d’afficher plusieurs items à la fois.

La propriété **SelectionMode** détermine si on peut sélectionner un ou plusieurs items. Avec la valeur Extended, il est possible d’utiliser les touches SHIFT et CTRL pour sélectionner des plages d’items consécutives ou non.

#### ListView

La [ListView](https://msdn.microsoft.com/fr-fr/library/ms750972(v=vs.110).aspx) permet de visualiser une collection de données sous différents formes appelées « vues ». La vue est définie par la propriété **View**. WPF fournit une vue par défaut nommé **GridView**, qui correspond à un tableau ayant des colonnes personnalisables de type **GridViewColumn**.

L’exemple ci-dessous montre comment créer une ListView avec ce type de vue.

<ListView >

<ListView.View>

<GridView AllowsColumnReorder="true">

<GridViewColumn Header="Prénom" Width="100"/>

<GridViewColumn Width="150">

<GridViewColumnHeader>Nom

<GridViewColumnHeader.ContextMenu>

<ContextMenu>

<MenuItem Header="Trier en montant" />

<MenuItem Header="Trier en descendant" />

</ContextMenu>

</GridViewColumnHeader.ContextMenu>

</GridViewColumnHeader>

</GridViewColumn>

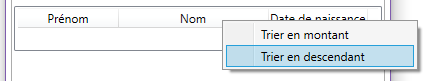
<GridViewColumn Header="Date de naissance"/>

</GridView>

</ListView.View>

</ListView>

Visuel correspondant :



Sur la GridView, AllowsColumnReorder="true" autorise l’utilisateur à réorganiser les colonnes dans l’ordre qu’il veut en les faisant glisser avec la souris.

Le GridViewColumnHeader représente l’en-tête de colonne. C’est un ContentControl, qui peut donc contenir un objet de n’importe quel type (Ici, nous n’avons mis que du texte). On peut lui associer un menu contextuel, comme le montre l’exemple.

On peut définir une vue personnalisée en dérivant la classe abstraite **ViewBase,** comme illustré dans [cette page MSDN](https://msdn.microsoft.com/fr-fr/library/ms748859(v=vs.110).aspx).

ListView est typiquement un contrôle qui s’utilise avec une liaison de données pour définir son contenu. Il n’a pas d’utilité sans cela.

# Les espaces de noms

## Déclaration des [espaces de noms](https://msdn.microsoft.com/fr-fr/library/ms747086(v=vs.110).aspx)

Quand on créer une nouvelle fenêtre, Visual Studio génère automatiquement un code qui ressemble à ceci :

1 <Window x:Class="MonAppli.MainWindow"

2 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

5 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

6 xmlns:local="clr-namespace:MonAppli"

7 mc:Ignorable="d"

8 Title="MainWindow" Height="300" Width="300">

xmlns signifie « XML Namespace », c’est-à-dire « **espace de noms XML** ». Il s’agit d’une notion similaire à celle des espaces de noms de la librairie de classes .net ; cela permet d’éviter les conflits de noms.

En XML, les espaces de noms sont identifiés par des url, et on y fait référence au moyen de **préfixes**, avec la syntaxe : <Préfixe>:<nom de l’élément>

Ainsi, la 3ème ligne déclare un espace de noms préfixé par x, et la 1ère ligne utilise l’attribut Class de cette espace de noms avec la syntaxe x:Class

La 2ème ligne déclare l’**espace de noms par défaut**, c’est-à-dire celui qui peut être utilisé sans préfixe. Tous les éléments WPF standards sont dans cet espace de noms, c’est pourquoi leurs noms ne sont pas préfixés dans le code xaml (et heureusement, car ce serait pénible à lire !).

On est ainsi en mesure de mieux comprendre le code généré :

* La ligne 1 déclare une fenêtre décrite par une classe nommée « MonAppli.MainWindow ». MonAppli est en fait le nom de l’espace de noms C# (par défaut identique au nom du projet). Cette première ligne fait donc le lien entre le code xaml et la classe C# de la fenêtre.
* La lignes 2 déclare l’espace de noms de tous les éléments WPF standards (appartenant à l’espace de noms System.Windows de la librairie de classes .net)
* La ligne 3 déclare l’espaces de noms du XAML (appartenant à l’espace de noms System.Windows.Markup de la librairie de classes .net).
* La ligne 4 déclare l’espace de noms du designer. En effet, celui-ci peut ajouter des attributs xaml pour ses propres besoins.
* Les lignes 5 et 7 (xmlns:mc=".." et mc:Ignorable="d") indiquent que les attributs préfixés par d (c’est-à-dire ceux ajoutés par le designer) seront ignorés à la compilation.
* La ligne 6 déclare un alias pour l’espace de noms C# courant (celui dans lequel se trouve la classe de la fenêtre).
* La ligne 8 décrit le titre et les dimensions de la fenêtre

## Accès aux types d’un espace de noms

Le code xaml ne se limite pas à l’utilisation des classes WPF. Il peut utiliser également des classes définies par nous-même, ou définies dans des assemblies externes.

### Accès aux types du projet courant

Revenons sur la ligne 6 de l’extrait de code précédent :

xmlns:local="clr-namespace:MonAppli"

L’attribut **clr-namespace** permet de faire référence à un espace de noms contenant des classes publiques qu’on souhaite utiliser dans le xaml. On affecte généralement le préfixe « local » à cet espace de noms, mais on pourrait utiliser un autre préfixe.

Supposons qu’on ait créé dans le projet un contrôle personnalisé nommée MonControle, dérivée de UIElement. On pourra alors l’utiliser dans le code XAML de la fenêtre, de cette façon :

<Window x:Class="MonAppli.MainWindow"

...

xmlns:local="clr-namespace:MonAppli"

Title="MainWindow" Height="300" Width="300">

...

<local:MonControle/>

...

</Window>

Cette technique s’applique à toutes sortes de types, et pas seulement des composants visuels. L’exemple suivant montre la syntaxe pour utiliser une classe de convertisseur (utilisé dans les liaisons de données), et une constante, c’est-à-dire une variable statique :

<Window x:Class="LiaisonDonnées.LiaisonCollection"

        ...

        xmlns:local="clr-namespace:LiaisonDonnées"

...

        Title="LiaisonCollection" Height="560" Width="400">

<Window.Resources>

<local:DecimalToColorBrushConverter x:Key="conv"/>

</Window.Resources>  
  
 <TextBlock Text="{x:Static local:MesConstantes.Titre}"  
</Window>

Le convertisseur est instancié par le code xaml lui-même. DecimalToColorBrushConverter est bien le nom d’une classe, et non celui d’une variable contenant un objet.

La variable statique Titre est déclarée dans une classe MesConstantes :

public class MesConstantes

{

public const string Titre = "Utilisation des espaces de noms";

}

### Accès aux types définis dans d’autres assemblies

Il peut parfois être nécessaire d’accéder à des types définis dans d’autre assemblies que l’assembly courante. Dans ce cas, il faut :

* Ajouter une référence vers cet assembly dans le projet courant
* Dans le xaml, préciser le nom de l’assembly (sans l’extension .dll), après celui de l’espace de noms

Exemple :

<Window x:Class="Ressources.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:sys="clr-namespace:System;assembly=mscorlib"

Title="MainWindow" Height="350" Width="525">

<Window.Resources>

<sys:Double x:Key="TitleFontSize">30</sys:Double>

</Window.Resources>

Nous créons ici une ressource de type Double. Ce type est défini dans l’assembly mscorlib.dll du .net framework, et appartient à l’espace de noms System. Nous déclarons donc un préfixe sys pour cet espace de nom et cet assembly.

NB/ mscorlib (Multilanguage Standard Common Object Runtime Library) est une des assemblies de base du runtime .net, qui contient entre autres les définitions de types simples. Tous le projets Desktop contiennent une référence implicite vers elle, il n’est donc pas nécessaire d’ajouter soi-même cette référence.

# Les ressources

Le mot « ressources » désigne plusieurs choses différentes :

* Des objets déclarés en xaml, à l’intérieur de dictionnaires (ex : styles, templates, convertisseurs…)
* Des fichiers externes incorporés dans l’application, mais non modifiés par elle (ex : fichiers images, sons, vidéos, textes…)
* Des valeurs saisies directement dans des fichiers resx. Il s’agit généralement, mais pas obligatoirement, de ressources localisables (ex : libellés traductibles).

Les objets ressources déclarés dans le xaml ne font pas partie de l’arbre visuel, mais sont utilisés par les éléments WPF. Ils permettent de référencer toutes sortes d’objets : styles, templates, instances de classes, constantes, ressources externes…etc.

**Intérêts des ressources :**

* La déclaration d’un objet en tant ressource permet de le partager entre tous les éléments qui en ont besoin. Par exemple, un objet Brush peut fournir une couleur utilisable partout dans l’application.
* En séparant les objets utilisés par l’interface, de l’interface elle-même, celle-ci devient plus évolutive et adaptable. On peut par exemple définir différents dictionnaires de ressources, adaptés à différentes situations (paramétrage, culture…)

Nous allons voir maintenant comment créer, ou charger des ressources, et y accéder par le code.

## Créer une ressource

### Déclarer un objet ressource en xaml

**Dans une fenêtre**

<Window x:Class="Ressources.MainWindow"

        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

        xmlns:sys="clr-namespace:System;assembly=mscorlib"

        xmlns:local="clr-namespace:Ressources"

        Title="Ressources" Height="350" Width="525">

   <Window.Resources>

      <ResourceDictionary>

         <!-- Style -->

         <Style x:Key="styleTitre">

            <Setter Property="TextBlock.FontFamily" Value="Verdana"/>

            <Setter Property="TextBlock.FontSize" Value="30"/>

            <Setter Property="TextBlock.TextAlignment" Value="Center"/>

         </Style>

         <!-- Ressource binaire incorporée à l'exe -->

         <Image x:Key="imgAsterix" Height="150"

                Source="pack://application:,,,/Images/Astérix.jpg"/>

         <!-- Type défini dans un autre assembly -->  
         <sys:String x:Key="strAsterix">Astérix</sys:String>

      </ResourceDictionary>

   </Window.Resources>

Nous déclarons ici 3 ressources de types différents dans le dictionnaire par défaut de la fenêtre (Window.resources) :

* La première est un élément style WPF qui s’applique à des TextBlock
* La seconde est une image incorporée à l’exe
* La dernière est de type String (type défini dans l’assembly externe mscorlib.dll, auquel on accède grâce au préfixe d’espace de nom sys déclaré plus haut)

**Au niveau de l’application**

On peut définir des ressources au niveau de l’application pour qu’elles soient accessibles par toutes les fenêtres de l’application. Pour cela, il suffit de les ajouter au dictionnaire Application.Resources défini dans le fichier App.xaml :

<Application x:Class="Ressources.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:Ressources"

StartupUri="MainWindow.xaml">

<Application.Resources>

</Application.Resources>

</Application>

**Caractéristiques :**

* Tout élément WPF dérivant de FrameworkElement possède une propriété **Resources,** de type ResourceDictionary, représentant son dictionnaire de ressources **local**.
* Les ressources d’un élément WPF sont accessibles par tous ses éléments enfants dans l’arbre visuel. Généralement, on définit les ressources sur l’élément Window.
* L’application possède également un dictionnaire de ressources utilisable partout dans l’application. Il se trouve dans le fichier **App.xaml**
* Dans un dictionnaire de ressources, chaque ressource est identifiée par une clé, représentée par la propriété **x:Key**. C’est cette clé qui est utilisée par les éléments WPF pour accéder à la ressource.

### Créer un dictionnaire de ressources

Nous avons vu que chaque élément WPF pouvait avoir un dictionnaire de ressources local. Mais il est également possible de créer un dictionnaire de ressources dans un fichier xaml séparé. Cela permet de regrouper et classer les ressources.

On peut éventuellement créer un projet spécifique contenant uniquement des dictionnaires de ressources, qui sera ensuite référencé par les autres projets de la solution, facilitant ainsi le partage des ressources.

Pour créer un dictionnaire de ressource dans un fichier séparé avec Visual Studio :

* Cliquer sur le menu « Projet \ Ajouter un dictionnaire de ressources », et donner un nom au dictionnaire. Celui-ci est créé sous forme de fichier xaml.
* Ajouter des ressources au dictionnaire. Exemple :

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<SolidColorBrush x:Key="appBrush" Color="AliceBlue"/>

</ResourceDictionary>

### Ajouter un fichier externe au projet

Pour être utilisable dans l’application, un fichier externe doit être référencé dans le projet. Pour cela, dans Visual Studio, il faut :

* Cliquer sur le menu Projet \ Ajouter un élément existant
* Sélectionner le fichier à ajouter
* Dans la fenêtre des propriétés, affecter les propriété « Action de génération » (Build Action) et « Copier dans le répertoire de sortie » du fichier

**La propriété « Action de génération »** définit la façon dont sera traité le fichier à la compilation. Il faut sélectionner l’une des 2 valeurs suivantes :

* **Ressource** : pour incorporer le fichier dans la dll ou l’exe
* **Contenu** : pour garder le fichier indépendant

Remarques :

* Il existe une valeur « Ressource incorporée », mais elle n’est pas adaptée aux fichiers externes, mais plutôt aux fichiers resx.
* La valeur « Contenu » sera surtout utilisée pour de gros fichiers qu’on ne souhaite pas incorporer à l’exe, ou pour des fichiers utilisés par plusieurs dll. Mais dans ce dernier cas, on peut aussi créer un projet spécifique pour les ressources partagées, et le référencer dans les autres projets.
* Les ressources incorporées ne sont pas modifiables sans recompiler le projet. Si on veut pouvoir modifier un fichier, il faut donc affecter la valeur Resource à sa propriété Action de génération.

**La propriété « Copier dans le répertoire de sortie »** définit si on veut ou non que le fichier soit copié à côté de la dll ou de l’exe lors de la compilation. La valeur « Toujours copier » est pertinente si « Action de génération = Contenu ». Sinon, il faut laisser la valeur « Ne pas copier ».

En copiant le fichier à côté de l’exe, on simplifie beaucoup son chemin d’accès, puisqu’il se résumera au nom du fichier.

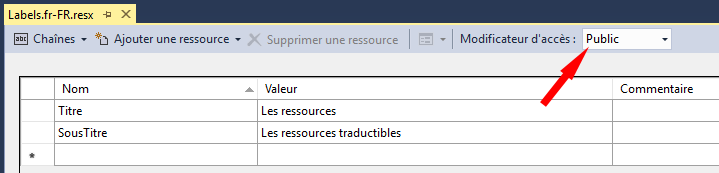
### Créer un fichier resx

Un fichier resx peut être ajouté facilement au projet en cliquant sur le menu « Projet \ Ajouter un nouvel élément », et en sélectionnant « Fichier de ressource ».

Ceci ajoute en fait 2 fichiers :

* Un fichier à l’extension « .Designer.cs », qui contient les classes permettant de stocker et d’accéder aux valeurs de ressources.
* Un fichier à l’extension « .resx » au format xml, qui contient les types et valeurs de ressources

Ces fichiers ne sont jamais édités directement, mais plutôt au travers d’une interface graphique qui présente les ressources sous forme de tableau, pour faciliter la saisie et la consultation. Cette interface s’ouvre automatiquement lorsqu’on double-clique sur le fichier resx :



/!\ Penser à rendre l’accès public (flèche rouge) pour que les ressources soient accessibles depuis le xaml.

Bonnes pratiques :

* Il est important de partager les ressources traductibles qui peuvent l’être pour réduire le nombre de ressources, et ainsi diminuer le travail de traduction
* Séparer les ressources traductibles des autres ressources pour faciliter leur extraction. On peut utiliser pour cela plusieurs fichiers resx.

**Gestion des cultures**

Les fichiers resx sont particulièrement adaptés au stockage de valeurs localisables, c’est-à-dire dépendant d’une culture. Une culture désigne un couple langue + pays.

On peut déployer plusieurs dll de ressources correspondant aux cultures visées par l’application. Le framework fournit un mécanisme permettant à l’application d’utiliser automatiquement la dll adaptée à la culture du système d’exploitation sur lequel l’application s’exécute.

Si une ressource n’est pas trouvée pour la culture du système d’exploitation, la ressource de la culture neutre est utilisée par défaut.

Pour profiter de ce mécanisme, on procédera de la façon suivante :

* Créer un premier fichier de ressources pour la culture neutre (ex : Labels.resx)
* Créer ensuite un fichier pour chaque culture qu’on souhaite prendre en charge. Le fichier doit avoir le même nom que le précédent, et être suffixé par la culture  
  ex : Labels.fr-FR.resx, Labels.en-US.resx…etc.

Les propriétés de ces fichiers sont par défaut :

* Action de génération : ressource incorporée
* Copier dans le répertoire de sortie : ne pas copier

Il faut laisser ces valeurs par défaut.

L’image ci-dessous montre l’exemple d’un projet nommé Ressources, contenant des fichiers resx placés dans un sous-répertoire FichiersResx du projet :

|  |  |
| --- | --- |
|  | La compilation de ce projet, produit :   * Un fichier Ressources.exe correspondant à l’application * Un répertoire pour chaque culture trouvée parmi les resx :     Chaque répertoire contient lui-même une dll nommée Ressources.resources.dll NB/ Le nom est constitué du nom du projet, suivi de « resources.dll » |

## Utiliser une ressource

### Objet ressource xaml

Un élément xaml peut utiliser un objet ressource xaml de 2 façons : statique ou dynamique.

**De façon Statique**

On utilise une ressource de façon statique au moyen d’une des syntaxes suivantes, selon que la ressource est utilisée comme élément WPF indépendant, ou comme propriété d’un autre élément :

<StaticResource ResourceKey="clé"/>

{StaticResource Clé}

Exemple d’utilisation des ressources définies plus haut :

<!-- Utilisation de l'image déclarée en tant que ressource -->

<StaticResource ResourceKey="imgAsterix"/>

<!-- Utilisation du style et de la chaîne déclarés en tant que ressources -->

<TextBlock Style="{StaticResource styleTitre}"

Text="{StaticResource strAsterix}"/>

Lorsqu’on fait référence à une ressource de façon statique, la ressource est chargée une fois pour toutes par l’élément qui la référence. L’objet ressource référencé reste toujours le même, mais cela n’empêche pas que ses propriétés changent (elles peuvent être modifiées par le code C# par exemple).

**De façon dynamique**

Pour utiliser une ressource de façon dynamique, on utilise la syntaxe :

{DynamicResource Clé}

Lorsqu’on fait référence à une ressource de façon dynamique, si l’objet ressource référencé change (c’est-à-dire si la ressource désigne une autre instance d’objet), l’élément WPF qui l’utilise en est notifié, et utilise cette nouvelle instance.

/!\ Les ressources dynamiques étant chargées à chaque fois qu’elle sont utilisées, leur utilisation diminue les performances de l’application. Sauf besoin spécifique, il est donc conseillé d’utiliser des ressources statiques

**Accès aux à un dictionnaire de ressource indépendant**

Pour pouvoir accéder aux ressources d’un dictionnaire crée dans un fichier xaml indépendant, il faut tout d’abord fusionner ce dictionnaire avec un dictionnaire déjà accessible (par exemple un dictionnaire local, ou celui de l’application).

Exemple : fusionnons un dictionnaire indépendant nommé DicoRes.xaml, avec celui de la fenêtre :

<Window.Resources>

<ResourceDictionary>

<!--Type simple-->

<sys:String x:Key="strAsterix">Astérix</sys:String>

<!--Fusion d'un dictionnaire externe-->

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="DicoRes.xaml"/>

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Window.Resources>

La fusion se fait par ajout du dictionnaire à la collection ResourceDictionary.MergedDictionaries

**Accès aux objets ressources depuis le code**

On peut également accéder à une ressource depuis le code, pour peu qu’on ait accès au dictionnaire de ressources qui la contient. La syntaxe est celle de l’utilisation classique d’une collection non générique de type dictionnaire. Les valeurs obtenues doivent être transtypées.

Ex1 : accès depuis le code-behind à une ressource stockée dans le dictionnaire d’une fenêtre :

string s = (string)Resources["strAstérix"];

Ex2 : accès à une ressource stockée dans le dictionnaire de l’application

decimal seuil = (decimal)Application.Current.Resources["Seuil"];

### Ressource externe incorporée à l’exe

L’exemple ci-dessous montre comment accéder à une image ajoutée au projet, et incorporée à l’exe

<DockPanel>

<!-- Utilisation directe d'une image incorporée à l'exe -->

<Image Source="pack://application:,,,/Images/Lucky Luke.jpg" Height="150"/>

</DockPanel>

La chaîne spécifiée pour la propriété Source est ce qu’on appelle une URI pack.

La première partie (pack://application:,,,/) désigne une ressource incorporée à l’exe

La seconde partie (/Images/Lucky Luke.jpg) désigne le chemin relatif de la ressource. Il reflète l’arborescence du projet. Ici, l’image a été placée dans un répertoire nommé « Images » à l’intérieur du projet.

Certaines ressources ne sont pas accessibles via une URI pack. Dans ce cas, on peut les charger par le code, de la façon suivante :

using System;

using System.IO;

using System.Windows;

using System.Text;

using System.Windows.Documents;

namespace Ressources

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

// Chargement d'une ressource incorporée à l'exe (ici un fichier texte)

var streamResInfo = Application.GetResourceStream(new Uri("Textes/Readme.txt", UriKind.Relative));

var reader = new StreamReader(streamResInfo.Stream, Encoding.Default);

string txt = reader.ReadToEnd();

// Le texte est chargé dans un document affiché dans le viewer

var doc = new FlowDocument(new Paragraph(new Run(txt)));

doc.TextAlignment = TextAlignment.Left;

docViewer.Document = doc;

}

}

}

Dans cet exemple, la ressource est un fichier texte nommé readme.txt, placé dans le sous-répertoire Textes du projet. Il est géré comme ressource externe incorporée à l’exe.

Pour charger cette ressource, on utilise la méthode statique GetResourceStream de la classe Application.

Le flux obtenu est ici lu au moyen d’un objet StreamReader, puis chargé dans un objet FlowDocument. docViewer est le nom d’un élément <FlowDocumentScrollViewer> placé dans le code xaml de la fenêtre, et qui permet d’afficher le document créé.

### Fichier à côté de l’exe

Si on veut accéder à un fichier non connu de l’application au moment de la compilation (c’est-à-dire pas ajouté au projet), on peut utiliser le code ci-dessous pour obtenir le chemin du répertoire de l'exe, et ainsi construire un chemin d’accès au fichier souhaité :

string dirPath = Path.GetDirectoryName(System.AppDomain.CurrentDomain.BaseDirectory);

### Ressource d’un fichier resx

Pour accéder aux ressources stockées dans un fichier resx, il faut tout d’abord ajouter un préfixe pour l’espace de noms dans lequel se trouvent les ressources. On accède ensuite à chaque ressource comme on accède à n’importe quelle propriété statique :

<Window x:Class="Ressources.MainWindow"

...

xmlns:resx="clr-namespace:Ressources.FichiersResx"

Title="Ressources" Height="350" Width="525">

<Grid Margin="5">

<!-- Utilisation d'une ressource traductible du fichier resx -->

<TextBlock Text="{x:Static resx:Labels.Titre}"/>  
 ...

Dans cet exemple, on accède à une valeur de ressource nommée « Titre », contenue dans un fichier resx nommé Labels. Le fichier resx est placé dans un sous-répertoire nommé « FichiersResx », du projet nommé « Ressources ». On a créé un préfixe d’espace de noms « resx » pour y accéder.

# Les styles

Les styles WPF peuvent être comparés aux styles CSS utilisés en HTML. Ils permettent de modifier l’apparence standard des éléments visuels, et de créer des applications visuellement cohérentes.

La notion de style est très vaste en WPF. Nous ne verrons ici que les bases.

## Création d’un style

Un style est défini au moyen de l’élément **Style**, dans lequel on place des éléments **Setter**

Un **Setter** permet d’affecter une valeur à une propriété d’un élément. Dans l’exemple ci-dessous, on affecte la couleur de fond des boutons :

<Style x:Key="Style1"> >

<Setter Property="Button.Background" Value="AliceBlue"/>

</Style>

La propriété à affecter est toujours désignée par la syntaxe Elément.Propriété.

Si on veut pouvoir affecter un style à plusieurs types d’éléments, on peut utiliser un type ancêtre. Par exemple, pour affecter la couleur de fond de n’importe que contrôle, on utilisera :

<Setter Property="Control.Background" Value="AliceBlue"/>

## Application d’un style

Les styles sont généralement appliqués sur plusieurs instances d’éléments, c’est pourquoi ils sont définis en tant que ressources, soit locales, soit de l’application.

Il y a 2 façons d’appliquer un style défini en tant que ressource :

* En faisant référence à sa clé de ressource :

<StackPanel>

<StackPanel.Resources>

<Style x:Key="StyleBouton1">

<Setter Property="Button.Background" Value="AliceBlue"/>

<Setter Property="Button.Height" Value="30"/>

</Style>

</StackPanel.Resources>

<Button Content="Un bouton" Style="{StaticResource StyleBouton1}"/>

</StackPanel>

* En définissant un style qui s’applique à tous les éléments WPF d’un type spécifique :

<StackPanel>

<StackPanel.Resources>

<Style TargetType="Button">

<Setter Property="Background" Value="AliceBlue"/>

<Setter Property="Height" Value="30"/>

</Style>

</StackPanel.Resources>

<Button Content="Un bouton"/>

</StackPanel>

Dans la seconde méthode, on note 3 différences par rapport à la première :

* Au lieu de spécifier une clé de ressource pour le style, on affecte sa propriété TargetType
* Il n’est plus nécessaire de spécifier le type de l’élément dans la propriété Property des Setter
* Les contrôles sur lesquels le style s’applique n’ont pas besoin de faire référence à une ressource. Le style leur est automatiquement appliqué.

On peut annuler le style automatique sur un contrôle particulier en lui affectant un style Null :

<Button Content="Un bouton" Style="{x:Null}"/>

Enfin, on peut affecter dynamiquement le style d’un élément par le code :

Btn1.Style = (Style)Resources["StyleBouton1"];

## Héritage

Comme pour l’héritage de classes, l’héritage de style permet de définir des styles dérivés d’un style de base, pour ajouter ou modifier des propriétés.

Pour mettre en œuvre l’héritage, on utilise la propriété **BasedOn** :

<Style x:Key="StyleBouton3" BasedOn="{StaticResource StyleBouton1}">

<Setter Property="Button.Foreground" Value="Red"/>

</Style>

Ici, StyleBouton3 hérite de tous les Setter de StyleBouton1, et en ajoute un qui lui est propre.

On peut hériter d’un style défini sur un type avec TargetType de la façon suivante :

<Style TargetType="Button">

<Setter Property="Height" Value="30"/>

<Setter Property="Foreground" Value="red"/>

</Style>

<Style x:Key="Style1" TargetType="Button"

BasedOn="{StaticResource {x:Type Button}}">

<Setter Property="Background" Value="AliceBlue"/>

<Setter Property="Foreground" Value="Blue"/>

</Style>

Un bouton auquel on applique Style1 ressemblera alors à ceci :



Style1 modifie la couleur de police en bleu, et ajoute la couleur de fond bleu clair. La hauteur est toujours définie par le style de base.

## Triggers

Les triggers permettent de décrire en xaml, comment les valeurs de certaines propriétés doivent être affectées par les changements de valeurs d’autres propriétés.

Exemple :

<Style TargetType="Button">

<Setter Property="Background" Value="AliceBlue"/>

<Setter Property="Height" Value="30"/>

<Style.Triggers>

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Foreground" Value="Red"/>

</Trigger>

</Style.Triggers>

</Style>

Ce trigger indique que lorsque la propriété IsMouseover d’un bouton à la valeur True, alors sa propriété Foreground doit être affectée à Red. Autrement dit, lorsque la souris survole le bouton, sa couleur de police doit être rouge :



**Caractéristiques**

* Un trigger doit être placé dans la collection Triggers d’un style
* Il existe en fait 5 types de triggers répondant à différents besoins :
* Trigger : se déclenche quand une propriété reçoit une valeur donnée
* MultiTrigger : se déclenche quand un ensemble de propriétés reçoivent des valeurs données
* DataTrigger : se déclenche quand la propriété liée par DataBinding reçoit une valeur donnée
* MultiDataTrigger : se déclenche quand un ensemble de propriétés liées par DataBinding reçoivent des valeurs données
* EventTrigger : initie une série d’actions quand un évènement est déclenché

Nous ne les détaillerons pas dans ce cours. A titre indicatif, voici un exemple de MultiTrigger, qui change la couleur de fond des zones de texte lorsqu’elles ont le focus et qu’elles sont survolées avec la souris :

<Style TargetType="TextBox">

<Style.Triggers>

<MultiTrigger>

<MultiTrigger.Conditions>

<Condition Property="IsKeyboardFocused" Value="True" />

<Condition Property="IsMouseOver" Value="True" />

</MultiTrigger.Conditions>

<MultiTrigger.Setters>

<Setter Property="Background" Value="LightGreen" />

</MultiTrigger.Setters>

</MultiTrigger>

</Style.Triggers>

</Style>

# La liaison de données (DataBinding)

## Principe du DataBinding

**Définition**

La databinding est un mécanisme permettant de lier de façon simple une propriété d’un élément WPF (qu’on appelle la propriété cible), à une source de données.

**Caractéristiques**

* La propriété cible peut être quasiment n’importe quelle propriété WPF
* La source peut être une propriété d’un autre élément WPF, ou un objet quelconque (éventuellement une collection) fourni par le code.
* La liaison est simplement **décrite** en xaml, et WPF se débrouille pour qu’elle soit effective en temps réel
* Le lien peut fonctionner dans les 2 sens, afin d’afficher et/ou de modifier les données sources

Le databinding fournit donc un mécanisme puissant pour créer des interfaces visuelles rafraîchies en temps réel, et transmettre à la source les modifications (au sens large) faites par l’utilisateur.

Exemple : on souhaite afficher en temps réel dans un bloc de texte, la valeur numérique sélectionnée au moyen d’un curseur :



Pour cela, il suffit de créer une liaison entre la propriété Value du Slider et la propriété Text du TextBlock.

La liaison de données entre la cible et sa source est réalisée à l'aide d'un objet [**Binding**](https://msdn.microsoft.com/fr-fr/library/system.windows.data.binding(v=vs.110).aspx), qui définit le comportement de la liaison. Voici les propriétés le plus importantes de cette classe :

|  |  |  |
| --- | --- | --- |
| **Propriété** | **Type** | **Description** |
| ElementName | String | Nom de l’élément WPF source. |
| RelativeSource | RelativeSource | Elément WPF source spécifié par son emplacement relatif par rapport à la cible |
| Source | Object | Objet source des données, dans le cas d’une liaison à un objet qui n’est pas un élément WPF |
| Path | PropertyPath | Chemin de la propriété de l’objet source qui participe à la liaison |
| Mode | BindingMode  Enumération avec 4 valeurs possibles | Définit la direction du flux de données.   * OneTime : la propriété cible est affectée uniquement à l’initialisation de l’application et n’est plus mise à jour ensuite * OneWay : la propriété cible est mise à jour à chaque changement de la propriété source * OneWayToSource : la propriété source est mise à jour à chaque changement de la propriété cible * TwoWay : les propriétés cible et source sont mises à jour à chaque changement de l’une ou de l’autre |
| UpdateSourceTrigger | UpdateSourceTrigger  Enumération avec 4 valeurs possibles | Définit quand les données sont transmises à la source :   * PropertyChange : à chaque changement de valeur * LostFocus : à la perte de focus * Explicit : par déclenchement explicite en appelant UpdateSource() |

**Remarques** :

* Seule l’une des 3 propriétés ElementName, RelativeSource ou Source peut être non null.
* Il n’y a pas de propriété pour définir la cible, car le binding est défini directement sur la propriété cible, comme nous allons le voir bientôt.

## Propriétés Mode et UpdateSourceTrigger

**La propriété Binding.Mode** détermine le comportement des propriétés liées en réponse à une modification de la source ou de la cible.

Nous avons décrit plus haut les 4 valeurs possibles de cette propriété, qui sont représentées par le schéma ci-dessous.

Objet

Objet WPF

Propriété WPF

Propriété

**Binding**

**Cible**

**Source**

OneTime

OneWay

OneWayToSource

TwoWay

NB/ Chaque propriété WPF a un mode par défaut. Pour déterminer ce mode, il faut regarder la section « Information sur les propriétés de dépendance » de la doc MSDN de la propriété. Cette section indique la valeur de la métadonnée BindsTwoWayByDefault. Si cette valeur est à vrai, la propriété à un mode TwoWay par défaut.

Les propriétés éditables par l’utilisateur, comme TextBox.Text, ComboBox.Text… sont généralement en TwoWay. Pour changer de mode, on fera par exemple :

<TextBox Text="{Binding Nom, Mode=OneWay}"/>

Une liaison en mode OneWay ou OneWayToSource surveille les modifications de la propriété cible, et met à jour la source en conséquence.

**La propriété Binding.UpdateSourceTrigger** détermine quand cette mise à jour a lieu.

La valeur par défaut de cette propriété diffère selon le type de propriété WPF sur laquelle est faite le binding.

En règle générale, la valeur par défaut est PropertyChanged, c’est-à-dire que la mise à jour intervient à chaque fois que la propriété cible change. Cependant, pour une propriété WPF Text, dont la valeur est saisissable par l’utilisateur, ce n’est pas approprié. En effet, faire la mise à jour à chaque pression d’une touche entraînerait des problèmes de performances, et empêcherait l’utilisateur de corriger les erreurs de frappes avant de valider la valeur. C’est pourquoi, pour la propriété Text, la valeur par défaut de UpdateSourceTrigger est LostFocus, c’est-à-dire que la mise à jour intervient lorsque la zone de saisie perd le focus.

On peut être amené à spécifier la valeur Explicit, si on souhaite maîtriser le déclenchement de la mise à jour, par exemple pour faire avant des traitements complexes de vérification.

## Liaison avec un objet unique

Nous allons voir concrètement comment créer une liaison d’une propriété d’un élément WPF avec un objet unique dans différents cas de figures.

La liaison peut être créée soit directement dans le code xaml (solution la plus utilisée), soit par le code C#.

### Autre élément WPF

Reprenons l’exemple précédent, dans lequel on affiche en temps réel dans un bloc de texte, la valeur numérique sélectionnée au moyen d’un curseur :



**Implémentation en xaml avec syntaxe longue :**

<Slider Name="sldAge" Height="20"

Maximum="120" LargeChange="10" SmallChange="1"

TickFrequency="1" IsSnapToTickEnabled="True" />

<TextBlock VerticalAlignment="Center">

<TextBlock.Text>

<Binding ElementName="sldAge" Path="Value"/>

</TextBlock.Text>

</TextBlock>

A l’aide d’un objet binding, on lie la propriété Text du TextBlock (propriété cible) à la propriété Value du Slider (propriété source). La source du binding étant un élément WPF, on utilise la propriété **ElementName**.

**Implémentation en xaml avec syntaxe condensée :**

La syntaxe suivante est équivalente à la précédente :

<Slider Name="sldAge" Height="20"

Maximum="120" LargeChange="10" SmallChange="1"

TickFrequency="1" IsSnapToTickEnabled="True" />

<TextBlock VerticalAlignment="Center"

Text="{Binding ElementName=sldAge, Path=Value}"/>

Cette syntaxe est la plus utilisée car moins verbeuse.

La paire d’accolades est ce que l’on appelle une [**Markup Extension**](https://msdn.microsoft.com/fr-fr/library/ee855815(v=vs.110).aspx) (extension de balisage). Elle permet d’affecter un objet xaml (ici Binding) à une propriété xaml (ici Text).

**Implémentation par le code**

On peut également créer la même liaison par le code C#, de cette façon :

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

var bdg = new Binding();

bdg.ElementName = "sldAge";

bdg.Path = new PropertyPath("Value");

lblAge.SetBinding(ContentProperty, bdg);

}

}

L’implémentation d’un binding par le code est surtout utile lorsqu’on veut créer la liaison dynamiquement.

### Objet statique ou ressource

La liaison peut être réalisée avec tout objet accessible depuis la vue xaml.

#### Liaison avec un objet statique

Dans l’exemple qui suit, on définit la couleur de fond d’un bouton grâce à une liaison avec un objet statique de type Brush :

<Button Content="Valider" Height="30"

Background="{Binding Source={x:Static SystemColors.InactiveCaptionBrush}}"/>

On utilise cette fois la propriété **Source** pour spécifier la source du binding.

InactiveCaptionBrush est une propriété statique de la classe statique SystemColors. On y accède donc en la faisant précéder de x:Static (il s’agit d’une autre Markup Extension).

Il n’y a ici pas besoin de spécifier la propriété Path puisqu’on fait le lien directement sur la propriété.

La syntaxe longue équivalente est la suivante :

<Button Content="Valider" Height="30">

<Button.Background>

<Binding>

<Binding.Source>

<x:Static Member="SystemColors.HighlightBrush"/>

</Binding.Source>

</Binding>

</Button.Background>

</Button>

NB/ Dans cet exemple précis, l'utilisation d'un binding n'a pas grand intérêt, car on peut utiliser la propriété statique de cette façon suivante :

<Button Content="Valider" Height="30"

Background="{x:Static SystemColors.InactiveCaptionBrush}"/>

Il aurait un intérêt si on appliquait par exemple un convertisseur sur la liaison (cf. plus loin).

#### Liaison avec une ressource

Exemple : créons une ressource pour la couleur d’arrière-plan des zones de saisies :

<Window.Resources>

<SolidColorBrush x:Key="bgColor" Color="AliceBlue"/>

</Window.Resources>

On peut utiliser cette ressource dans un binding de la façon suivante :

<TextBox Background="{Binding Source={StaticResource bgColor}}" Text="…"/>

La syntaxe est très similaire à celle d’une liaison avec un objet statique.

### Objet instancié par le code

Le plus souvent, les données affichées par le visuel sont fournies par des objets qu’on instancie soi-même dans le code C#. Typiquement il s’agira d’objets contenant des données issues d’une base de données.

Dans ce cas, l’objet n’est pas affecté à la propriété Source du Binding, mais plutôt à la propriété **DataContext** de l’élément WPF cible ou de l’un de ses parents. C’est pourquoi on appelle cet objet « contexte de la liaison ».

NB/ DataContext est une propriété de type Object définie par la classe FrameworkElement. Elle est donc disponible sur tous les éléments WPF.

Exemple : nous allons créer une classe Personne avec quelques propriétés, instancier un objet de cette classe, puis le définir comme contexte d’une liaison de données avec les contrôles d’une interface visuelle simple.

Classe Personne :

public class Personne

{

public string Nom { get; set; }

public string Prénom { get; set; }

public DateTime DateNais { get; set; }

}

Instanciation et utilisation comme contexte de données sur la grille principale de la fenêtre :

public partial class MainWindow : Window

{

public Personne Adupont { get; }

public MainWindow()

{

InitializeComponent();

// Création d'une personne

Adupont = new Personne {

Nom = "Dupont",

Prénom = "Albert",

DateNais = new DateTime(1990,01,25) };

// Utilisation comme contexte de données sur la grille

// principale du formulaire WPF

gridForm.DataContext = Adupont;

}

}

Ici, gridForm est le nom de la grille qui sert de conteneur principal de la fenêtre dont voici le code :

<Grid x:Name="gridForm" Margin="5">

<!-- Création des lignes -->

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<!-- Création des colonnes -->

<Grid.ColumnDefinitions>

<ColumnDefinition Width="1\*"/>

<ColumnDefinition Width="3\*"/>

</Grid.ColumnDefinitions>

<!-- Ligne 0 -->

<Label Content="Nom"/>

<TextBox Grid.Column="1" Text="{Binding Nom}"/>

<!-- Ligne 1 -->

<Label Content="Prénom" Grid.Row="1"/>

<TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Prénom}"/>

<!-- Ligne 2 -->

<Label Content="Date nais." Grid.Row="2"/>

<DatePicker Grid.Row="2" Grid.Column="1" VerticalAlignment="Center"

SelectedDate="{Binding DateNais}"/>

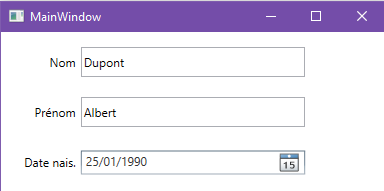
Tous les éléments WPF à l’intérieur de la grille partagent la valeur de la propriété DataContexte de la grille (définie dans le code C#).

Pour lier les propriétés des éléments WPF aux propriétés de l’objet Adupont (de type Personne), il suffit alors de faire un binding xaml tout simple avec l’une des 2 syntaxes suivante, qui sont équivalentes :

Text="{Binding Path=Nom}"/>  
Text="{Binding Nom}"/>

NB/ La propriété Path est la propriété par défaut de Binding, c’est pourquoi on peut omettre son nom.

Le visuel obtenu est le suivant :



**Recherche du contexte**

Si la propriété ElementName, RelativeSource ou Source d’un Binding n’est pas définie, WPF cherche une valeur de DataContext, en commençant par l’élément sur lequel est créé le Binding, et en remontant l’arbre visuel jusqu’à trouver un élément dont la propriété DataContext est définie.

Ceci permet de lier tous les éléments d’un conteneur à une même source de données.

**Instanciation du contexte dans le xaml**

L’objet qui sert de contexte est généralement instancié par le code C#, mais il est possible de l’instancier par le code xaml, comme ceci :

<Window x:Class="MonAppli.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:MonAppli"

mc:Ignorable="d"

Title="MainWindow" Height="300" Width="400">

<Grid x:Name="gridForm" Margin="5">

<Grid.DataContext>

<local:Personne/>

</Grid.DataContext>

Ceci crée une instance de la classe Personne et l’affecte à la propriété DataContexte de la grille. Cela n’a pas d’intérêt en l’état, car à moins de définir des valeurs par défaut pour les propriétés dans le constructeur de la classe Personne, les bindings définis plus haut n’auront aucune donnée à afficher.

## Liaison avec une collections d’objets

Nous allons voir ici des exemples de liaisons avec des collections d’objets, dans différents scénarios.

Comme nous l’avons vu plus haut, les contrôles spécialisés dans l’affichage de collections d’objets sont ceux qui dérivent de **ItemsControls**. Cette classe possède les propriétés suivantes concernant la liaison de données :

|  |  |  |
| --- | --- | --- |
| **Propriété** | **Type** | **Description** |
| ItemsSource | IEnumerable | Représente la collection d’objets source pour le contenu du contrôle C’est la propriété sur laquelle on fait le Binding |
| DisplayMemberPath | String | Propriété des objets de la collection qui sera utilisée pour l’affichage dans le contrôle |
| ItemStringFormat | String | Spécifie le format d’affichage des éléments s'ils sont affichés en tant que chaînes. Cela peut être un format prédéfini (G, F, D, X…), ou un format personnalisé (ex : MMMM/yyyy) |
| ItemTemplate | DataTemplate | DataTemplate utilisé pour afficher chaque élément (cf. plus bas) |

Les scénarios qui suivent illustrent ces propriétés avec des contrôles de sélection ListBox, ComboBox et ListView.

### Collection simple

Dans l’exemple qui suit, nous allons afficher une liste d’achats dans une ListBox.   
La liste sera remplie à partir d’un fichier achats.xml dont voici le contenu :

<?xml version="1.0" encoding="UTF-8"?>

<Achats>

<Achat Id="1" Date="2016-01-15T20:32:01" NbArticles="2" Montant="118.70"/>

<Achat Id="2" Date="2016-02-10T12:45:02" NbArticles="1" Montant="15.90"/>

<Achat Id="3" Date="2016-03-24T19:30:03" NbArticles="2" Montant="47.25"/>

<Achat Id="4" Date="2016-04-30T14:03:04" NbArticles="3" Montant="245.00"/>

<Achat Id="5" Date="2016-05-15T19:54:05" NbArticles="1" Montant="9.60"/>

</Achats>

Pour cela, créons tout d’abord la classe Achat :

using System;

using System.Xml.Serialization;

namespace LiaisonDonnées

{

public class Achat

{

[XmlAttribute("Id")]

public int Id { get; set; }

[XmlAttribute("Date")]

public DateTime Date { get; set; }

[XmlAttribute("NbArticles")]

public int NbArticles { get; set; }

[XmlAttribute("Montant")]

public decimal Montant { get; set; }

}

}

Nous décorons chaque propriété d’un attribut indiquant à quel attribut xml elle correspond.

Dans le constructeur de la fenêtre, chargeons les données du fichier xml dans la liste :

public partial class LiaisonObjets : Window

{

public List<Achat> Achats { get; }

public LiaisonObjets()

{

InitializeComponent();

// Remplissage de la liste d'achats à partir d'un fichier xml

XmlSerializer deserializer = new XmlSerializer(typeof(List<Achat>),

new XmlRootAttribute("Achats"));

FileStream fs = new FileStream("Achats.xml", FileMode.Open);

Achats = (List<Achat>)deserializer.Deserialize(fs);

// Utilisation comme contexte de données sur la ListBox

lstAchats.DataContext = Achats;

}

}

NB/ Nous utilisons ici la classe XMLSerializer et sa méthode Deserialize pour charger automatiquement le contenu du fichier xml dans la liste d’achats. Le fichier xml doit être déployé à côté de l’exe.

La dernière ligne affecte la liste comme contexte d’un contrôle ListBox nommé lstAchats. Il ne reste plus qu’à réaliser la liaison en xaml :

<!--Affichage des dates des achats au format long-->

<ListBox Name="lstAchats"

ItemsSource="{Binding}" DisplayMemberPath="Date" ItemStringFormat="f"/>

Comme la propriété Source du binding n’est pas affectée, WPF recherche si la propriété DataContext a été affectée quelque part. Nous l’avons fait dans le code C#.

ItemStringFormat=”f” permet d’afficher la date au format long et l’heure au format court.

NB/ Notez que pour que les dates s’affichent bien selon la culture courante (et non pas au format anglais), il faut ajouter la ligne suivante dans le constructeur de la fenêtre :

Language = System.Windows.Markup.XmlLanguage.GetLanguage(System.Threading.Thread.CurrentThread.CurrentCulture.Name);

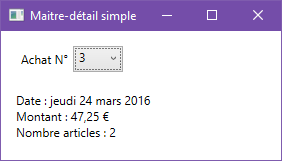
### Maître-détail simple

Dans ce type de vue, le détail ne fait que reprendre les données de l’objet maître sélectionné, pour les afficher sous une forme plus détaillée. Sa réalisation requiert 2 choses :

* Le maître et le détail doivent avoir le même contexte de données
* Sur le maître, la propriété IsSynchronizedWithCurrentItem doit être à True

La propriété **IsSynchronizedWithCurrentItem** force la source de données (la collection) à changer d’item courant, ce qui provoque la mise à jour de l’affichage du détail. Il n’y a aucun lien direct entre le maître et le détail au niveau du visuel.

Voici un exemple affichant une liste d’achats en maître-détail :



<!-- Maître -->

<StackPanel Orientation="Horizontal" Margin="10">

<Label Content="Achat N°"/>

<ComboBox ItemsSource="{Binding}" DisplayMemberPath="Id"

IsSynchronizedWithCurrentItem="True" Width="50"/>

</StackPanel>

<!-- Détail -->

<StackPanel Margin="10">

<TextBlock Text="{Binding Path=Date, StringFormat='Date : {0:D}'}"/>

<TextBlock Text="{Binding Path=Montant, StringFormat='Montant : {0:C}'}"/>

<TextBlock Text="{Binding Path=NbArticles, StringFormat='Nombre articles : {0}'}"/>

</StackPanel>

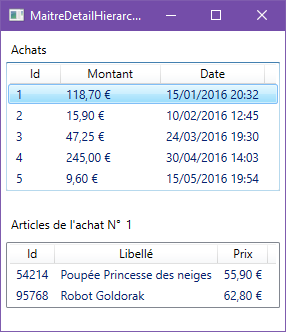
Dans le code C#, la liste d’achats a été définie comme contexte pour la ListBox et le StackPanel.

Notons au passage que la propriété StringFormat s’utilise de façon similaire à la méthode String.Format(). La chaîne est placée entre simples cotes pour ne pas perturber les doubles côtes autour du binding.

### Maître-détail hiérarchique

Dans ce type de vue, on affiche 2 collections d’objets, avec un lien hiérarchique entre les 2.

Ex : supposons que nous souhaitions afficher le détail des articles achetés pour l’achat sélectionné :



Pour cela, il nous faut ajouter un membre de type liste à la classe Achat :

public class Achat

{

public int Id { get; set; }

public DateTime Date { get; set; }  
 ...

public List<Article> Articles { get; set; }

}

Nous pouvons ensuite définir une liste d’objets Achat comme contexte de la fenêtre.

Dans le code xaml de la fenêtre, les listes maître et détail seront affichées comme ceci :

<!-- Maître -->

<Label Content="Achats"/>

<ListView ItemsSource="{Binding}" IsSynchronizedWithCurrentItem="True">

<ListView.View>

<GridView AllowsColumnReorder="true">

<GridViewColumn Header="Id" Width="50"

DisplayMemberBinding="{Binding Path=Id}"/>

<GridViewColumn Header="Montant" Width="100"

DisplayMemberBinding="{Binding Path=Montant, StringFormat='\{0:C\}'}"/>

<GridViewColumn Header="Date"

DisplayMemberBinding="{Binding Path=Date, StringFormat='\{0:g\}'}"/>

</GridView>

</ListView.View>

</ListView>

<!-- Détail -->

<StackPanel Orientation="Horizontal" Margin="0,20,0,5">

<Label Content="Articles de l'achat N°"/>

<TextBlock Text="{Binding Path=Id}" VerticalAlignment="Center"/>

</StackPanel>

<ListView ItemsSource="{Binding Path=Articles}">

<ListView.View>

<GridView>

<GridViewColumn Header="Id" DisplayMemberBinding="{Binding Path=Id}"/>

<GridViewColumn Header="Libellé" DisplayMemberBinding="{Binding Path=Libellé}"/>

<GridViewColumn Header="Prix" DisplayMemberBinding="{Binding Path=Prix, StringFormat='\{0:C\}'}"/>

</GridView>

</ListView.View>

</ListView>

Le contexte de la fenêtre est toujours un objet List<Achats>. La ListView du maître l’utilise directement comme source de données, tandis que la ListView du détail utilise la propriété Article**s** de l’élément courant comme source de données.

Pour chaque colonne de la GridView, on spécifie un binding avec une propriété de l’objet source. On notera au passage que sur une GridViewColumn, le binding se fait sur la propriété **DisplayMemberBinding**.

NB/ Dans la propriété StringFormat, comme la chaîne de format commence par une accolade, il faut utiliser le caractère d’échappement « \ ».

## Liaisons avec d’autres sources de données

### Fichier XML

Nous avons vu plus haut comment charger un fichier xml dans une collection d’objets, puis lier cette collection à un contrôle de sélection. WPF fournit un autre mécanisme plus direct, mais aussi plus limité, pour charger et afficher le contenu d’un fichier XML : le XmlDataPRovider.

Voici un exemple de mise en œuvre :

<Window.Resources>

<XmlDataProvider x:Key="xmlAchats" Source="Achats.xml" XPath="Achats/Achat"/>

<local:StringToDateConverter x:Key="StringToDateConverter"/>

</Window.Resources>

<Grid>

<ListView ItemsSource="{Binding Source={StaticResource xmlAchats}}">

<ListView.View>

<GridView>

<GridViewColumn Header="Id"  
 DisplayMemberBinding="{Binding XPath=@Id}"/>

<GridViewColumn Header="Date"

DisplayMemberBinding="{Binding XPath=@Date,

Converter={StaticResource StringToDateConverter}}"/>

<GridViewColumn Header="Nb Articles"

DisplayMemberBinding="{Binding XPath=@NbArticles}"/>

<GridViewColumn Header="Montant"

DisplayMemberBinding="{Binding XPath=@Montant}"/>

</GridView>

</ListView.View>

</ListView>  
</Grid>

Limitations :

* Le binding est en mode OneWay uniquement ; on ne peut pas modifier le fichier xml source
* Comme on ne peut pas spécifier d’attributs de sérialisation, toutes les données xaml sont interprétées comme des chaînes. Si elles contiennent des dates, nombres, booléens…, il faut utiliser des convertisseurs pour les interpréter correctement.

### Base de données

ADO.NET est l’API couramment utilisée pour interagir avec les bases de données relationnelles. Elle fournit entre autres les classes Dataset et DataTable pour stocker en mémoire les données récupérées ou à envoyer à la base. Un Dataset est un ensemble de DataTable.

La liaison de données avec une DataTable est strictement identique à la liaison avec une collection :

<ListBox ItemsSource="{Binding}" DisplayMemberPath="Libellé"/>

Le contexte est la DataTable, et la propriété affichée est une colonne de la table.

La liaison de données avec un DataSet n’est pas très différente :

<ListBox ItemsSource="{Binding Path=Articles}" DisplayMemberPath="Libellé"/>

Le contexte est le Dataset. Comme il contient plusieurs tables, il faut spécifier le nom de la table dans la source du binding.

## DataTemplate

Lorsqu’on relie une collection d’objets à un contrôle ListBox ou ComboBox, on doit spécifier la propriété de l’item à afficher au moyen de la propriété DisplayMemberPath du contrôle :

|  |  |
| --- | --- |
|  | <ListBox Name="lstAchats"  ItemsSource="{Binding}"  DisplayMemberPath="Date"  ItemStringFormat="f"/> |

La liaison se limite donc à afficher le texte d’une seule propriété de l’item.

On pourrait créer une nouvelle propriété sur la classe de l’item, qui rassemble les valeurs de plusieurs autres propriétés. Mais on serait toujours limité à l’affichage d’un texte, sans mise en forme possible.

Le **DataTemplate** lève ces limitations. C’est un morceau de code xaml qui décrit comment les données de chaque item doivent être affichées. En particulier, il permet de :

* Lier plusieurs propriétés de l’item de collection
* Appliquer des mises en forme

Voici un exemple de mise en œuvre d’un DataTemplate :

<ListBox HorizontalContentAlignment="Stretch"

ItemsSource="{Binding}">

<ListBox.ItemTemplate>

<DataTemplate>

<DockPanel>

<TextBlock FontWeight="Bold"

Text="{Binding Path=Id, StringFormat='#{0}'}"/>

<TextBlock DockPanel.Dock="Right" Foreground="Blue"

Text="{Binding Path=Montant, StringFormat='\{0:C\}'}"/>

<TextBlock HorizontalAlignment="Center" FontFamily="Consolas"

Text="{Binding Path=Date, StringFormat='\{0:g\}'}"/>

</DockPanel>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

On utilise l’élément DataTemplate comme valeur de la propriété ItemTemplate de la ListBox. Il permet ici d’afficher les 3 propriétés Id, Date et Montant de chaque commande.

A l’intérieur du DataTemplate, on est libre de faire toutes les mises en forme qu’on souhaite.

NB/ Sur l’élément ListBox en lui-même, il est généralement préférable de définir l’alignement horizontal du contenu à la valeur Stretch pour avoir plus de liberté dans la mise en forme.

Voici le visuel obtenu (à gauche avec ListBox, à droite avec un template similaire sur une ComboBox)

|  |  |
| --- | --- |
|  |  |

Le DataTemplate peut aussi être défini en tant que ressource au niveau de la fenêtre. La ListBox y fera alors référence à partir de sa clé, comme ceci :

<ListBox Name="lstAchats"

HorizontalContentAlignment="Stretch"

ItemsSource="{Binding}"

ItemTemplate="{StaticResource TemplateAchat}">

**DataTemplate pour des données hiérarchiques**

Des données hiérarchiques peuvent être présentées sous forme d’un maître-détail tel nous l’avons vu plus haut. Toutefois, si le nombre de niveaux est supérieur à 2, une représentation sous forme d’arbre est plus adaptée. Dans ce cas, on utilisera un objet [HierarchicalDataTemplate](https://msdn.microsoft.com/en-us/library/system.windows.hierarchicaldatatemplate(v=vs.110).aspx) pour décrire cette représentation.

**DataTemplate sur un ContentControl**

Le DataTemplate est très utilisé avec les ItemsControl, car il permet de définir un modèle commun pour tous les éléments affichés par le contrôle.

L’utilisation d’un DataTemplate sur un ContentControl à moins d’intérêt, du fait que ce dernier ne peut afficher qu’un élément de la collection à laquelle il est éventuellement lié. Le DataTemplate est affecté au contrôle au moyen de sa propriété ContentTemplate. Ex :

<ContentControl ContentTemplate="{StaticResource MonTemplate}"/>

NB/ Un ContentControl a 2 templates, qu’il ne faut pas confondre :

* Celui qui s’applique au contrôle lui-même, accessible par la propriété Template, de type ControlTemplate (héritée de Control). Il décrit la structure et le comportement du contrôle lui-même, et non de son contenu
* Celui qui décrit la structure de son contenu (propriété ContentTemplate, de type DataTemplate)

## Navigation, tri, filtrage et regroupement

### Navigation

Un scénario courant est d’afficher les éléments d’une collection un par un, et d’implémenter une navigation.

Le visuel ci-dessous est un exemple d’interface minimaliste pour naviguer dans une collection :



La barre d’outils contient des boutons pour naviguer parmi les éléments de la collection, c’est-à-dire aller au premier, au précédent, au suivant ou au dernier élément. L’item courant est affiché en dessous.

Nous allons implémenter cela en reprenant l’exemple de la liste d’achats.  
Commençons par créer le visuel en xaml :

<ToolBar Grid.Row="1" VerticalAlignment="Top" ButtonBase.Click="ToolBarButton\_Click">

<Button Tag="F" Content=" |&lt; "/>

<Button Tag="P" Content=" &lt; "/>

<Button Tag="N" Content=" &gt; "/>

<Button Tag="L" Content=" |&gt; "/>

</ToolBar>

<TextBlock Grid.Row="1" Margin="0,30,0,0" Foreground="Blue">

<TextBlock Text="{Binding Path=Id, StringFormat='Achat N°{0}'}"/>

<TextBlock Text="{Binding Path=Date, StringFormat=' ({0:g}) : '}"/>

<TextBlock Text="{Binding Path=Montant, StringFormat='\{0:C\}'}"/>

</TextBlock>

Au lieu d’appeler un gestionnaire d’évènement sur chaque bouton, nous utilisons l’évènement attaché ButtonBase.Click sur la Toolbar. L’évènement est émis à chaque clic sur un des boutons de la barre d’outils.

Le gestionnaire ToolBarButton\_Click implémente la navigation dans la collection. Pour savoir dans quelle direction il faudra naviguer, nous stockons l’information dans la propriété Tag de chaque bouton (F=First, P=Previous, N=Next, L=Last).

Pour afficher la commande courante, nous utilisons des bindings sur des TextBlock.

Voyons maintenant le code du gestionnaire d’évènement :

// Clic sur un des boutons de navigation

private void ToolBarButton\_Click(object sender, RoutedEventArgs e)

{

// Récupère la vue associée à la collection des achats

ICollectionView view = CollectionViewSource.GetDefaultView(Achats);

// Récupère la valeur de la propriété Tag du contrôle qui a émis l'évènement

string dir = ((FrameworkElement)e.Source).Tag.ToString();

// Navigue dans la collection selon la direction souhaitée

if (dir == "F")

view.MoveCurrentToFirst(); // premier élément

else if (dir == "P" && !view.IsCurrentBeforeFirst)

view.MoveCurrentToPrevious(); // élément précédent

else if (dir == "N" && !view.IsCurrentAfterLast)

view.MoveCurrentToNext(); // élément suivant

else if (dir == "L")

view.MoveCurrentToLast(); // dernier élément

}

Pour comprendre ce code, il faut savoir que lorsqu’on crée un contexte avec un objet de type collection, WPF crée automatiquement un objet de type **ICollectionView** qui implémente le mécanisme de navigation dans les items de la liste. Tout ce que nous avons à faire, c’est donc de récupérer une référence à cet objet et d’appeler ses méthodes de navigation !

**Les vues de collections**

Les principales méthodes définies par l’interface [ICollectionView](https://msdn.microsoft.com/fr-fr/library/system.componentmodel.icollectionview(v=vs.110).aspx) sont :

* Contains(object) : Retourne une valeur qui indique si un élément donné appartient à cette vue de collection.
* MoveCurrentTo(object) : Définit l'élément spécifié comme devant être le CurrentItem de la vue.
* MoveCurrentToFirst() : Définit le premier élément de la vue comme CurrentItem.
* MoveCurrentToLast() : Définit le dernier élément de la vue comme CurrentItem.
* MoveCurrentToNext() : Définit l'élément après CurrentItem dans la vue comme CurrentItem.
* MoveCurrentToPosition(Int32) : Définit l'élément situé à l'index spécifié comme CurrentItem
* MoveCurrentToPrevious() : Définit l'élément avant CurrentItem dans la vue comme CurrentItem.

L’interface ICollectionView est implémentée par la classe CollectionView, qui est la classe de base pour BindingListCollectionView, ListCollectionView, et ItemCollection.

Le diagramme de classes ci-dessous présente les classes et interfaces qui interviennent dans la liaison entre un contrôle de type ItemsControl et une collection d’objets :



Les items du contrôle WPF sont stockés dans une collection de type ItemCollection.

La collection d’objets qui est source de données implémente une des interfaces représentées en vert.

Le lien entre les 2 collections est fait par une classe qui implémente ICollectionView, et qu’on peut appeler « vue » (représentée en bleu)

Le diagramme montre quel type de vue est utilisé selon l’interface implémentée par la collection d’objets avec laquelle on fait la liaison. Par exemple, si on fait une liaison avec une collection qui implémente Ilist, mais pas IBindingList, c’est une vue de type ListCollectionView qui est utilisée.

/!\ Ne pas confondre la vue de collection (ICollectionView), qui est indépendante de tout contrôle WPF, et la vue associée à une ListView (comme la GridView)

En plus de la vue fournie par défaut, on peut créer soi-même d’autres vues de collections, soit par le code, soit en XAML.

ICollectionView ne fournit pas seulement des méthodes de navigation. Elle possède également des propriétés pour le tri, le filtrage et le regroupement des données, comme nous allons le voir maintenant.

### Tri

ICollectionView possède une propriété **SortDescriptions**, qui est une collection d’objets SortDescription. Cet objet décrit la propriété et le sens selon lesquels on veut trier les données. Si on veut trier les données selon plusieurs propriétés, il suffit d’ajouter plusieurs descriptions de tri dans la collection SortDescription, dans l’ordre souhaité.

Ajoutons des ComboBox pour sélectionner la propriété et le sens du tri de notre liste d’achats :



<WrapPanel>

<Label Content="Trier la liste par :"/>

<ComboBox Name="cbChampTri" Margin="5,0">

<ComboBoxItem Content="Date" IsSelected="True"/>

<ComboBoxItem Content="Montant"/>

</ComboBox>

<ComboBox Name="cbSensTri" Margin="10,0">

<ComboBoxItem Content=" &#x25b2;" IsSelected="True"/>

<ComboBoxItem Content=" &#x25bc;"/>

</ComboBox>

<Button Name="btnTrier" Content="OK" Width="50" Click="btnTrier\_Click"/>

</WrapPanel>

private void btnTrier\_Click(object sender, RoutedEventArgs e)

{

// Applique le tri à la collection d'achats

ICollectionView view = CollectionViewSource.GetDefaultView(Achats);

view.SortDescriptions.Clear();

var sens = cbSensTri.SelectedIndex == 0 ? ListSortDirection.Ascending :   
 ListSortDirection.Descending;

var tri = new SortDescription(cbChampTri.Text, sens);

view.SortDescriptions.Add(tri);

}

Pas de difficulté particulière sur ce code. Il faut juste penser à vider la collection SortDescription avant d’appliquer de nouveaux tris.

On peut aussi définir un tri personnalisé en utilisant la propriété **CustomSort** de ICollectionView. Cette propriété est de type IComparer, qui est une interface définissant une méthode Compare.

### Filtrage

ICollectionView possède une propriété **Filter**, de type prédicat. Ce prédicat représente une fonction qui prend en entrée un élément de la collection à filtrer, et qui renvoie un booléen indiquant si l’élément doit être retenu ou non.

Ajoutons l’interface suivante pour filtrer notre collection d’achats d’après le montant :



<WrapPanel Grid.Row="3">

<Label Content="Filtrer les achats dont le montant dépasse"/>

<TextBox Name="txtMontant" Width="50" Margin="10,0" VerticalContentAlignment="Center"/>

<Button Name="btnFiltrer" Content="OK" Width="50" Click="btnFiltrer\_Click"/>

</WrapPanel>

private void btnFiltrer\_Click(object sender, RoutedEventArgs e)

{

// Applique le filtre à la collection d'achats

CollectionViewSource.GetDefaultView(Achats).Filter = FiltrerSurMontant;

}

// Filtre

private bool FiltrerSurMontant(object o)

{

decimal seuil;

if (decimal.TryParse(txtMontant.Text, out seuil))

return ((Achat)o).Montant >= seuil;

return true;

}

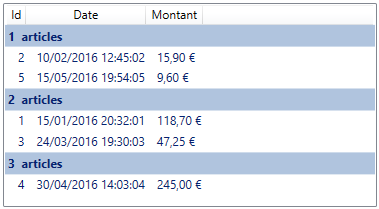
Lorsqu’on clique sur le bouton OK, la fonction FiltrerSurMontant est appelée automatiquement sur chaque élément de la collection. Le type « object » de son paramètre est imposée par la propriété Filter ; il faut donc transtyper l’objet reçu en paramètre en « Achat », puis retourner vrai si le montant de l’achat satisfait à la condition choisie.

### Regroupement

Le regroupement est assez semblable au tri du point de vue de l’API :

Il faut ajouter des objets PropertyGroupDescription à la collection GroupDescriptions de la ICollectionView.

La présentation visuelle nécessite toutefois un peu plus de travail. En effet, si on veut que les groupes se distinguent visuellement, comme ceci…



…il faut définir un modèle de données (DataTemplate) pour l’en-tête de groupe. Voici comment faire :

<ListView Grid.Row="4" ItemsSource="{Binding}">

<ListView.GroupStyle>

<GroupStyle>

<GroupStyle.HeaderTemplate>

<DataTemplate>

<TextBlock Background="LightSteelBlue" FontWeight="Bold" Padding="3">

<TextBlock Text="{Binding Path=Name}"/>

<TextBlock Text=" articles"/>

</TextBlock>

</DataTemplate>

</GroupStyle.HeaderTemplate>

</GroupStyle>

</ListView.GroupStyle>

<ListView.View>

<GridView>

<GridViewColumn Header="Id" DisplayMemberBinding="{Binding Path=Id}"/>

<GridViewColumn Header="Date" DisplayMemberBinding="{Binding Path=Date}"/>

<GridViewColumn Header="Montant" DisplayMemberBinding="{Binding Path=Montant, StringFormat='\{0:C\}'}"/>

</GridView>

</ListView.View>

</ListView>

NB/ Le DataTemplate pourrait être défini en tant que ressource.

/!\ Le binding de la donnée affichée dans l’en-tête de groupe se fait non pas avec le nom de la propriété de regroupement, mais avec celui de la propriété du PropertyGroupDescription, à savoir « Name ».

Enfin, pour regrouper les achats selon le nombre d’articles, et afficher les groupes dans l’ordre croissant, nous ajoutons le code suivant dans le constructeur de la fenêtre :

ICollectionView view = CollectionViewSource.GetDefaultView(Achats);

view.SortDescriptions.Add(new SortDescription("NbArticles", ListSortDirection.Ascending));

view.GroupDescriptions.Add(new PropertyGroupDescription("NbArticles"));

On trie tout d’abord la vue selon la propriété de regroupement, puis on applique un regroupement en ajoutant un objet PropertyGroupDescrption à la liste GroupDescriptions.

## Débogage

Les liaisons de données sont évaluées à l’exécution et ne lèvent pas d’exceptions quand elles échouent ; c’est pourquoi elles ne sont pas toujours faciles à déboguer.

On constate une erreur de binding par le fait que la ou les donnée(s) correspondante(s) ne s’affiche(nt) pas dans l’application. On peut donc cibler au moins un élément WPF particulier comme point de départ de nos investigations.

Voyons les techniques utilisables pour identifier l’erreur.

### Utiliser un convertisseur comme point d’entrée de débogage

Comment s’assurer qu’une expression de binding est correcte ?

Si l’expression n’est pas correcte, le binding ne sera même pas évalué. Si elle est correcte, le binding sera au moins évalué, c’est-à-dire qu’on doit pouvoir l’espionner en mode debug.

Cependant, on n’a par défaut aucun endroit dans le code C# pour poser un point d’arrêt, puisque le binding est décrit par le xaml. L’astuce consiste donc à ajouter un convertisseur au binding, Le convertisseur étant décrit par le code, il fournit un point d’entrée pour poser le point d’arrêt.

Voici le code de la classe du convertisseur :

using System;

using System.Diagnostics;

using System.Windows.Data;

using System.Globalization;

namespace LiaisonDonnées

{

// Convertisseur fictif pour pouvoir déboguer les binding

public class DebugDummyConverter : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

Debugger.Break();

return value;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

Debugger.Break();

return value;

}

}

}

Debugger.Break a le même effet que de poser un point d’arrêt dans Visual Studio.

Le convertisseur peut être référencé comme ressource de l’application, de façon à être utilisable dans n’importe quelle fenêtre de l’application :

<Application x:Class="LiaisonDonnées.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:LiaisonDonnées"

StartupUri="MainWindow.xaml">

<Application.Resources>

<local:DebugDummyConverter x:Key="DebugDummyConverter"/>

</Application.Resources>

</Application>

Utilisons-le maintenant dans une expression de binding dans une des fenêtres de l’application :

<TextBlock Text="{Binding Path=NbArticles, Converter={StaticResource DebugDummyConverter}}"/>

En faisant, cela, si l’expression de binding est correcte, dès que le binding sera évalué, Visual Studio interrompra l’exécution de l’application et se mettra en attente sur la ligne Debugger.break. On pourra en profiter pour regarder la valeur renvoyée par le binding.

Si cela ne se produit pas, c’est que la syntaxe du binding est erronée. On peut alors investiguer plus au moyen du panneau de sortie.

### Utiliser le panneau de sortie

Une erreur courante dans une expression de binding est de saisir un nom de propriété incorrect. Cette erreur est assez facile à trouver, car il y a une trace suffisamment précise dans l’onglet de sortie de Visual Studio (Menu Affichage \ Sortie ou CTRL+ALT+O).

Exemple : supposons qu’on ait oublié le s à la fin du nom de la propriété NbArticles dans le binding suivant :

<StackPanel Name="StackPanel1" Margin="5">

...

<TextBlock Text="{Binding Path=NbArticle, StringFormat='Nombre articles : {0}'}"/>

</StackPanel>

Visual Studio affiche le message suivant dans le panneau Output :

System.Windows.Data Error: 40 : BindingExpression path error: 'NbArticle' property not found on 'object' ''List`1' (HashCode=5432205)'. BindingExpression:Path=NbArticle; DataItem='List`1' (HashCode=5432205); target element is 'TextBlock' (Name=''); target property is 'Text' (type 'String')

### Ajuster le niveau de trace

Supposons maintenant que le contexte de la liaison ne soit pas correctement affecté par le code C#. Comme il peut être affecté dynamiquement, il n’y a pas d’erreur ni à la compilation, ni à l’exécution. Et cette fois, Visual Studio n’affiche aucun message dans le panneau de sortie. Mais on peut forcer WPF à afficher plus d’informations sur les bindings, en passant le paramètre TraceLevel à la valeur High dans le code xaml, comme ceci :

<Window x:Class="LiaisonDonnées.MaitreDetail"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:LiaisonDonnées"

xmlns:diag="clr-namespace:System.Diagnostics;assembly=WindowsBase"

...

<TextBlock Text="{Binding Path=NbArticles, diag:PresentationTraceSources.TraceLevel=High}"/>

Résultat : dans le panneau de sortie, on voit un ensemble de lignes ressemblant à ceci :

System.Windows.Data Warning: 56 : Created BindingExpression (hash=21644229) for Binding (hash=36052245)

System.Windows.Data Warning: 58 : Path: 'NbArticles'

System.Windows.Data Warning: 60 : BindingExpression (hash=21644229): Default mode resolved to OneWay

System.Windows.Data Warning: 61 : BindingExpression (hash=21644229): Default update trigger resolved to PropertyChanged

System.Windows.Data Warning: 62 : BindingExpression (hash=21644229): Attach to System.Windows.Controls.TextBlock.Text (hash=63658128)

System.Windows.Data Warning: 67 : BindingExpression (hash=21644229): Resolving source

System.Windows.Data Warning: 70 : BindingExpression (hash=21644229): Found data context element: TextBlock (hash=63658128) (OK)

System.Windows.Data Warning: 71 : BindingExpression (hash=21644229): DataContext is null

System.Windows.Data Warning: 65 : BindingExpression (hash=21644229): Resolve source deferred

System.Windows.Data Warning: 67 : BindingExpression (hash=21644229): Resolving source

...

System.Windows.Data Warning: 67 : BindingExpression (hash=21644229): Resolving source

System.Windows.Data Warning: 70 : BindingExpression (hash=21644229): Found data context element: TextBlock (hash=63658128) (OK)

System.Windows.Data Warning: 71 : BindingExpression (hash=21644229): DataContext is null

System.Windows.Data Warning: 67 : BindingExpression (hash=21644229): Resolving source (last chance)

System.Windows.Data Warning: 70 : BindingExpression (hash=21644229): Found data context element: TextBlock (hash=63658128) (OK)

System.Windows.Data Warning: 78 : BindingExpression (hash=21644229): Activate with root item <null>

System.Windows.Data Warning: 106 : BindingExpression (hash=21644229): Item at level 0 is null - no accessor

System.Windows.Data Warning: 80 : BindingExpression (hash=21644229): TransferValue - got raw value {DependencyProperty.UnsetValue}

System.Windows.Data Warning: 88 : BindingExpression (hash=21644229): TransferValue - using fallback/default value ''

System.Windows.Data Warning: 89 : BindingExpression (hash=21644229): TransferValue - using final value ''

On voit que WPF cherche la source du binding pour le TextBlock, mais aboutit plusieurs fois au résultat « DataContext is null » et finit par utiliser la valeur **DependencyProperty.UnsetValue**, qui se traduit par une chaîne vide.

NB/ Comme la sortie est très verbeuse, et pas facile à exploiter dans VS, on peut par exemple copier/coller son contenu dans un éditeur de texte et rechercher la valeur UnsetValue avec la fonction de recherche de l’éditeur, pour aller plus vite.

# Conversion, validation et notification des changements

Nous allons voir dans ce chapitre 3 opérations essentielles dans la création d’une fenêtre de saisie réelle.

## Conversion

L’affichage des données nécessite souvent un formatage ou une transformation, que nous regroupons sous le terme de **conversion**.

Les besoins de conversions les plus communs sont :

* Adapter la forme textuelle des dates, des prix, des divers types de numéros (téléphone, identifiant) …etc.
* Modifier de façon conditionnelle la couleur de fond, la police, la langue…etc.
* Associer un autre objet à la donnée initiale. Par exemple, afficher l’image correspondant à un chemin, afficher le libellé correspondant à un code…etc.

Nous avons déjà vu plus haut une façon simple d’appliquer un format aux nombres et dates dans une liaison de données, à l’aide de la propriété StringFormat. Nous allons voir ici aborder la notion de convertisseur, qui couvre de façon générale tous les besoins de conversion.

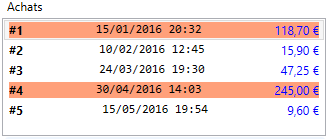
### Créer un convertisseur

Un convertisseur est un objet qui implémente l’interface **IValueConverter**. Cette interface déclare 2 méthodes :

* Convert : pour convertir les données vers l’affichage
* ConvertBack : pour convertir la saisie vers les données

Souvent, on peut se passer d’implémenter ConvertBack. Cependant, si on utilise des binding TwoWay, à moins d’être absolument sûr que ConvertBack ne sera pas appelée, il faut l’implémenter.

**Exemple** : nous allons créer un convertisseur qui affiche sur fond coloré tous les items d’une liste d’achats dont le montant est ≥ 100 €, come illustré par la capture ci-dessous :



La couleur de fond d’un conteneur est appliquée au moyen d’un objet SolidColorBrush. Le montant d’un achat est une valeur décimale. Nous avons donc besoin d’un convertisseur Decimal 🡪 SolidColorBrush, et uniquement dans ce sens.

Voici le code de ce convertisseur :

// Convertisseur de Decimal en Brush avec seuil

public class DecimalToColorBrushConverter : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

decimal d = (decimal)value;

decimal seuil = (decimal)parameter;

Color c = (d >= seuil ? Colors.LightSalmon : Colors.White);

return new SolidColorBrush(c);

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

throw new NotImplementedException();

}

}

**value** est la valeur à convertir (ici, le montant de l’achat). Elle est passée sous forme d’objet, c’est pourquoi nous devons faire un cast.

**parameter** est un paramètre de conversion. Ici, c’est le montant seuil à partir duquel on souhaite mettre l’achat sur fond coloré.

**targetType** est le type de la propriété qui a appelé le convertisseur. On pourrait s’en servir pour déterminer dynamiquement le type de l’objet à renvoyer, afin de faire un convertisseur plus générique.

Le code de la méthode Convert crée et renvoie un objet SolidColorBrush (dérivé de Brush) avec une couleur LightSalmon si la valeur décimale est ≥100, et White sinon.

Comme nous sommes sûr que ConvertBack ne sera pas appelée, nous ne l’implémentons pas.

### Utiliser un convertisseur dans un binding

Utilisons le convertisseur créé précédemment dans le code xaml :

<Window x:Class="LiaisonDonnées.LiaisonObjets"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:LiaisonDonnées"

xmlns:sys="clr-namespace:System;assembly=mscorlib"

Title="Liaison avec objets" Height="450" Width="350">

<Window.Resources>

<local:DecimalToColorBrushConverter x:Key="DecimalToColorBrushConverter"/>  
 <sys:Decimal x:Key="MontantSeuil">100</sys:Decimal>

</Window.Resources>

...

<ListBox.ItemTemplate>

<DataTemplate>

<DockPanel Background="{Binding Path=Montant,

Converter={StaticResource DecimalToColorBrushConverter},

ConverterParameter={StaticResource MontantSeuil}}">

...

</DockPanel>

</DataTemplate>

</ListBox.ItemTemplate>

On commence par référencer le convertisseur en tant que ressource de la fenêtre.

On crée également une ressource pour la valeur du seuil, afin de la centraliser.

Puis, dans le DataTemplate de la liste des achats, on fait un binding de la propriété Background (de type Brush) du conteneur d’item sur le montant de l’achat, en appliquant le convertisseur au moyen de la ressource, et en lui passant la valeur de seuil en paramètre.

## Validation

Dans une application réelle, l’utilisateur saisit des données, et l’application doit s’assurer qu’elles sont valides avant de les enregistrer. Cette validation peut être déclenchée :

* A chaque modification d’un champ de saisie (zone de texte ou autre)
* Lors du passage à un autre item si on manipule une collection.
* Par clic sur un bouton d’enregistrement ou de validation
* …

La validation passe par l’exécution de **règles de validation**.

### Règles de validation

Le .net framework fournit la classe abstraite **ValidationRule** pour modéliser les règles de validation. Il fournit également 2 classes dérivées concrètes ExceptionValidationRule et DataErrorValidationRule, et on peut bien entendu créer nos propres classes dérivées.

/!\ Le mécanisme de validation basé sur ces classes est peu compatible avec le modèle d’architecture MVVM. Dans une application MVVM, on utilise d’autres techniques avancées qui sortent du cadre de ce cours.

Les règles de validations s’appliquent sur un Binding, en les ajoutant à sa collection ValidationRules.

Exemple :

<TextBox>

<TextBox.Text>

<Binding Path="Prio">

<Binding.ValidationRules>

<ExceptionValidationRule/>

</Binding.ValidationRules>

</Binding>

</TextBox.Text>

</TextBox>

Ici, on ajoute une règle de validation de type ExceptionValidationRule au binding entre TextBox.Text et une propriété Prio du contexte. La règle sera évaluée lorsque le binding Prio 🡪 Text sera évalué, c’est-à-dire lorsque la TextBox perdra le focus.

Rappel : par défaut, dans une liaison, la mise à jour de la propriété source à partir de la cible a lieu :

* En sortie de zone pour tous les contrôles qui ont une propriété Text
* Dès que la valeur change pour les autres contrôles

**ExceptionValidationRule** est une règle de validation prédéfinie, qui a la particularité de générer une erreur de validation pour chaque exception déclenchée au cours de la mise à jour de la propriété source du binding.

Dans notre cas, toute exception levée dans la partie « set » de la propriété Prio sera interprétée comme une erreur de validation.

Dans le code de la propriété Prio, on peut donc utiliser ce mécanisme pour faire des contrôles de validité :

public int Prio

{

get { return \_prio; }

set {

if (value < 1 || value > 3)

throw new ArgumentException("La valeur doit être comprise entre 1 et 3");

\_prio = value;

}

}

Ici, on lève une exception si la valeur saisie n’est pas comprise entre 1 et 3.

**DataErrorValidationRule** est une règle de validation prédéfinie, qui vérifie les erreurs levées par l’objet source du binding, si celui-ci implémente l’interface IDataErrorInfo.

NB/ depuis .net 3.5, il existe des syntaxes plus simples pour mettre en œuvre les règles de validation prédéfinies. On peut ainsi écrire simplement :

<TextBox Text="{Binding Path=Prio, ValidatesOnExceptions=True}"/>  
ou

<TextBox Text="{Binding Path=Prio, ValidatesOnDataErrors=True}"/>

La syntaxe générale présentée plus haut reste toutefois indispensable pour mettre en œuvre des règles de validation personnalisées.

**Règles de validation personnalisées**

Généralement, on préférera créer nos propres règles de validation dérivées de ValidationRule. Voici par exemple une règle simple qui vérifie qu’un champ est bien renseigné :

public class RegleChampObligatoire : ValidationRule

{

public override ValidationResult Validate(object value, System.Globalization.CultureInfo cultureInfo)

{

if (String.IsNullOrWhiteSpace(value.ToString()))

return new ValidationResult(false, "La valeur est obligatoire");

return ValidationResult.ValidResult;

}

}

La seule chose à faire est de redéfinir la méthode Validate et de renvoyer un objet ValidationResult.

Cette règle sera mise en œuvre de la façon suivante dans le xaml :

### Processus de validation

Lors de l’affectation d’une règle de validation à un binding, on peut spécifier la propriété ValidationStep, comme le montre cet exemple :

<TextBox>

<TextBox.Text>

<Binding Path="Prio">

<Binding.ValidationRules>

<local:RegleChampObligatoire ValidationStep="RawProposedValue"/>

</Binding.ValidationRules>

</Binding>

</TextBox.Text>

</TextBox>

Cette propriété définit à quel moment est exécutée la méthode Validate de la règle de gestion. Elle peut prendre 4 valeurs : RawProposedValue, ConvertedProposedValue, UpdatedValue, CommitedValue.

Le moteur de binding valide une zone de saisie selon le processus suivant :

* Il exécute les règles dont ValidationStep=RawProposedValue
* Il appelle le convertisseur, s'il y en a un
* Si la conversion réussit, il exécute les règles dont ValidationStep=ConvertedProposedValue
* Il affecte la propriété source
* Il exécute les règles dont ValidationStep=UpdatedValue. Ceci inclut les règles de type DataErrorValidationRule (les règles personnalisées sont jouées en premier)
* Il exécute les règles dont ValidationStep=CommitedValue

A chaque étape, dès qu'une règle renvoie une erreur, ou qu'une autre erreur survient, le processus est arrêté.

### Visualisation des erreurs de validation

Par défaut, WPF signale une erreur de validation en laissant le focus dans la zone de saisie, et en encadrant cette dernière en rouge, sans message d’erreur. Voici le visuel correspondant à l’exemple précédent :



Il est cependant possible de personnaliser ce visuel en appliquant un style à l’élément WPF concerné. Le code ci-dessous montre comment afficher les détails de l’erreur en info-bulle :

<Style x:Key="textBoxInError" TargetType="TextBox">

<Style.Triggers>

<Trigger Property="Validation.HasError" Value="true">

<Setter Property="ToolTip"

Value="{Binding RelativeSource={x:Static RelativeSource.Self},

Path=(Validation.Errors)[0].ErrorContent}"/>

</Trigger>

</Style.Triggers>

</Style>

...

<TextBox Style="{StaticResource textBoxInError}"

Text="{Binding Path=Prio, ValidatesOnExceptions=True}"/>

## Notification des changements

Jusqu’ici, dans les divers exemples de code, nous n’avons fait qu’afficher des données initialisées **avant** l’affectation du DataContext. Cependant, si les données sont modifiées par le code après cette affectation, le visuel ne sera par défaut pas mis à jour, même si les bindings sont en mode TwoWay.

Considérons le schéma suivant qui représente un binding TwoWay :

Modif de la valeur par le code

Contexte

Objet WPF

Propriété WPF

Propriété

**Binding TwoWay**

**Cible**

**Source**

Modif de la valeur de la zone de saisie

… le rafraîchissement de la propriété cible WPF (flèche verte) n’est par défaut pas réalisé si on ne fait rien.

Pour que les bindings soient rafraîchis dans le sens Source 🡪 cible, le contexte doit **notifier** toutes les modifications de données faites par le code. Cela comprend :

* Les modifications de valeurs de propriétés sur les objets du contexte
* L’ajout et la suppression d’items dans les collections d’objets du contexte

Pour réaliser ces notifications, le contexte doit implémenter les interfaces [INotifyPropertyChanged](https://msdn.microsoft.com/fr-fr/library/system.componentmodel.inotifypropertychanged(v=vs.110).aspx) et/ou [INotifyCollectionChanged](https://msdn.microsoft.com/fr-fr/library/system.collections.specialized.inotifycollectionchanged(v=vs.110).aspx).

**Implémentation de l’interface INotifyPropertyChanged**

Cette interface comprend un seul élément, qui est l’évènement suivant :

event PropertyChangedEventHandler PropertyChanged;

La notification du changement de valeur d’une propriété est réalisée en déclenchant cet évènement dans l’accesseur set de la propriété. Voici un exemple :

public class Contexte : INotifyPropertyChanged

{

private string \_nom;

public event PropertyChangedEventHandler PropertyChanged;

public string Nom

{

get { return \_nom; }

set

{

\_nom = value;

PropertyChanged(this, new PropertyChangedEventArgs("Nom"));

}

}

...

/!\ la chaîne passée au constructeur de PropertyChangedEventArgs doit toujours correspondre au nom de la propriété dont on souhaite notifier les modifications.

NB/ PropertyChangedEventHandler est un type délégué dont la signature est la suivante :

public delegate void PropertyChangedEventHandler(object sender,

PropertyChangedEventArgs e)

* Le premier argument est l’objet qui a déclenché l’évènement (le contexte ou un de ses objets)
* Le second argument est un objet PropertyChangedEventArgs, qui décrit la propriété qui a changé. Cet objet ne possède qu’une seule propriété de type chaîne, nommée PropertyName.

**Implémentation de l’interface INotifyCollectionChanged**

Cette interface comprend un seul élément qui est l’évènement suivant :

event NotifyCollectionChangedEventHandler CollectionChanged

La notification de l’ajout ou de la suppression d’un élément dans la collection est réalisée en déclenchant cet évènement au moment de l’ajout ou de la suppression.

Le framework .net fournit cependant un type de collection tout fait qui implémente cette interface : **ObservableCollection**

Si le contexte est ou contient une collection d’objets, il suffit donc que celle-ci soit de type ObservableCollection pour que les ajouts et suppressions d’items soient notifiés au visuel.

Exemple :

public ObservableCollection<Tache> Taches { get; private set; }

# Les évènements routés

## Présentation

Un événement routé est un type d'événement qui peut être envoyé successivement par plusieurs éléments WPF d’un arbre visuel, plutôt que seulement par l’élément source qui a déclenché l'événement.

/!\ La sémantique est importante: l’élément source est celui qui déclenche (= émet initialement = lève) l’évènement. Les autres éléments peuvent aussi émettre (=envoyer) l’évènement, sans en être la source. On dit que l’évènement se propage ou est routé dans l’arbre visuel.

Il y a 3 stratégies de routages possibles :

* **Routage direct** : seul l’élément source peut émettre l’évènement
* **Routage montant** : l’élément source émet l’évènement en premier. Puis les éléments parents l’émettent à leur tour successivement en remontant l’arbre visuel, jusqu’à la racine.
* **Routage descendant**: les éléments parents émettent l’évènement en premier, en partant de la racine de l’arbre visuel, jusqu’à l’élément source de l’évènement.

Chaque évènement à une seule stratégie de routage, non modifiable. On peut donc parler « d’évènement montant » ou « d’évènement descendant ».

[Page MSDN](https://msdn.microsoft.com/fr-fr/library/ms742806(v=vs.110).aspx) de vue d’ensemble des évènements routés.

**Paires d’évènements**

Certains évènements routés vont par paire : un descendant + un montant

C’est le cas de la plupart des évènements liés à la saisie.  
Exemples : PreviewMouseDown + MouseDown, PreviewKeyDown + KeyDown…etc.

L’évènement descendant est toujours préfixé par **Preview,** et émis avant l’évènement montant.

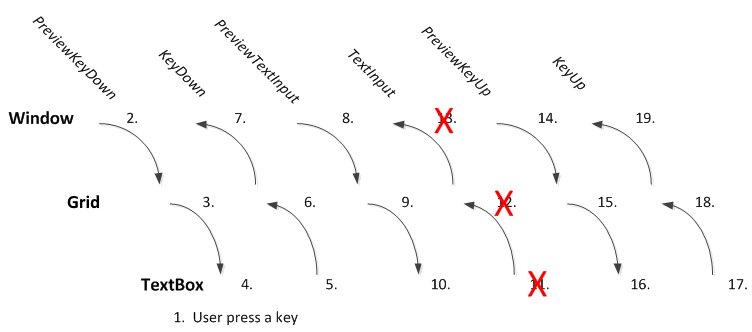
NB/ Certaines actions de saisies impliquent 2 paires d’évènements routés. Exemples :

* L’appui sur une touche du clavier, sur un élément qui a le focus, se décompose en :
  + PreviewKeyDown + KeyDown (enfoncement de la touche)
  + PreviewKeyUp + KeyUp (relâchement de la touche)
* Un clic sur un élément avec le bouton gauche de la souris se décompose en :
  + PreviewMouseLeftButtonDown + MouseLeftButtonDown (enfoncement du bouton)
  + PreviewMouseLeftButtonUp + MouseLeftButtonUp (relâchement du bouton)

**Scénario complet de routage**

Considérons une fenêtre contenant une grille, contenant elle-même une TextBox.

Voici ce qui se passe lorsque l’utilisateur tape un caractère dans la TextBox :



(Image issue de <https://wpf.2000things.com/tag/keydown/>)

* Etape 1 : L’utilisateur enfonce une touche du clavier alors que le focus est sur la TextBox. La TextBox déclenche l’évènement descendant PreviewKeyDown.
* Etape 2 à 4 : les éléments Window, Grid et TextBox envoient successivement l’évènement PreviewKeyDown
* Etapes 5 à 7 : La TextBox déclenche l’évènement montant KeyDown, et les éléments TextBox, Grid et Window envoient successivement cet évènement
* Dans les étapes suivantes, le même processus se répète avec les couples d’évènements PreviewTextInput + TextInput et PreviewKeyup et KeyUp.

NB/ En réalité, dans le cas spécifique de la TextBox, l’évènement TextInput n’est pas déclenché, mais il l’est pour d’autres composants.

Chaque gestionnaire a la possibilité d’interrompre la propagation de l’évènement.

Par exemple, si la propagation est interrompue à l’étape 3 par le gestionnaire de la grille, l’évènement PreviewKeyDown n’est pas envoyé par la TextBox, et l’évènement KeyDown n’est pas déclenché.

## Paramètres

La signature d’un gestionnaire d’évènement routé ressemble à ceci :

private void Gestionnaire(object sender, RoutedEventArgs e)

Le paramètre sender désigne l’objet qui a **envoyé** l’évènement

**RoutedEventArgs** est une classe contenant des informations sur l’évènement routé. Elle a notamment les propriétés suivantes :

* **Source** : objet qui a **déclenché** l’évènement (pas forcément identique à l’objet sender)
* **Handled** : booléen permettant de stopper la propagation de l’évènement. Un évènement routé se propage tant qu'il n'est pas stoppé par le code
* **RoutedEvent** : objet RoutedEvent correspondant à l’évènement (utile dans le cas où le gestionnaire gère plusieurs évènements différents)

NB/ Selon l’évènement, le second paramètre passé au gestionnaire peut être une classe dérivée de RoutedEventArgs, fournissant des informations supplémentaires (ex : KeyEventArgs, MouseEventArgs…)

**Partage de l’instance de RoutedEventArgs :**

Dans les situations suivantes, les gestionnaires partagent la même instance de RoutedEventArgs :

* Un évènement géré par des gestionnaires de plusieurs éléments différents de la même arborescence visuelle
* Deux évènements qui vont par paire (descendant + montant)

Ce partage permet de communiquer entre gestionnaires.

Si la propagation de l’évènement descendant est stoppée, l’évènement montant n’est pas émis.

## Evènement attaché

Au même titre qu’un élément peut avoir des propriétés attachées (ex : Grid.Row), il peut avoir des évènements attachés. Il peut ainsi envoyer un évènement qui n’est pas défini sur sa classe.

Prenons l’exemple d’un DockPanel contenant plusieurs boutons. L’évènement Click est défini sur la classe Button, mais par sur la classe Grid. Il est cependant possible d’écrire ceci :

<DockPanel Grid.Row="4" ButtonBase.Click="Button\_Click">

<Button Name="btnViderSaisie" DockPanel.Dock="Left" Content="Vider les zones de saisie"/>

<Button Name="btnFermer" DockPanel.Dock="Right" Content="Fermer"/>

<Button Name="btnViderLog" Margin="30,0,30,0" Content="Vider le log"/>

</DockPanel>

ButtonBase.Click est un évènement attaché.

Ceci permet de partager facilement un gestionnaire d’évènement entre plusieurs éléments, et facilite ainsi la centralisation du code. Le gestionnaire doit dans ce cas vérifier quel élément est la source de l’évènement pour exécuter le code approprié. Exemple :

private void Button\_Click(object sender, RoutedEventArgs e)

{

switch (((FrameworkElement)e.Source).Name)

{

case "btnViderSaisie":

TextBox1.Clear();

DatePicker1.SelectedDate = null;

break;

case "btnViderLog":

tbLog.Text = string.Empty;

break;

case "btnFermer":

Close();

break;

}

}

On convertit l’objet Source en FrameworkElement pour accéder à sa propriété Name

## Mise en œuvre

L’exemple qui suit illustre la propagation d’un évènement routé montant, et permet de bien comprendre la différence entre les paramètres Sender et Source.

Code XAML :

<Window x:Class="Evts\_Commandes.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Evènements routés" Height="300" Width="500"

KeyDown="KeyDown\_Handler">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="30"/>

<RowDefinition Height="30"/>

<RowDefinition Height="30"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="30"/>

</Grid.RowDefinitions>

<TextBox Name="TextBox1" KeyDown="KeyDown\_Handler"/>

<DatePicker Name="DatePicker1" Grid.Row="2"/>

<ScrollViewer Grid.Row="3" BorderBrush="Blue" BorderThickness="1">

<TextBlock Name="tbLog"/>

</ScrollViewer>

<DockPanel Grid.Row="4" ButtonBase.Click="Button\_Click">

<Button Name="btnViderSaisie" DockPanel.Dock="Left"

Content="Vider les zones de saisie"/>

<Button Name="btnFermer" DockPanel.Dock="Right" Content="Fermer"/>

<Button Name="btnViderLog" Margin="30,0,30,0" Content="Vider le log"/>

</DockPanel>

</Grid>

</Window>

Ce code décrit une fenêtre dont le conteneur principal est une grille, qui contient entre autres une zone de texte et une zone de saisie de date. Nous avons branché un gestionnaire de l’évènement KeyDown (enfoncement d’une touche) sur les éléments Window et TextBox. Voici le code de ce gestionnaire :

private void KeyDown\_Handler(object sender, KeyEventArgs e)

{

tbLog.Text += string.Format("Source : {0}, émetteur : {1}\r\n",

((FrameworkElement)e.Source).Name,

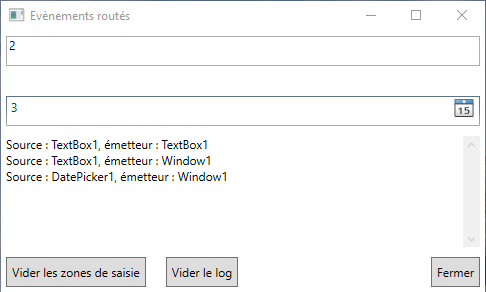
((FrameworkElement)sender).Name);

}

Il affiche dans le bloc de texte les noms des éléments source et émetteur de l’évènement.

**Résultat**

Exécutons ce code et appuyons sur une touche en mettant le focus successivement sur la fenêtre, sur la zone de texte, et sur la zone de saisie de date. Voici le résultat obtenu :



Le 1er appui sur une touche génère la 1ère ligne dans le TextBlock. Comme la fenêtre a le focus, c’est elle la source de l’évènement. Comme elle est reliée directement au gestionnaire, elle est aussi l’émetteur.

Le 2d appui (touche « 2 » dans la zone de texte) appelle 2 fois le gestionnaire :

* La première fois avec la zone de texte comme émetteur
* La seconde fois avec la fenêtre comme émetteur.

Dans les 2 cas, la source de l’évènement reste la zone de saisie, car c’est sur elle qu’on a enfoncé la touche.

Le 3ème appui (touche 3 dans la zone de saisie de date) génère la dernière ligne, qui indique que la source est le DatePicker, et l’envoyeur la fenêtre. En effet, l’évènement se propage en remontant l’arbre visuel jusqu’à la fenêtre, sur laquelle un gestionnaire est branché.

### Arrêt de la propagation

Complétons le code précédent en ajoutant la case à cocher suivante dans le xaml :

<Window x:Class="Evts\_Commandes.MainWindow"

...

KeyDown="KeyDown\_Handler">

<Grid Margin="5">

...

<TextBox Name="TextBox1" KeyDown="KeyDown\_Handler"/>

<CheckBox Name="chkStop" Grid.Row="1" IsChecked="True"

Content="Stopper la propagation de l'évènement KeyDown"/>

<DatePicker Name="DatePicker1" Grid.Row="2"/>

Utilisons cette case pour activer ou désactiver la propagation de l’évènement KeyDown. Pour cela, modifions le code du gestionnaire comme ceci :

private void KeyDown\_Handler(object sender, KeyEventArgs e)

{

e.Handled = (bool)chkStop.IsChecked;

tbLog.Text += string.Format("Source : {0}, émetteur : {1}\r\n",

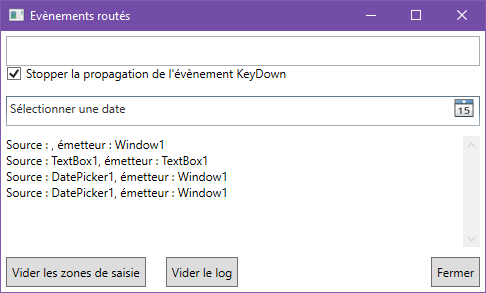
((FrameworkElement)e.Source).Name,

((FrameworkElement)sender).Name);

}

**Résultat**

Exécutons ce code et refaisons exactement le même scénario que plus haut, en cochant la case :



Nous observons 2 différences par rapport au résultat précédent :

* Les caractères tapés ne s’affichent pas dans les zones de saisie de texte et de date
* L’appui sur une touche dans la zone de texte ne génère plus qu’une seule ligne dans le bloc de texte

Explications :

Le passage de la propriété Handled à True bloque la propagation de l’évènement KeyDown, qui ne peut donc plus être émis par la fenêtre. Dans la logique interne de la TextBox, cela a aussi pour effet de bloquer l’affichage des caractères saisis à l’écran.

# Les commandes

## Présentation

Les commandes font le lien entre des tâches de l’application (ex : un calcul, une recherche, un tri…) et les éléments de l’interface utilisateur qui les déclenchent.

Elles fournissent les avantages suivants par rapport à une gestion classique par évènement :

* Une commande est indépendante. Elle peut ainsi être appelée par plusieurs sources, qu’elles soient visuelles (ex : boutons, menus) ou non (ex : raccourci clavier)
* L’activation/désactivation de toutes les sources reliées à une commande est centralisée
* Les commandes s’inscrivent dans la logique du modèle MVVM pour le découplage entre visuel et logique applicative

[Page MSDN](https://msdn.microsoft.com/en-us/library/ms752308.aspx) de vue d’ensemble des commandes

Une commande est une classe qui implémente l’interface ICommand. Cette interface définit notamment 2 méthodes :

bool CanExecute(object parameter)

void Execute(object parameter)

* La première définit les conditions dans lesquelles la commande est active (retour True) ou inactive (retour False)
* La seconde définit la logique de la tâche à exécuter

## Evènements vs commandes

Comparons ce qu’il faut faire pour gérer une tâche avec des évènement et avec des commandes :

**Avec des évènements :**

* Définir un gestionnaire à exécuter pour la tâche (qui contient la logique applicative)
* Sur chaque élément WPF, associer l’évènement approprié (souvent Click) au gestionnaire
* Gérer l’activation des éléments WPF et des raccourcis clavier en fonction du contexte (ex : un texte sélectionné, un élément sélectionné dans une liste, une feuille activée…). Cela conduit généralement à gérer d’autres évènements.

L’activation des éléments d’interface est souvent difficile à gérer et pas centralisée, comme nous allons le voir dans l’exemple qui suit. Le code correspondant peut amener un couplage fort entre la logique de l’application et l’interface.

**Avec une commande :**

* Définir une méthode à exécuter pour la tâche, et une méthode qui définit dans quelles conditions la commande est active.
* Créer un objet ICommand associé aux 2 méthodes précédentes
* Affecter la commande à la propriété Command de chaque élément WPF

Conséquences :

* L’ajout d’un nouvel élément source de déclenchement de l’action consiste simplement à affecter sa propriété Command
* La gestion de l’activation de toutes les sources de déclenchement est centralisée
* La liaison de la commande avec les 2 gestionnaires est également réalisée de façon centralisée

## Commandes routées

WPF fournit 2 implémentations de ICommand avec les classes RoutedCommand et RoutedUICommand. La seconde dérive de la première et ne fait qu’ajouter une propriété Text.

Dans ces commandes, les méthodes Execute et CanExecute ne contiennent pas directement la logique applicative, mais lèvent des évènements routés montants nommés Executed et CanExecute, sur lesquels on branche nos gestionnaires.

Les commandes routées ont 2 inconvénients majeurs :

* Elles ne sont pas compatibles avec le modèle MVVM (cf. plus loin), car elles ne permettent pas de découpler complètement le visuel de la logique applicative (à cause de l’utilisation d’évènements)
* L’évènement CanExecute est déclenché extrêmement souvent. Si on y branche du code complexe, les performances se dégradent fortement et l’interface devient peu réactive.

**Commandes prédéfinies**

WPF fournit une centaine de commandes prédéfinies pour des tâches classiques (ouvrir, copier, lire…). Bien entendu c’est à nous d’écrire les gestionnaires associés en fonction de nos besoins, mais souvent des raccourcis claviers sont déjà mappés.

Les commandes sont implémentées par des propriétés statiques réparties sur 5 classes statiques :

* [ApplicationCommands](https://msdn.microsoft.com/en-us/library/system.windows.input.applicationcommands.aspx)
* [ComponentCommands](https://msdn.microsoft.com/en-us/library/system.windows.input.componentcommands.aspx)
* [EditingCommands](https://msdn.microsoft.com/en-us/library/system.windows.documents.editingcommands.aspx)
* [MediaCommands](https://msdn.microsoft.com/en-us/library/system.windows.input.mediacommands.aspx)
* [NavigationCommands](https://msdn.microsoft.com/en-us/library/system.windows.input.navigationcommands.aspx)

**Commandes personnalisées**

Il est bien sûr possible de créer nos propres commandes routées. Il faut pour cela suivre l’exemple des commandes prédéfinies, en créant des classes statiques qui exposent des propriétés statiques pour les différentes commandes. Nous ne nous attarderons pas sur cette possibilité, car il est plus courant de créer des commandes relais, que nous verrons plus loin.

**Exemple**

Nous allons utiliser les commandes prédéfinies Delete, Copy et Paste dans l’application suivante :

|  |  |
| --- | --- |
|  | Cette application permet de :   * Copier dans le presse-papier le texte sélectionné dans la zone de saisie * Coller le texte du presse-papier dans cette même zone.   Le bouton Effacer permet simplement de vider la zone de saisie.  Nous souhaitons gérer correctement l’activation des boutons Copier et Coller, c’est-à-dire faire en sorte que :   * Le bouton Copier soit inactif si aucun texte n’est sélectionné dans la zone de texte * Le bouton Coller soit inactif si le presse-papier contient autre chose que du texte |

On profite ici du fait que la TextBox possède des méthodes Copy et Paste toutes faites pour gérer le presse-papier.

On fait en sorte de désactiver le bouton Copier si le texte sélectionné est vide, et d’activer le bouton Coller uniquement si le presse-papier contient du texte.

Comme les commandes sont prédéfinies, nous n’avons pas besoin de les créer. Nous devons :

* Définir les méthodes à exécuter par ces commandes
* Brancher les commandes sur les boutons

Voici le code xaml correspondant :

<StackPanel Grid.Row="1">

<!-- Branchement des commandes Delete, Copy et Paste sur leurs méthodes -->

<StackPanel.CommandBindings>

<CommandBinding Command="ApplicationCommands.Delete"

Executed="Delete\_Executed"/>

<CommandBinding Command="ApplicationCommands.Copy"

Executed="Copy\_Executed"

CanExecute="Copy\_CanExecute"/>

<CommandBinding Command="ApplicationCommands.Paste"

Executed="Paste\_Executed"

CanExecute="Paste\_CanExecute"/>

</StackPanel.CommandBindings>

<Label Content="Commandes"/>  
 <!-- Affectation des commandes aux boutons -->

<Button Content="Effacer" Command="ApplicationCommands.Delete"/>

<Button Content="Copier" Command="ApplicationCommands.Copy"/>

<Button Content="Coller" Command="ApplicationCommands.Paste"/>

</StackPanel>

Pour définir les méthodes à exécuter par les commandes, nous utilisons des éléments CommandBinding, placés dans la collection CommandBindings du StackPanel qui contient les boutons

Chaque élément WPF possède une collection **CommandBindings**. Le binding d’une commande est recherché sur l’élément lui-même, puis sur ses parents en remontant l’arbre visuel. On définit généralement les bindings sur la fenêtre. Nous les avons ici définis sur le conteneur des boutons pour la démonstration.

L’élément **CommandBinding** possède une propriété **Command** pour indiquer le nom de l’objet ICommand sur lequel on veut brancher les gestionnaires des évènements Executed et CanExecute. Ici nous avons utilisé 3 commandes prédéfinies du groupe ApplicationCommands : Delete, Copy et Paste

Pour pouvoir exécuter les commandes, nous les branchons sur les boutons grâce à la propriété Command.

Il ne reste plus qu’à définir le code des méthodes associées aux commandes :

private void Delete\_Executed(object sender, ExecutedRoutedEventArgs e)

{

TextBox1.Clear();

}

// Delete\_CanExecuted n'étant pas défini, la commande est toujours active

private void Copy\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

// Commande activée uniquement si du texte est sélectionné dans la zone de saisie

e.CanExecute = !String.IsNullOrEmpty(TextBox1.SelectedText);

}

private void Copy\_Executed(object sender, ExecutedRoutedEventArgs e)

{

TextBox1.Copy();

}

private void Paste\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

// Commande activée uniquement si le presse-papier contient du texte

e.CanExecute = Clipboard.ContainsText();

}

private void Paste\_Executed(object sender, ExecutedRoutedEventArgs e)

{

TextBox1.Paste();

}

Notons un point important sur ce code : on n’a pas besoin de faire référence aux boutons pour gérer leur état d’activation. On se contente de spécifier les conditions dans lesquelles les commandes sont actives, ce qui affecte automatiquement l’état d’activation des éléments sur lesquelles elles sont branchées.

Lorsqu’on exécute l’application, on constate que :

* Le bouton Copier est automatiquement désactivé par défaut, tant qu’aucun texte n’est sélectionné
* Dès que le presse-papier ne contient plus de texte (par exemple si on copie une image), le bouton Coller se désactive

Obtenir le même comportement en utilisant des évènements à la place des commandes nécessiterait de gérer les évènements TextBox. SelectionChanged et Window.Activated, et le résultat ne serait pas si bon.

**NB/ Autre méthode**

Il est en fait possible de gérer de façon encore plus simple le copier/coller sur une TextBox., uniquement avec le code xaml suivant (sans code C#) :

<StackPanel Grid.Row="3">

<Button Content="Copier" Command="ApplicationCommands.Copy"

CommandTarget="{Binding ElementName=TextBox1}"/>

<Button Content="Coller" Command="ApplicationCommands.Paste"

CommandTarget="{Binding ElementName=TextBox1}"/>

</StackPanel>

En effet, certains contrôles de saisie comme la TextBox ont des gestionnaires par défaut pour les commandes courantes telles que Cut, Copy, Paste, Undo et Redo.

La seule chose à faire dans ce cas est de définir le contrôle comme cible de la commande, au moyen de la propriété **CommandTarget**, comme le montre le code ci-dessus.

## Commandes relais

La classe RelayCommand est une implémentation de l’interface ICommand particulièrement adaptée à la séparation du visuel et de la logique métier. Elle ne fait pas partie de la bibliothèque du framework .net, mais de la bibliothèque MVVM Light. Elle est couramment dans les applications WPF. Voici son code :

using System;

using System.Windows.Input;

namespace Evts\_Commandes

{

public class RelayCommand : ICommand

{

// Champs privés

private readonly Action<object> \_execute;

private readonly Predicate<object> \_canExecute;

// Constructeurs

public RelayCommand(Action<object> execute) : this(execute, null)

{

}

public RelayCommand(Action<object> execute, Predicate<object> canExecute)

{

if (execute == null)

throw new ArgumentNullException("execute");

\_execute = execute;

\_canExecute = canExecute;

}

// Implémentation de l'interface ICommand

public event EventHandler CanExecuteChanged

{

add

{

if (\_canExecute != null)

CommandManager.RequerySuggested += value;

}

remove

{

if (\_canExecute != null)

CommandManager.RequerySuggested -= value;

}

}

public void Execute(object parameter)

{

\_execute(parameter);

}

public bool CanExecute(object parameter)

{

return \_canExecute == null ? true : \_canExecute(parameter);

}

}

}

Cette classe permet de créer une commande, dont le code des méthodes Execute et CanExecute est défini au moyen de délégués. Ces délégués sont passés en paramètre du constructeur de la commande. Le second est optionnel.

Action<Object> est un type délégué qui représente une méthode prenant un paramètre de type Object, et ne retournant rien.

Predicate<Object> est un type délégué qui représente une fonction prenant un paramètre de type Object, et retournant un booléen.

L’utilisation de délégués permet de séparer la déclaration de la commande, et la logique métier exécutée par cette commande, comme nous allons le voir dans l’exemple qui suit.

**Exemple : calculatrice simplifiée**

Considérons l’application suivante, qui permet de faire une opération simple entre 2 nombres :



Le calcul est exécuté par une commande déclenchée par le bouton « = »

Le type d’opération (+ - x /) est sélectionnable au moyen d’une liste déroulante. Il sera passé en paramètre à la méthode Execute de la commande.

Voici le code xaml :

<DockPanel Margin="5">

<TextBox Text="{Binding Path=Operande1}"/>

<ComboBox Name="cmbOperateur" Width="50" FontSize="20">

<ComboBox.Items>

<ComboBoxItem Content="+" IsSelected="True"/>

<ComboBoxItem Content="-"/>

<ComboBoxItem Content="X"/>

<ComboBoxItem Content="/"/>

</ComboBox.Items>

</ComboBox>

<TextBox Text="{Binding Path=Operande2}"/>

<Button Content="=" Width="50" FontSize="20"

Command="{Binding CmdCalculer}"

CommandParameter="{Binding ElementName=cmbOperateur, Path=SelectedIndex}"/>

<Label Content="{Binding Path=Resultat}"

HorizontalContentAlignment="Center"

VerticalContentAlignment="Center"

Background="AliceBlue"/>

</DockPanel>

La commande est branchée sur le bouton au moyen de sa propriété Command. On passe en paramètre à cette commande l’indice de l’item sélectionné dans la liste déroulante. On utilise pour cela la propriété CommandParameter du bouton.

On voit que la syntaxe est plus simple et naturelle que pour l’utilisation de commandes routées, car il n’est pas nécessaire de définir de CommandBinding au préalable. Comme les commandes relais ne sont pas associées à des évènements, on n’est pas obligé de mettre le code métier dans le code-behind de la fenêtre.

Le seul code-behind ajouté est la ligne suivante dans le constructeur de la fenêtre :

DataContext = new Contexte();

Cela permet d’affecter le contexte de la fenêtre avec une instance de la classe décrite ci-dessous :

using System.ComponentModel;

using System.Windows.Input;

namespace Evts\_Commandes

{

public class Contexte : INotifyPropertyChanged

{

private double \_resultat;

// Implémentation de INotifyPropertyChanged

public event PropertyChangedEventHandler PropertyChanged;

// Propriétés

public double Operande1 { get; set; }

public double Operande2 { get; set; }

public double Resultat

{

get { return \_resultat; }

private set

{

\_resultat = value;

PropertyChanged(this, new PropertyChangedEventArgs("Resultat"));

}

}

// Commande

private ICommand \_cmdCalculer;

public ICommand CmdCalculer

{

get

{

if (\_cmdCalculer == null)

\_cmdCalculer = new RelayCommand(Calculer);

return \_cmdCalculer;

}

}

// Fonction appelée par la commande

private void Calculer(object operateur)

{

switch ((int)operateur)

{

case 0:

Resultat = Operande1 + Operande2;

break;

case 1:

Resultat = Operande1 - Operande2;

break;

case 2:

Resultat = Operande1 \* Operande2;

break;

case 3:

Resultat = Operande1 / Operande2;

break;

}

}

}

}

Cette classe définit les propriétés sources pour les 2 opérandes et pour le résultat de l’opération. Comme le résultat est calculé par le code, il faut pouvoir notifier son changement de valeur ; c’est pourquoi on appelle PropertyChange dans l’accesseur set de la propriété Resultat.

La classe contient également la définition de la commande CmdCalculer. Le code est très concis, car il consiste simplement à instancier une RelayCommand en passant la méthode qui représente l’action à son constructeur.

NB/ Si on voulait brancher une méthode dont la signature ne respecte pas le type délégué, il suffirait d’utiliser une méthode anonyme ou une expression lambda comme adaptateur.

La méthode Calculer en elle-même ne présente aucune difficulté. Son paramètre est de type object pour être compatible avec le type délégué Action<object>.

# Le modèle MVVM

MVVM est un modèle d’architecture dérivé de modèles plus anciens tels que MVP (Model View Presenter), qui permet de tirer pleinement partie des possibilités de WPF pour séparer complètement le code métier et l’interface visuelle. Cela permet :

* Une collaboration plus simple entre les designers et les développeurs
* La production d’un code cloisonné, plus facile à faire évoluer, à réutiliser et à tester

En fait, WPF et MVVM ont été conçus conjointement par Microsoft. John Gossman, un des architectes de WPF a exposé le modèle pour la première fois en 2005 dans [cet article de blog](https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/). MVVM a ensuite été utilisé avec d’autres technologies, telles que HTML5, Java…

La séparation du code métier de l’interface visuelle signifie qu’il n’y a aucun code métier dans les fichiers .xaml.cs (qu’on appelle code-behind). Pour cela, MVVM s’appuie notamment sur 2 mécanismes centraux de WPF : les liaisons de données (binding) et les commandes.

[Page synthétique](https://msdn.microsoft.com/en-us/library/hh848246.aspx) sur MSDN

Modèle, vue et vue-modèle

**MVVM** signifie Model View ViewModel. Voyons ce qui se cache derrière ces 3 éléments :

**Le modèle** représente la couche de données, et la logique métier associée (règles de gestion). Il manipule généralement les données au travers d’objets (entités) en mémoire, et utilise une base de données ou des fichiers pour leur stockage. Il n’a aucune connaissance de l’interface graphique, et n’a pas besoin de WPF.

**La vue** représente le visuel, l’interface utilisateur. Elle est décrite en xaml pur, avec un code-behind très limité, qui ne contient pas de logique métier.

Une vue peut être décomposée en plusieurs sous-vues au moyen de UserControl.

Une vue peut avoir sa propre vue-modèle ou hériter de celle de sa vue parente. Elle obtient les données à afficher de sa vue-modèle par binding, ou en appelant des commandes.

**La vue-modèle** (ou modèle de vue) est une abstraction de la vue, qui décrit son état et son comportement. C’est un adaptateur entre la vue et le modèle qui réalise plusieurs choses :

* Elle interagit avec le modèle en appelant des méthodes sur les classes du modèle
* Elle expose les données du modèle via des propriétés exploitables par la vue
* Elle fournit les commandes que l’utilisateur peut déclencher via la vue
* Elle mémorise l’état de la vue (éléments sélectionnés dans les contrôles, état d’édition…etc.) et décrit ses comportements dans les différents états (indiquer un état d’édition ou de consultation, une opération en cours…etc)

La vue-modèle constitue le **contexte** (DataContext) de la vue.

**Relations entre eux**

Le schéma ci-dessous représente de façon synthétique les interactions entre les 3 éléments :

Modèle

Vue modèle

Etat et comportement

Envois de notifications

Binding et commandes

Mise à jour du modèle

Dans le modèle MVVM :

* La vue ne connait que la vue-modèle
* La vue-modèle ne connait que le modèle
* Le modèle ne connaît que lui-même

Les liens entre la vue et la vue-modèle sont basés sur le databinding, les commandes, et les comportements (behaviors, que nous n’aborderons pas dans ce cours). Ces concepts techniques permettent en particulier la mise à jour de la vue à partir de la vue-modèle, sans que la seconde connaisse la première.

Pour affecter la vue-modèle en tant que contexte de la vue, on peut :

* Instancier la vue-modèle dans le constructeur de la vue (ce que nous avons fait tout au long de ce cours)
* Injecter la vue-modèle dans la vue en utilisant un conteneur d’inversion de contrôle (IoC). Nous reviendrons sur l’IOC à la fin de ce cours.

**Centralisation de l’accès aux vues modèles**

Dans une application basée sur MVVM, les différentes vues modèles sont souvent exposées comme propriétés d’un singleton appelé conventionnellement ViewModelLocator, et déclaré dans les ressources de l’application. Les vues peuvent ensuite se lier aux vues modèles par databinding.

Ex de classe ViewModelLocator avec une propriété fournissant la vue-modèle de la fenêtre principale

namespace MVVMApp.ViewModel

{

public class ViewModelLocator

{

public MainViewModel Main

{

get { return new MainViewModel(); }

}

}

}

Cette classe est instanciée dans le dictionnaire de ressources de l’application :

<Application.Resources>

<vm:ViewModelLocator x:Key="ViewModelLocator"

xmlns:vm="clr-namespace:MVVMApp.ViewModel" />

</Application.Resources>

On peut ensuite utiliser cette ressource dans le code xaml de la fenêtre, pour définir son contexte :

<Window x:Class="MVVMApp.MainWindow"

...

Title="MainWindow" Height="350" Width="525"

DataContext="{Binding Source={StaticResource Locator}, Path=Main}">

# Les contrôles utilisateur (UserControl)

Pour des besoins avancés, on peut être amené à créer nos propres contrôles WPF. C’est le cas notamment lorsqu’un morceau d’interface visuelle est utilisé dans plusieurs fenêtres de l’application, ou dans plusieurs applications.

Un contrôle utilisateur est un assemblage de contrôles WPF existants, et est modélisé par une classe dérivant de **UserControl**.

NB/ Il est également possible de créer des contrôles personnalisés (custom controls en anglais), qui héritent directement de Control ou ContentControl, mais nous ne traiterons pas ce sujet dans ce cours.

## Créer et utiliser un UserControl

Créer un contrôle utilisateur dans la solution courante

Si le contrôle est destiné à être utilisé uniquement dans la solution Visual Studio courante, il peut être créé de le même façon qu’une fenêtre. Dans le menu contextuel du projet, choisir « ajouter \ Contrôle utilisateur », puis donner un nom au contrôle. Ceci crée un nouveau fichier xaml semblable à celui d’une fenêtre, avec l’élément UserControl à la place de Window :

<UserControl x:Class="ControleUtilisateur.MonControle"  
...

</UserControl>

On peut ensuite créer le visuel du contrôle, comme celui d’une fenêtre.

Créer un contrôle utilisateur utilisable dans d’autres solutions

Si le contrôle est destiné à être utilisé dans d’autres solutions Visual Studio, il faut créer un nouveau projet de type « Bibliothèque de contrôles utilisateurs ». Cette bibliothèque sera générée sous forme de dll, et pourra être distribuée sous cette forme. Pour l’utiliser dans un projet, il suffit alors d’ajouter une référence vers cette dll dans le projet.

Utiliser un contrôle utilisateur

Pour utiliser le contrôle dans une fenêtre, il faut faire référence à son espace de noms et éventuellement à son assembly, s’il est dans un autre projet.

Exemple dans le cas où le contrôle est dans le même projet que la fenêtre :

<Window x:Class="ControleUtilisateur.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:ControleUtilisateur"

Title="MainWindow" Height="350" Width="525">

<Grid>

<local:MonControle/>

</Grid>

</Window>

Exemple dans le cas où le contrôle est dans un autre projet :

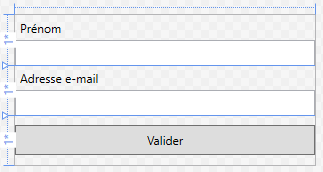
xmlns:wtk="clr-namespace:Xceed.Wpf.Toolkit;assembly=Xceed.Wpf.Toolkit"

NB/ L’[Extended WPF Toolkit](http://wpftoolkit.codeplex.com/) est une librairie de contrôles WPF très connue, créée par la société Xceed. L’édition Community est gratuite et disponible sous forme de package NuGet.

## Ajouter des propriétés

Lorsqu’on crée un contrôle utilisateur, on souhaite généralement qu’il expose des propriétés, comme les contrôles WPF natifs. Pour qu’elles supportent le Databinding et la notification des changements, ces propriétés doivent être des propriétés de dépendance (Dependency Properties).

Considérons le contrôle utilisateur suivant, qui modélise une interface d’inscription à une newsletter :



Nous aimerions pouvoir utiliser ce contrôle de la façon suivante :

<local:UCInscription Prenom="{Binding Path=Prenom}" Email="{Binding Path=Email}"

Command="{Binding Path=CmdValider}"/>

… c’est à dire disposer de propriétés nommées Prenom, Email et Command pour pouvoir faire des liaisons de données dessus. Pour cela nous devons créer des propriétés de dépendance dans le codebehind du contrôle. Voici le code pour créer la propriété Email :

using System;

using System.Windows;

using System.Windows.Controls;

namespace ControleUtilisateur

{

public partial class UCInscription : UserControl

{

public UCInscription()

{

InitializeComponent();

}

public static readonly DependencyProperty EmailProperty =

DependencyProperty.Register("Email", // Nom de la propriété

typeof(string), // Type de la propriété

typeof(UCInscription), // Type de la classe

new PropertyMetadata(String.Empty)); // Valeur par défaut

// Propriété ordinaire pour accéder à la DP

public string Email

{

get { return (string)GetValue(EmailProperty); }

set { SetValue(EmailProperty, value); }

}  
 }  
}

Il y a 2 étapes :

* Créer une nouvelle propriété de dépendance au moyen de la méthode statique DependencyProperty.Register et récupérer l’instance obtenue dans une variable statique publique en lecture seule.
* Créer une propriété ordinaire encapsulant la propriété de dépendance pour pouvoir lire et écrire sa valeur.

Remarques :

* Le snippet « propdp » de Visual Studio permet de générer très facilement une propriété de dépendance.
* Le dernier paramètre de la méthode Register, permettant de spécifier la valeur par défaut de la propriété, est optionnel.
* Il est important de ne pas mettre de code supplémentaire dans la propriété ordinaire, car le runtime accède à la propriété de dépendance en appelant directement GetValue et SetValue, et non la propriété ; le code supplémentaire ne serait dans ce cas pas exécuté.

Il ne reste maintenant plus qu’à relier les propriétés de dépendance aux éléments WPF du contrôle :

<UserControl x:Class="ControleUtilisateur.UCInscription"

...

Name="ucInscription">

<Grid>

...

<Label Content="Prénom"/>

<TextBox Text="{Binding ElementName=ucInscription, Path=Prenom}"

MinHeight="30" Margin="0,25,0,0"/>

<Label Content="Adresse e-mail" Grid.Row="1"/>

<TextBox Text="{Binding ElementName=ucInscription, Path=Email}"

MinHeight="30" Margin="0,25,0,0" Grid.Row="1"/>

<Button Command="{Binding ElementName=ucInscription, Path=Command}"

Content="Valider" MinHeight="30" Margin="0,10" Grid.Row="2" />

</Grid>

</UserControl>

Les propriétés de dépendance sont définies sur le UserControl lui-même. Pour les relier aux éléments WPF, on utilise donc le UserControl comme source de binding, via ElementName.

# Références bibliographiques

Documentation en ligne [MSDN](https://msdn.microsoft.com/fr-fr/library/ms742119(v=vs.110).aspx) sur WPF

Livre « MCTS Self-Paced Training Kit (Exam 70-502): Microsoft .NET Framework 3.5—Windows Presentation Foundation » de Matthew A. Stoecker

Site <http://www.wpf-tutorial.com/>

Exemples de codes WPF, initialement sur MSDN : <https://github.com/Microsoft/WPF-Samples>

Poster de la hiérarchie de classes WPF : <https://kittencode.wordpress.com/category/c/wpf/>

Tables de caractères Unicode (pour les icônes) : <http://www.alanwood.net/demos/wingdings.html>