Les principaux contrôles

Le framework .NET fournit en standard de nombreux contrôles WPF couvrant la majorité des besoins relatifs à la création d'une interface graphique moderne. Ils sont définis dans la librairie `PresentationFramework.dll`, dans l'espace de noms `System.Windows.Controls`.

La majorité de ces contrôles possède un jeu de propriétés communes permettant de gérer leur comportement dans la mise en page de l'application ainsi que leur aspect. Les principales sont listées ci-dessous :

`Height` et `Width`	Définit la hauteur et la largeur du contrôle.
`HorizontalAlignment` et `VerticalAlignment`	Définit comment le contrôle doit se placer dans l'espace qui est disponible horizontalement et verticalement. Les valeurs peuvent être `Left` (alignement à gauche), `Center` (centré), `Right` (alignement à droite) et `Stretch` (occupation de tout l'espace disponible) pour l'alignement horizontal, et `Top` (alignement en haut), `Bottom` (alignement en bas), `Center` et `Stretch` pour l'alignement vertical.
`Margin`	Définit l'espace qui doit rester libre autour des limites du contrôle.
`Padding`	Définit l'espace qui doit rester libre entre le bord du contrôle et son contenu.
`BorderBrush`	Définit la couleur de la bordure du contrôle.
`BorderThickness`	Définit l'épaisseur de trait de la bordure du contrôle.
`Visibility`	Définit l'état de visibilité du contrôle. Trois valeurs sont disponibles : `Visible`, `Hidden` (le contrôle est masqué mais la zone qu'il devrait occuper est indisponible), `Collapsed` (le contrôle est masqué mais la zone qu'il devrait occuper est libérée).

Les contrôles WPF peuvent être regroupés en plusieurs catégories définies par leur objectif.

1. Contrôles de fenêtrage

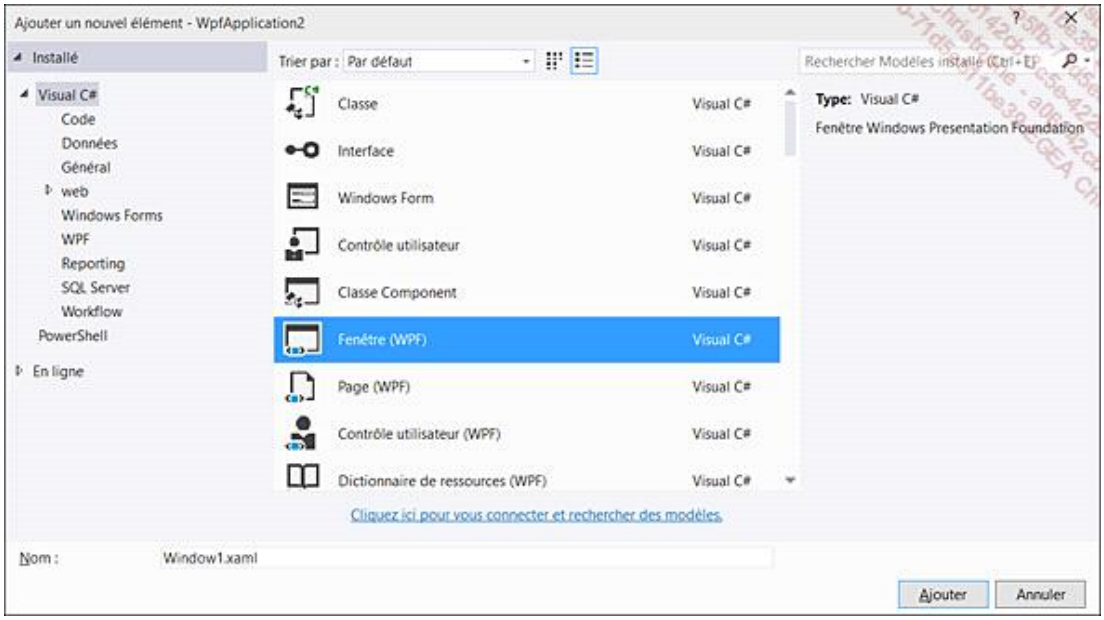
Les contrôles de fenêtrage sont essentiels dans une application WPF. Ce sont eux qui sont les conteneurs de l'interface graphique que vous allez créer.

Deux contrôles standards appartiennent à cette catégorie. Ils correspondent à des modes de navigation distincts.

a. Window

Le contrôle `Window` permet, comme son nom l'indique, de définir une fenêtre d'application. C'est le contrôle de fenêtrage le plus fréquemment utilisé.

Il peut être créé à l'aide de la boîte de dialogue **Ajouter un nouvel élément** (clic droit sur le projet puis **Ajouter - Nouvel élément** ou [Ctrl][Shift] A) :



La validation du choix **Fenêtre (WPF)** à l'aide du bouton **Ajouter** crée un fichier `.xaml` et un fichier `.xaml.cs` et les intègre au projet en cours.

Le contenu du fichier `.xaml` tout juste créé est le suivant :

```
<Window x:Class="WpfApplication1.Window1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="300" Width="300">
    <Grid>

    </Grid>
</Window>
```

Comme vu précédemment, le code de ce fichier est formé de balises. L'objet `Window` est la racine de l'arbre XAML.

L'attribut `x:Class` de cet objet correspond à la classe C# associée à la fenêtre. Cette classe se charge de lancer l'initialisation des contrôles graphiques dans son constructeur. C'est aussi dans cette classe que sont placés les gestionnaires d'événements associés aux contrôles de la fenêtre.

Le contrôle possède de nombreuses propriétés permettant de définir son aspect, son comportement ou son emplacement. Le tableau ci-dessous recense et détaille les plus importantes.

`Title`	Définit le titre de la fenêtre.
`ResizeMode`	Définit les modes de redimensionnement disponibles pour la fenêtre.
`ShowInTaskbar`	Définit si la fenêtre doit être visible dans la barre des tâches de Windows.
`WindowState`	Définit si la fenêtre doit être maximisée, minimisée ou dimensionnée normalement.
`WindowStartupLocation`	Définit si la fenêtre doit être centrée par rapport à sa fenêtre parente ou à l'écran.

Topmost	Définit si la fenêtre doit rester en avant-plan même si l'application n'a pas le focus.

Pour ouvrir une nouvelle fenêtre, il faut l'instancier puis utiliser sa méthode Show, ou sa méthode ShowDialog si la fenêtre doit être modale.

```
Window1 window = new Window1();
window.Show();
```

Ce contrôle définit aussi plusieurs événements, dont Loaded qui permet d'exécuter une portion de code lorsque le contenu de la fenêtre a été chargé mais pas encore affiché. Les événements Closing et Closed sont déclenchés respectivement avant et après la fermeture de la fenêtre. Le premier permet d'empêcher la fermeture de la fenêtre dans son gestionnaire de la manière suivante :

```
private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = true;
}
```

La fenêtre de démarrage d'une application est définie dans le fichier App.xaml. Une des propriétés de l'objet Application décrit dans ce fichier est StartupUri. Celle-ci accepte comme valeur le chemin relatif menant à la fenêtre qui doit être ouverte au démarrage du programme.

```
<Application x:Class="WpfApplication2.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

b. NavigationWindow

WPF propose, en plus du mode multifenêtré traditionnel, un mode de navigation similaire à celui d'un navigateur web. Dans ce mode, l'utilisateur n'a plus plusieurs fenêtres à sa disposition mais une seule, dans laquelle il peut naviguer entre plusieurs pages.

La fenêtre principale est de type NavigationWindow et les différents contenus sont placés dans des objets Page. Comme un navigateur web, un objet NavigationWindow dispose de deux boutons **Précédent** et **Suivant** et gère donc un historique de navigation entre chaque objet Page.

NavigationWindow est une classe dérivée de la classe Window, elle a donc accès à toutes les propriétés vues précédemment. Elle possède aussi des propriétés propres à son mode de navigation.

Source	Définit l'URI de la page en cours d'affichage.
BackStack	Contient l'historique des pages précédentes.

ForwardStack	Contient la liste des pages suivantes.
CanGoBack	Définit s'il est possible pour la fenêtre de naviguer vers une page précédente.
CanGoForward	Définit s'il est possible pour la fenêtre de naviguer vers une page suivante.

La création d'une NavigationWindow nécessite quelques manipulations car il n'existe pas de modèle pour ce type d'objet dans Visual Studio :

- Ouvrez la fenêtre **Ajouter un nouvel élément** (clic droit sur le projet puis **Ajouter - Nouvel élément** ou [Ctrl][Shift] A) et créez une **Fenêtre (WPF)**.
- Dans le fichier .xaml créé, il faut modifier la déclaration de l'objet racine : Window devient NavigationWindow. Supprimez également les balises <Grid> et </Grid> contenues dans l'objet NavigationWindow. En effet, ce composant ne supporte pas le contenu direct.
- Dans le fichier .xaml.cs associé, il faut modifier le nom de la classe mère de la même manière :

```
public partial class Window1 : Window
```

devient :

```
public partial class Window1 : NavigationWindow
```

À ce stade, la fenêtre est utilisable. Il est donc possible de valoriser la propriété StartupUri de l'objet Application défini dans App.xaml avec le chemin relatif menant à la fenêtre nouvellement créée.

```
StartupUri="Window1.xaml"
```

La création d'un objet Page peut en revanche être effectuée à l'aide d'un modèle. Dans la fenêtre **Ajouter un nouvel élément**, ce modèle est nommé **Page (WPF)**. Celui-ci ajoute un fichier Page1.xaml au projet ainsi qu'un fichier de code-behind associé.

Le code XAML de cette page est relativement semblable à celui d'une nouvelle fenêtre :

```
<Page x:Class="WpfApplication2.Page1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
    Title="Page1">
    <Grid>

    </Grid>
</Page>
```

Le contenu de cette page sera affiché dans la `NavigationWindow` lorsque la propriété `Source` de la fenêtre aura pour valeur "Page1.xaml". Cette valeur peut être modifiée par affectation d'un chemin de fichier ou par l'utilisation de la méthode `Navigate` de la fenêtre. Cette méthode accepte comme paramètre le chemin relatif du fichier `.xaml` définissant une page.

```
NavigationWindow fenetre = new NavigationWindow();
fenetre.Navigate("Page1.xaml");
```

2. Contrôles de disposition

Les contrôles de disposition sont des conteneurs permettant de définir le positionnement de chacun des contrôles graphiques qu'ils contiennent. Plusieurs types de contrôles de disposition sont définis dans le framework .NET. Ils répondent à la grande majorité des besoins en matière de définition d'interface graphique.

a. Grid

Le contrôle `Grid` représente une **grille** pour laquelle on définit des **lignes** et des **colonnes**. Chacun des composants graphiques est positionné dans cette grille à l'aide d'un index de ligne et de colonne et peut s'étendre sur plusieurs lignes et/ou colonnes.

Le contrôle Grid est le conteneur principal par défaut défini dans le modèle de fenêtre WPF de Visual Studio.

Pour définir une grille avec trois lignes et trois colonnes, on écrit le code suivant :

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
</Grid>
```

Les propriétés `RowDefinitions` et `ColumnDefinitions` du contrôle `Grid` sont des collections pouvant contenir respectivement autant d'objets `RowDefinition` ou `ColumnDefinition` que nécessaire.

À l'exécution, la fenêtre contenant cette définition de grille peut sembler bien vide... En effet, par défaut, notre grille est transparente et vide. Pour visualiser la grille, il est possible de valoriser sa propriété `ShowGridLines` à `True`.

```
<Grid ShowGridLines="True">
    ...
</Grid>
```

Il est évidemment possible de définir la hauteur de chacune des lignes au travers de la propriété `Height` de chaque objet `RowDefinition` et la largeur de chaque colonne grâce à la propriété `Width` des objets de type `ColumnDefinition`.

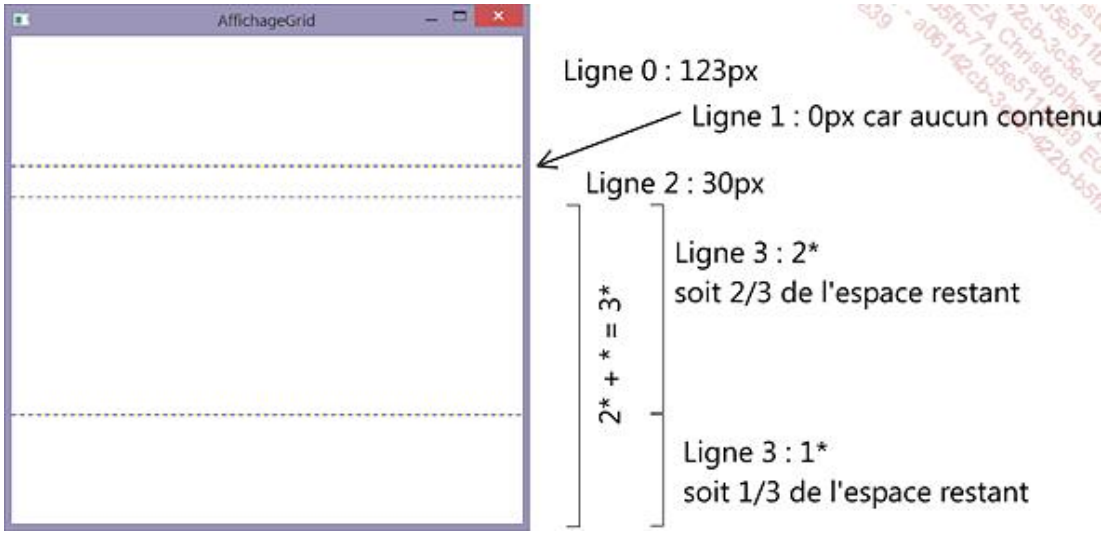
Il existe trois moyens de spécifier ces tailles qui sont, par ordre de priorité :

- Une valeur en nombre de pixels.
- `Auto` : la ligne (ou colonne) s'adapte à son contenu. Si elle n'a pas de contenu, elle ne sera pas visible.
- * (Star) : l'utilisation de ce mode de mesure est un peu plus complexe. Un coefficient est affecté à chaque mesure "Star". Ce coefficient vaut 1 s'il n'est pas spécifié explicitement. La taille effective est calculée par la formule suivante :

$$\text{Taille} = \text{Taille libre restante} \times \frac{\text{Coeff. pour la ligne / colonne}}{\text{Somme des coeff. des lignes / colonnes}}$$

Ci-dessous, un exemple annoté montrant l'application des différents types de mesures.

```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="123" />
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="30"/>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
</Grid>
```



Pour positionner des contrôles dans la grille, il est nécessaire de les déclarer et de valoriser pour chacun d'eux les propriétés attachées `Grid.Row` et `Grid.Column`. La valeur par défaut de ces propriétés est 0, ce qui veut dire que si elles ne sont pas utilisées explicitement sur un contrôle, celui-ci sera positionné dans la première ligne et la première colonne de la grille.

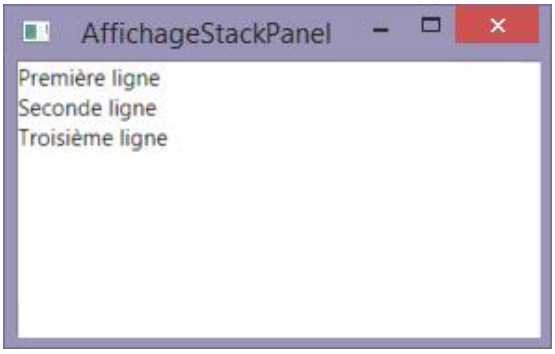
```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="123" />
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="30"/>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <!-- Insertion d'un contrôle Grid dans la deuxième ligne
(index 1) -->
    <Grid Background="Red" Grid.Row="1" Height="50" />
</Grid>
```

b. StackPanel

Le contrôle `StackPanel` permet d'accoler verticalement ou horizontalement plusieurs composants visuels. Sa propriété `Orientation` contrôle le sens dans lequel les éléments sont empilés, et sa valeur par défaut est `Vertical`.

```
<StackPanel Orientation="Vertical">
    <TextBlock Text="Première ligne" />
    <TextBlock Text="Seconde ligne" />
    <TextBlock Text="Troisième ligne" />
</StackPanel>
```

Le rendu de cette portion de code est le suivant :



c. DockPanel

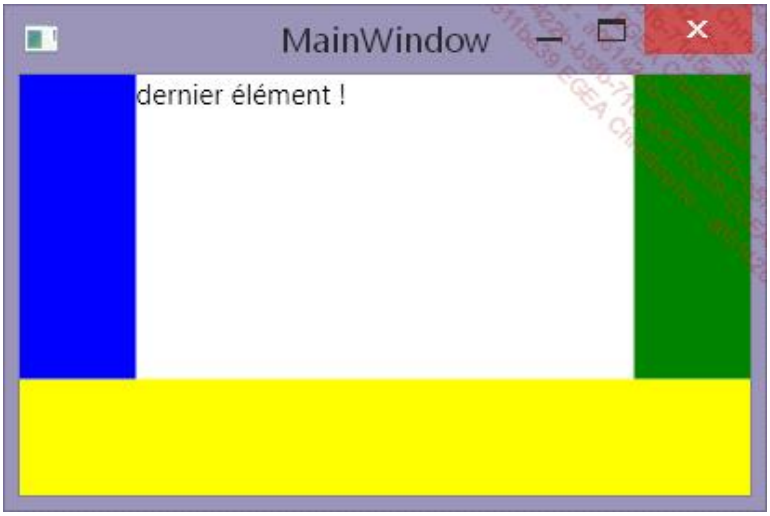
Le contrôle `DockPanel` donne la possibilité d'ancrer ses contrôles enfants sur chacun de ses quatre côtés. Pour cela, il faut utiliser la propriété attachée `DockPanel.Dock` sur les contrôles à ancrer.

```
<DockPanel>
    <Grid DockPanel.Dock="Right" Background="Green" Width="50" />
    <Grid DockPanel.Dock="Bottom" Background="Yellow" Height="50" />
    <Grid DockPanel.Dock="Left" Background="Blue" Width="50" />
    <TextBlock Text="dernier élément !" />
</DockPanel>
```

Ce code produit l'affichage suivant :

Il est important de noter que les contrôles sont ancrés dans l'ordre de leur instanciation. Ceci implique que le code précédent et l'exemple suivant n'aboutissent pas au même résultat.

```
<DockPanel>
    <Grid DockPanel.Dock="Bottom" Background="Yellow" Height="50" />
    <Grid DockPanel.Dock="Right" Background="Green" Width="50" />
    <Grid DockPanel.Dock="Left" Background="Blue" Width="50" />
    <TextBlock Text="dernier élément !" />
</DockPanel>
```



Plusieurs contrôles peuvent être ancrés au même bord : ceux-ci sont alors "empilés" relativement à ce bord, et dans l'ordre de leur instanciation.

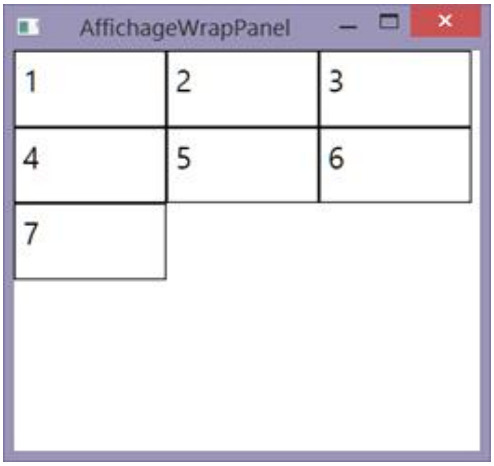
```
<DockPanel>
    <Grid DockPanel.Dock="Right" Background="Yellow" Width="50" />
    <Grid DockPanel.Dock="Right" Background="Green" Width="50" />
    <Grid DockPanel.Dock="Left" Background="Blue" Width="50" />
    <TextBlock Text="dernier élément !" />
</DockPanel>
```

La class `DockPanel` a une propriété `LastChildFill` qui permet d'indiquer quel doit être le comportement de son dernier contrôle enfant. Si cette propriété a pour valeur `True`, alors l'élément recouvre la totalité de l'espace restant dans le `DockPanel`.

d. WrapPanel

Le contrôle `WrapPanel` peut être décrit comme un `StackPanel` amélioré (bien que la classe `WrapPanel` n'hérite pas de `StackPanel`). En effet, de la même manière que le contrôle `StackPanel`, il permet d'accoler plusieurs composants graphiques.

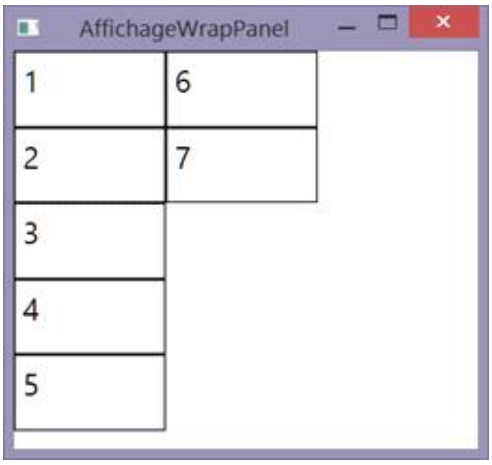
La véritable différence entre ces deux contrôles réside dans le fait que le `WrapPanel` renvoie le contenu sur une nouvelle ligne (ou colonne) quand l'espace disponible est insuffisant sur l'alignement en cours de remplissage.



Cet écran est le résultat du XAML suivant :

```
<WrapPanel>
    <Label Content="1" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="2" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="3" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="4" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="5" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="6" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
    <Label Content="7" FontSize="20" Height="50" Width="100"
BorderThickness="1" BorderBrush="Black" />
</WrapPanel>
```

Par défaut, la propriété `Orientation` du `WrapPanel` est `Horizontal`. En passant cette valeur à `Vertical`, on obtient le rendu suivant :



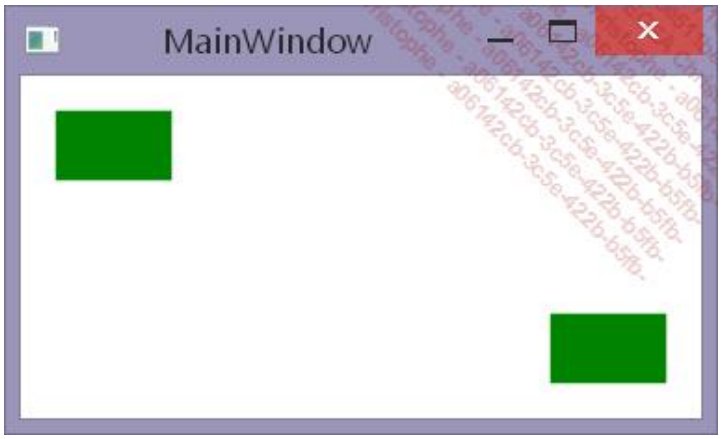
Le contrôle `WrapPanel` occupe par défaut tout l'espace disponible dans son contrôle parent, ce qui permet

d'avoir un comportement dynamique : si on redimensionne le parent à l'exécution, le `WrapPanel` est redimensionné aussi, et ses éléments enfants peuvent être repositionnés en fonction de ses nouvelles dimensions.

e. Canvas

Le contrôle `Canvas` est destiné à contenir des contrôles graphiques positionnés de manière absolue.

Le positionnement des éléments est mis en œuvre à l'aide des propriétés attachées `Canvas.Top`, `Canvas.Left`, `Canvas.Right` et `Canvas.Bottom`. Celles-ci permettent respectivement de spécifier la distance entre le contrôle positionné et le bord du contrôle `Canvas`. Le code suivant positionne deux rectangles, l'un à 15 pixels du bord gauche et du haut, l'autre à 15 pixels du bord droit et du bas.




```
<Canvas>
    <Rectangle Width="50" Height="30" Fill="Green"
Canvas.Top="15" Canvas.Left="15" />
    <Rectangle Width="50" Height="30" Fill="Green"
Canvas.Right="15" Canvas.Bottom="15" />
</Canvas>
```

3. Contrôles d'affichage de données

Toute application graphique a le besoin d'afficher des informations, qu'elles soient de type texte ou image, temporaires ou permanentes. Différents contrôles sont disponibles dans WPF pour gérer ces cas.

a. TextBlock

Le contrôle `TextBlock` est le contrôle le plus adapté pour l'affichage de texte. Son contenu est spécifié au travers de sa propriété `Text`.

```
<TextBlock Text="Développer avec VS2015 et C#" />
```

Plusieurs autres propriétés permettent de contrôler la manière dont le texte est affiché : couleur, police ou encore gestion des retours à la ligne.

Les principales sont listées dans le tableau suivant :

Text	Définit le texte affiché par le contrôle.
FontFamily	Définit la police de caractères utilisée par le contrôle.
FontSize	Définit la taille de police utilisée pour afficher le contenu.
FontStyle	Définit le style de la police : Italique, Normal ou Oblique (italique simulé par transformation si la police n'a pas de version Italique).
FontWeight	Définit la graisse de typographie (épaisseur du texte) utilisée pour le contenu.
Foreground	Définit la couleur du texte.
TextAlignment	Définit l'alignement du texte (gauche, droite, centré, justifié).

```
<TextBlock Text="Développer avec VS2015 et C#" FontSize="30"
FontFamily="Segoe Print"/>
```

b. Label

Le contrôle `Label`, dans son utilisation la plus simple, ressemble beaucoup à un contrôle `TextBlock`. Il n'a pourtant pas de propriété `Text` mais une propriété `Content`. La raison de cette différence est le fait que le contrôle `Label` peut contenir tout type de contrôle.

Ce contrôle implémente également une fonctionnalité bien plus intéressante. Il permet en effet de donner le focus à un contrôle qui lui est associé par le biais des Access Keys.

Les Access Keys sont des combinaisons de touches permettant d'accéder à un contrôle. Sous Windows, cette combinaison implique généralement la touche [Alt] et un caractère alphabétique. Pour utiliser cette fonctionnalité, il faut :

- Définir un `Label` et valoriser sa propriété `Content` avec une chaîne de caractères dont une lettre est précédée par le symbole _. Cette lettre sera par la suite utilisée dans la combinaison de touches [Alt] + [lettre].
- Définir un objet qui sera associé au `Label`. C'est celui-ci qui prendra le focus lors de l'utilisation de l'Access Key.
- Lier le `Label` et l'objet associé à l'aide de la propriété `Target` du `Label` en utilisant le binding suivant :

```
Target="{Binding ElementName=<nom de l'objet associé>}"
```

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <StackPanel Grid.Row="0" Orientation="Horizontal" Height="25">
```

```
        <Label Content="_Nom" Target="{Binding ElementName=tbNom}" />
        <TextBox x:Name="tbNom" Width="150" />
    </StackPanel>

    <StackPanel Grid.Row="1" Orientation="Horizontal" Height="25">
        <Label Content="_Prenom" Target="{Binding
ElementName=tbPrenom}" />
        <TextBox x:Name="tbPrenom" Width="150" />
    </StackPanel>
</Grid>
```

À l'exécution, un appui sur la touche [Alt] provoque la mise en évidence des Access Keys utilisables.

Ici, la combinaison de touches [Alt] N met le focus sur le champ d'édition Nom, tandis que l'appui simultané sur [Alt] et P donnera le focus au champ d'édition Prénom.

c. Image

Le contrôle `Image` permet de visualiser entre autres le contenu de fichiers bmp, jpeg ou png.

Sa propriété `Source` permet de définir l'emplacement de l'image à afficher. Cet emplacement peut être un chemin absolu ou relatif, une URI pointant sur les ressources de l'application ou même une URL. Dans ce dernier cas, le téléchargement de l'image est effectué automatiquement avant l'étape d'affichage.

La propriété `Stretch` de ce contrôle définit le mode d'étirement de l'image en fonction de l'espace disponible.

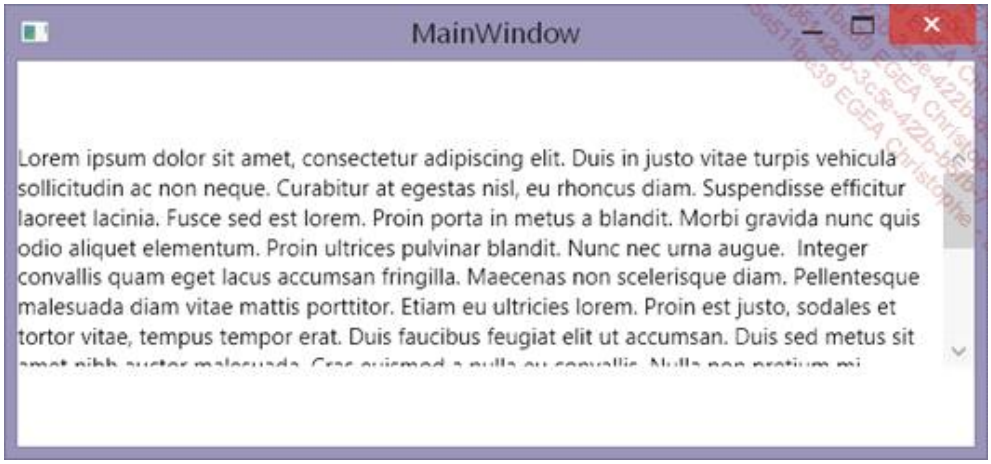
```
<Image Source="http://upload.wikimedia.org/wikipedia/commons/a/af/Jimi_Hendrix
_1967_uncropped.jpg" Stretch="Uniform" />
```

Ce code affiche le résultat suivant :

d. ScrollViewer

Certains contrôles ont besoin de plus d'espace que celui qui leur est alloué. Habituellement, ce type de cas est géré par l'utilisation de barres de défilement horizontales ou verticales, mais certains contrôles WPF ne disposent pas nativement de ces barres. Le contrôle `ScrollViewer` est une implémentation externalisée de ces barres qui peut être placée autour de n'importe quel contrôle afin de permettre le défilement.

```
<ScrollViewer Height="120" >
    <TextBlock TextWrapping="Wrap" Text="Placez ici un texte très
très long..." />
</ScrollViewer>
```



e. ItemsControl

L'affichage d'une liste d'éléments peut être réalisé par l'utilisation d'un objet de type `ItemsControl`. Celui-ci peut contenir une collection d'objets de n'importe quel type. Ces objets sont présentés par défaut sous la forme d'une liste verticale, sans barre de défilement. Lorsque le rendu de la liste est plus grand que le contrôle, la liste est tronquée. Pour gérer ce cas d'utilisation, il convient de placer le contrôle de liste dans un objet de type `ScrollViewer`.

Les valeurs à afficher sont fournies au contrôle par l'intermédiaire de sa propriété `Items`.

```
<ItemsControl Height="30">
    <ItemsControl.Items>
        <system:String>Choix n°1</system:String>
        <system:String>Choix n°2</system:String>
        <system:String>Choix n°3</system:String>
        <system:String>Choix n°4</system:String>
    </ItemsControl.Items>
</ItemsControl>
```

Pour ajouter des objets de type `string` au contrôle, l'espace de noms XML `system` a été ajouté à la fenêtre à l'aide de la déclaration suivante :

```
xmlns:system="clr-namespace:System;assembly=mscorlib"
```

La propriété `Items` étant la propriété par défaut du contrôle, il est possible de condenser le code en omettant les déclarations ouvrantes et fermantes qui concernent cette propriété :

```
<ItemsControl Height="30">
    <system:String>Choix n°1</system:String>
    <system:String>Choix n°2</system:String>
    <system:String>Choix n°3</system:String>
    <system:String>Choix n°4</system:String>
</ItemsControl >
```

Il arrive très souvent que les éléments contenus par le contrôle doivent être définis de manière dynamique. Pour cela, il est préférable d'utiliser la propriété `ItemsSource` en la valorisant avec la déclaration d'un binding.

```
<ItemsControl Height="30" ItemsSource="{Binding ListeDeChoix}" />
```

Dans les deux cas, chaque élément de la collection `ListeDeChoix` génère un élément dans la liste visuelle. Les deux sont liés : le contexte de données de l'élément visuel est l'élément correspondant de la liste `ListeDeChoix`.

Cette relation est importante lorsque les données fournies sont de type complexe. La représentation visuelle par défaut de chaque élément étant déterminée par l'utilisation de sa méthode `ToString`, le rendu visuel n'est souvent pas celui qui est souhaité.

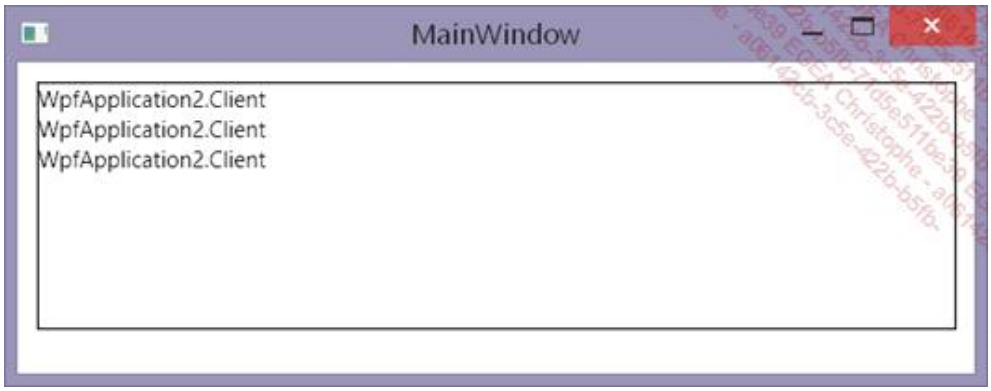
L'exemple suivant est basé sur le type de données défini ci-dessous.

```
public class Client
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public decimal MontantTotalAchats { get; set; }
}
```

```
<ItemsControl VerticalAlignment="Top" BorderBrush="Black"
BorderThickness="1" Margin="10" Height="150">
    <local:Client Nom="DUPOND" Prenom="Jean"
MontantTotalAchats="127.42" />
    <local:Client Nom="MARTIN" Prenom="Eric"
MontantTotalAchats="98.02" />
    <local:Client Nom="TUCQUE" Prenom="Sophie"
MontantTotalAchats="241.95" />
</ItemsControl>
```

L'espace de noms `local` est préalablement défini au niveau de l'élément racine du fichier.

Le rendu du contrôle en l'état est le suivant :

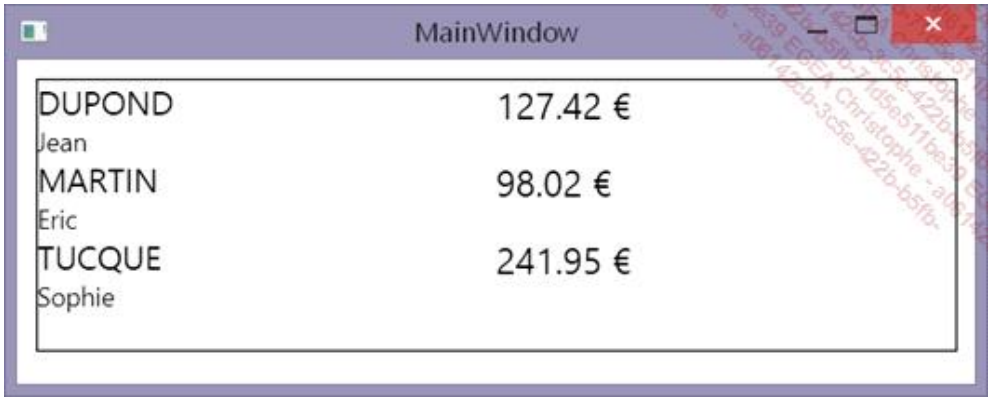


Pour améliorer la visualisation des données, il faut utiliser la propriété `ItemTemplate` du contrôle. Celle-ci accepte comme valeur un objet de type `DataTemplate` qui définit un modèle visuel pour le contexte de données.

Le code XAML suivant définit une grille contenant deux lignes et deux colonnes dans lesquelles les trois propriétés de la classe `Client` sont placées à l'aide d'expressions de binding.

```
<ItemsControl VerticalAlignment="Top" BorderBrush="Black"
BorderThickness="1" Margin="10" Height="150">
    <local:Client Nom="DUPOND" Prenom="Jean"
MontantTotalAchats="127.42" />
    <local:Client Nom="MARTIN" Prenom="Eric"
MontantTotalAchats="98.02" />
    <local:Client Nom="TUCQUE" Prenom="Sophie"
MontantTotalAchats="241.95" />

    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition />
                    <ColumnDefinition />
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition />
                    <RowDefinition />
                </Grid.RowDefinitions>

                <TextBlock Grid.Row="0" Grid.Column="0"
FontSize="18" Text="{Binding Nom}" />
                <TextBlock Grid.Row="1" Grid.Column="0"
FontSize="14" Text="{Binding Prenom}" />
                <TextBlock Grid.Row="0" Grid.Column="1"
Grid.RowSpan="2" FontSize="20" Text="{Binding
MontantTotalAchats, StringFormat='{}{0:N2} €'}" />
            </Grid>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

Le rendu de l'interface tient compte du `DataTemplate` fourni et adapte l'affichage en conséquence, ce qui donne le résultat suivant :



f. StatusBar

Le contrôle `StatusBar` représente une barre de statut habituellement placée en bas d'une fenêtre d'application. Cette barre contient des informations sur le travail actuellement en cours : numéro de page, état de la sauvegarde d'un fichier, etc.

Ce contrôle peut contenir des objets de type `StatusBarItem`, chacun contenant les contrôles nécessaires à l'affichage d'une information. Il est possible de marquer une séparation entre ces objets à l'aide du contrôle `Separator`.

```
<DockPanel LastChildFill="False">
    <StatusBar DockPanel.Dock="Bottom" Height="30">
        <StatusBarItem>Aucun fichier ouvert</StatusBarItem>
        <Separator />
        <StatusBarItem>Chargement du concepteur en
cours...</StatusBarItem>
    </StatusBar>
</DockPanel>
```

g. ToolTip

Le contrôle `ToolTip` (**infobulle**) permet d'afficher un message d'information lorsque le curseur de la souris est arrêté sur un contrôle. Un cas d'utilisation fréquent est l'affichage d'un message explicatif lorsqu'une saisie est erronée (format d'e-mail incorrect, par exemple).

Les composants graphiques possèdent une propriété `ToolTip` qui correspond au contenu à afficher. Ce contenu, dans la forme la plus simple, est une chaîne de caractères, mais il peut être plus complexe pour afficher une image

avec du texte, par exemple.

```
<TextBlock Text="Ce TextBlock a un Tooltip">
    <TextBlock.ToolTip>
        <ToolTip>
            <TextBlock Text="Ceci est une infobulle" />
        </ToolTip>
    </TextBlock.ToolTip>
</TextBlock>
```

Cette portion de code donne le résultat suivant :

4. Contrôles d'édition de texte

Les contrôles de saisie sont des points d'entrée privilégiés pour l'enregistrement d'informations par les utilisateurs. Ils permettent en effet une certaine liberté au niveau du contenu et constituent donc de véritables points de création de données.

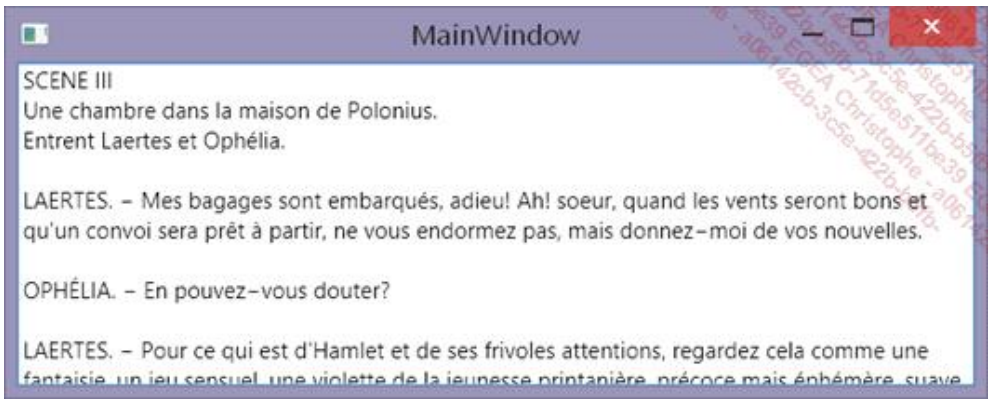
a. TextBox

Le contrôle `TextBox` est le contrôle le plus simple et le plus utilisé pour l'édition de texte. Son contenu est défini par sa propriété `Text`. Par défaut, il est impossible de revenir à la ligne dans ce contrôle.

La propriété `AcceptsReturn`, lorsqu'elle est définie à `true`, autorise le retour à la ligne à l'aide de la touche [Entrée]. Une seconde propriété permet le retour à la ligne automatique lorsque le texte est plus long que le contrôle : `TextWrapping`. Il suffit alors de définir sa valeur à `Wrap` pour obtenir le résultat souhaité.

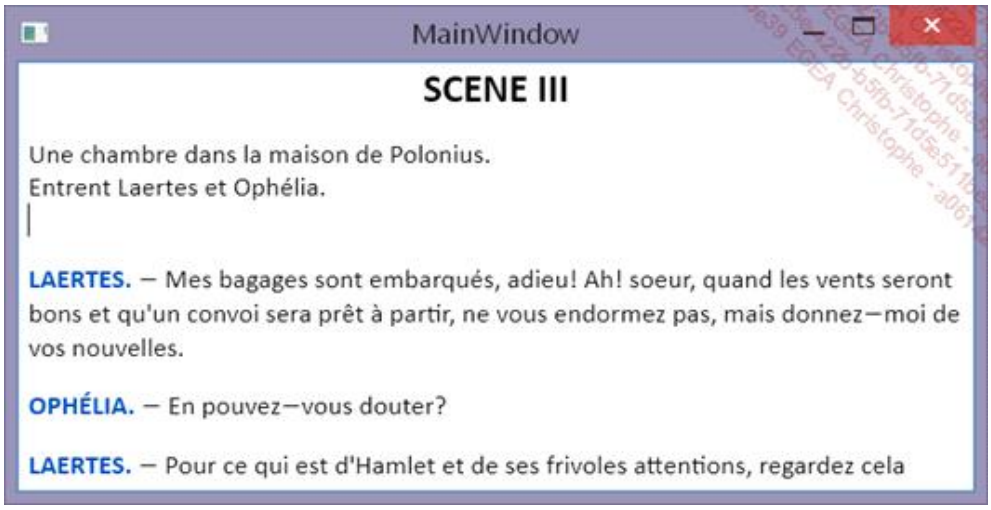
Le code suivant permet d'obtenir le résultat ci-dessous :

```
<TextBox AcceptsReturn="True" TextWrapping="Wrap" />
```



b. RichTextBox

Le texte présenté ci-dessus peut paraître difficile à lire, tous les éléments de texte étant formatés de la même manière. Le contrôle `RichTextBox` permet de s'affranchir de cette limitation en affichant du texte au format RTF (comme l'application WordPad de Windows). En éditant ce texte dans WordPad et en effectuant un copier/coller vers un composant `RichTextBox`, on peut obtenir ceci :



En l'état, il n'est pas possible de modifier le format de ce texte. En effet, WPF ne fournit pas la barre d'outils nécessaire à une utilisation avancée du contrôle `RichTextBox`. Mais le contrôle fournit les propriétés et méthodes nécessaires à son extension. La propriété `Selection` permet de récupérer la zone de texte sélectionnée au travers d'un objet de type `TextSelection`. Ce type implémente notamment la méthode `ApplyPropertyValue(DependencyProperty formattingProperty, object value)` qui permet de modifier une propriété du texte sélectionné.

Ici, nous ajoutons un bouton permettant de modifier la graisse du texte sélectionné. Le fichier `MainWindow.xaml.cs` définit un gestionnaire pour l'événement `Click` du bouton placé au-dessus du contrôle `RichTextBox`.

- `MainWindow.xaml` :

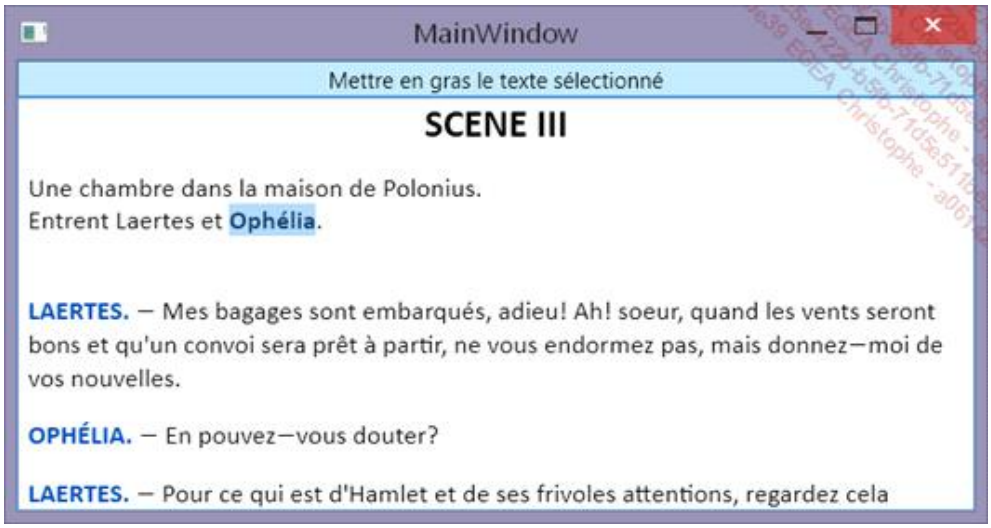
```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
    </Grid.RowDefinitions>

    <Button Grid.Row="0" Content="Mettre en gras le texte
sélectionné" Click="Button_Click" />
    <RichTextBox Grid.Row="1" AcceptsReturn="True"
x:Name="zoneSaisie" />
</Grid>
```

- `MainWindow.xaml.cs` :

```
private void Button_Click(object sender, RoutedEventArgs e)
{
```

```
    zoneSaisie.Selection.ApplyPropertyValue(FontWeightProperty,
FontWeights.Bold);
}
```

Le résultat est celui-ci :



Le fonctionnement du contrôle `Button` est détaillé plus loin dans ce chapitre.

c. PasswordBox

Les règles de sécurité informatique incluent toutes le non-partage des mots de passe afin d'éviter toute utilisation non désirée d'un compte applicatif. Le premier moyen d'éviter qu'un mot de passe ne tombe entre de mauvaises mains est son masquage. Cette solution est d'ailleurs utilisée par la quasi-totalité des éditeurs.

Avec WPF, l'implémentation de ce masquage peut être effectuée à l'aide du contrôle `PasswordBox`. Celui-ci remplace automatiquement les caractères saisis par le caractère spécifié dans sa propriété `PasswordChar`, tout en conservant précieusement ce qui a été saisi dans sa propriété `Password`.

5. Contrôles de sélection

Les contrôles de sélection sont nettement moins permissifs que les contrôles de saisie, puisqu'ils permettent de choisir une ou plusieurs options parmi une liste finie d'éléments. Mais ce comportement leur permet en contrepartie d'être une aide précieuse pour les utilisateurs puisqu'il permet d'accélérer le remplissage de formulaires tout en réduisant les possibilités d'erreurs lors d'opérations de catégorisation ou de saisie répétitive.

a. RadioButton

Le contrôle `RadioButton` représente une option de sélection exclusive qui peut être cochée ou non cochée. Cet état peut être récupéré à l'aide de la propriété `IsChecked`. Lorsque le contrôle est coché, il est impossible de le décocher en cliquant à nouveau dessus. Il faut pour cela cocher un autre contrôle `RadioButton` appartenant au même groupe. Le groupe auquel appartient le contrôle est défini par la propriété `GroupName`.

```
<StackPanel Orientation="Horizontal">
    <RadioButton GroupName="Choix" Content="Choix n°1" />
```

```
    <RadioButton GroupName="Choix" Content="Choix n°2" />
</StackPanel>
```

Ce code produit le résultat affiché ci-dessous.

Ce contrôle est rarement utilisé lorsque plus de cinq choix sont proposés. Dans ce type de cas, le contrôle `ComboBox` est souvent préféré car il est moins encombrant qu'un empilement de `RadioButton` et est au moins aussi lisible.

b. CheckBox

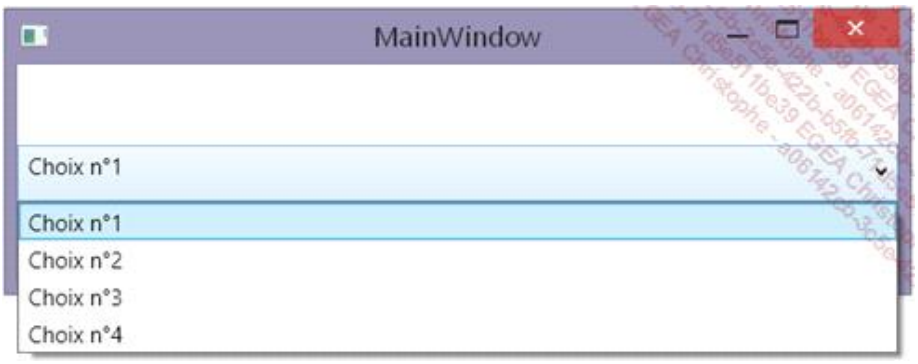
Le contrôle `CheckBox` permet, comme le `RadioButton`, de sélectionner ou non une valeur, mais son mode de sélection n'est pas exclusif : plusieurs `CheckBox` peuvent être cochées. L'état de chaque `CheckBox` est défini par la valeur de sa propriété `IsChecked`.

L'utilisation du code suivant permet d'afficher deux contrôles `CheckBox` :

```
<StackPanel Orientation="Horizontal">
    <CheckBox Content="Choix n°1" />
    <CheckBox Content="Choix n°2" />
</StackPanel>
```

c. ComboBox

Ce contrôle propose une liste déroulante d'éléments dont un seul au maximum peut être sélectionné. Ce contrôle est souvent apprécié pour les sélections uniques en raison de sa simplicité d'utilisation et de son encombrement minimal.



Ce contrôle est dérivé du type `ItemsControl`, ce qui implique que leur utilisation est similaire. Les données à afficher sont fournies par l'intermédiaire des propriétés `Items` ou `ItemsSource`. Cette seconde propriété est destinée à être valorisée à l'aide d'une expression de binding.

De la même manière que pour le type `ItemsControl`, il est possible de personnaliser la présentation des éléments affichés en valorisant la propriété `ItemTemplate` du contrôle avec un objet de type `DataTemplate`.

L'utilisation des `DataTemplate` pour la personnalisation de l'affichage est détaillée en même temps que le type `ItemsControl` (cf. section ItemsControl).

Lorsqu'un objet est sélectionné, sa valeur est enregistrée dans la propriété `SelectedItem` du contrôle.

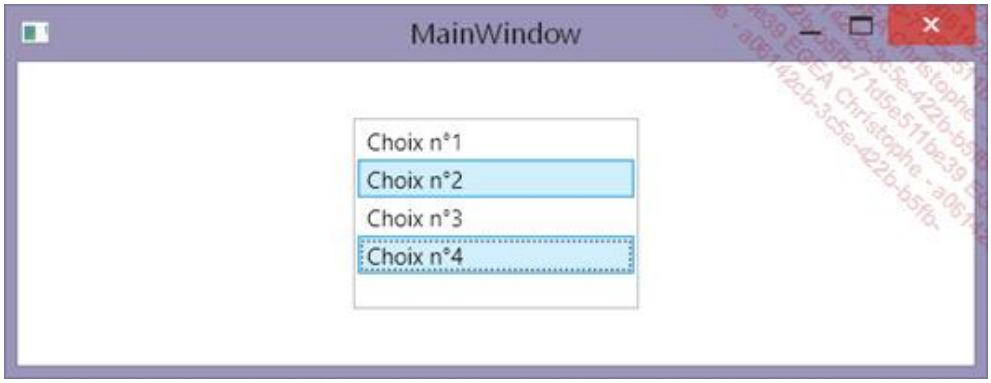
d. ListBox

Le contrôle `ListBox` affiche une liste d'éléments dont un ou plusieurs sont sélectionnables. Le mode de sélection est défini par la valeur de la propriété `SelectionMode` du contrôle : `Single` signifie qu'une seule valeur est sélectionnable, tandis que `Multiple` et `Extended` permettent la sélection multiple. La différence entre ces deux derniers modes est la suivante :

- `Multiple` : la sélection ou désélection d'un élément est faite par un simple clic.
- `Extended` : il est nécessaire de maintenir la touche [Ctrl] appuyée pendant les opérations de sélection/désélection.

La liste de données contenues par le contrôle est fournie par l'intermédiaire de ses propriétés `Items` ou `ItemsSource`, de la même manière que pour le type `ItemsControl`. Comme pour les autres contrôles permettant l'affichage de collections, il est possible de personnaliser le rendu en valorisant la propriété `ItemTemplate` avec un objet de type `DataTemplate`.

L'utilisation des `DataTemplate` pour la personnalisation de l'affichage est détaillée en même temps que le type `ItemsControl` (cf. section ItemsControl).



Ce résultat est obtenu en utilisant l'extrait de code ci-dessous.

```
<ListBox SelectionMode="Multiple" Height="100" Width="150">
    <ListBox.Items>
        <system:String>Choix n°1</system:String>
        <system:String>Choix n°2</system:String>
        <system:String>Choix n°3</system:String>
```

```
        <system:String>Choix n°4</system:String>
    </ListBox.Items>
</ListBox>
```

Lorsqu'un seul élément est sélectionné, sa valeur est stockée dans la propriété `SelectedItem` du contrôle. Si plusieurs valeurs sont sélectionnées, il est possible de les récupérer dans la propriété `SelectedItems`.

e. ListView

Le contrôle `ListView`, dérivé du type `ItemsControl`, permet d'afficher une liste d'éléments sous la forme d'une grille. Celle-ci est définie à l'aide de la propriété `View` qui est de type `ViewBase`. La seule classe du framework .NET qui hérite de ce type est la classe `GridView`, c'est donc un objet de ce type que nous passerons à la propriété `View`.

Il est tout à fait possible de définir un autre format de présentation en développant un contrôle héritant de `ViewBase`.

Les éléments affichés par le contrôle sont, comme pour le type `ItemsControl`, passés au travers des propriétés `Items` ou `ItemsSource`.

Le type de données utilisé dans cet exemple est défini comme suit :

```
public class Client
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public decimal MontantTotalAchats { get; set; }
}
```

Une fois ce type déclaré, il faut ajouter des éléments à la propriété `Items` du contrôle `ListView`.

```
<ListView>
    <ListView.Items>
        <local:Client Nom="DUPOND" Prenom="Jean"
MontantTotalAchats="127.42" />
        <local:Client Nom="MARTIN" Prenom="Eric"
MontantTotalAchats="98.02" />
        <local:Client Nom="TUCQUE" Prenom="Sophie"
MontantTotalAchats="241.95" />
    </ListView.Items>
</ListView>
```

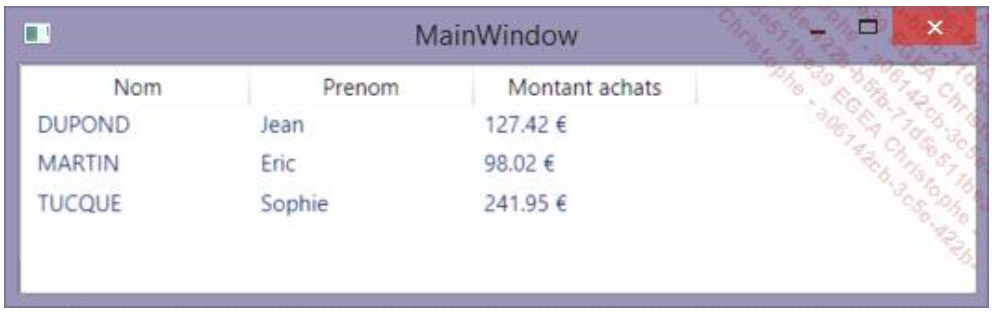
L'espace de noms `local` est défini dans la déclaration du contrôle `Window` parent. L'utilisation des espaces de noms est détaillée dans la première partie de ce chapitre (section XAML).

En l'état, ce contrôle a le même aspect qu'un contrôle `ListBox` : il affiche chacun de ses éléments en exécutant leur méthode `ToString` sous la forme d'une liste verticale.

La mise en place du format de la grille est effectuée en valorisant la propriété `View` avec un objet de type `GridView`. Cette grille doit contenir une ou plusieurs colonnes. Pour chacune, il faut définir la valeur à afficher en valorisant sa propriété `DisplayMemberBinding` avec une expression de binding.

```
<ListView>
    <ListView.Items>
        <local:Client Nom="DUPOND" Prenom="Jean"
MontantTotalAchats="127.42" />
        <local:Client Nom="MARTIN" Prenom="Eric"
MontantTotalAchats="98.02" />
        <local:Client Nom="TUCQUE" Prenom="Sophie"
MontantTotalAchats="241.95" />
    </ListView.Items>
    <ListView.View>
        <GridView>
            <GridViewColumn Header="Nom" Width="120"
DisplayMemberBinding="{Binding Nom}" />
            <GridViewColumn Header="Prenom" Width="120"
DisplayMemberBinding="{Binding Prenom}" />
            <GridViewColumn Header="Montant achats" Width="120"
DisplayMemberBinding="{Binding MontantTotalAchats,
StringFormat={}{0} €}" />
        </GridView>
    </ListView.View>
</ListView>
```

Le résultat obtenu est présenté ci-dessous :



f. TreeView

Le contrôle `TreeView` offre la possibilité de visualiser des données sous une forme hiérarchique. Chacun des objets peut en effet avoir un ou plusieurs enfants, qui peuvent eux-mêmes être les parents d'autres nœuds, et ainsi de suite.

WPF permet de décrire cette hiérarchie simplement et clairement. Il suffit en effet de considérer que chaque nœud de notre arbre est un objet de type `TreeViewItem` qui peut contenir un ou plusieurs autres objets du même type.

Le texte correspondant à chacun des nœuds est valorisé par l'utilisation de la propriété `Header` de chaque objet `TreeViewItem`. Cette propriété est de type `object`, ce qui signifie indirectement qu'il n'est pas obligatoire de la valoriser avec une chaîne de caractères, et qu'elle peut donc contenir des images, du texte, une case à cocher, etc.

Le `TreeView` présenté ci-dessus est codé de la manière suivante :

```
<TreeView>
    <TreeViewItem Header="Amérique">
        <TreeViewItem Header="Amérique du Nord">
            <TreeViewItem Header="Canada" />
            <TreeViewItem Header="Etats-Unis" />
        </TreeViewItem>
        <TreeViewItem Header="Amérique centrale">
            <TreeViewItem Header="Mexique" />
            <TreeViewItem Header="Panama" />
            <TreeViewItem Header="Honduras" />
        </TreeViewItem>
        <TreeViewItem Header="Amérique du Sud">
            <TreeViewItem Header="Brésil" />
            <TreeViewItem Header="Argentine" />
```

```
            <TreeViewItem Header="Chili" />
            <TreeViewItem Header="Pérou" />
            <TreeViewItem Header="Colombie" />
        </TreeViewItem>
    </TreeViewItem>
    <TreeViewItem Header="Europe">
        <TreeViewItem Header="Union européenne">
            <TreeViewItem Header="Zone Euro">
                <TreeViewItem Header="Portugal" />
                <TreeViewItem Header="France" />
                <TreeViewItem Header="Luxembourg" />
                <TreeViewItem Header="Irlande" />
                <TreeViewItem Header="Finlande" />
                <TreeViewItem Header="Italie" />
                <TreeViewItem Header="Autriche" />
            </TreeViewItem>
            <TreeViewItem Header="Royaume-Uni" />
            <TreeViewItem Header="Danemark" />
            <TreeViewItem Header="Suède" />
        </TreeViewItem>
        <TreeViewItem Header="Suisse" />
    </TreeViewItem>
</TreeView>
```

La classe `TreeViewItem` dérive de la classe `HeaderedItemsControl` qui représente une liste possédant un en-tête. Cet en-tête est l'élément visualisé comme étant le nœud, tandis que la partie affichant les éléments de la collection est en fait représentée visuellement par les nœuds enfants. L'écriture complète de la déclaration d'un `TreeViewItem` et ses enfants est en fait la suivante :

```
<TreeViewItem Header="Amérique du Nord">
    <TreeViewItem.Items>
        <TreeViewItem Header="Canada" />
        <TreeViewItem Header="Etats-Unis" />
    </<TreeViewItem.Items>
</TreeViewItem>
```

L'utilisation de la propriété `ItemsSource` de la classe `TreeViewItem` peut s'avérer un peu plus délicate. Celle-ci impose l'utilisation d'un `ItemTemplate` afin de gérer la hiérarchie. Le type d'objet à fournir est ici `HierarchicalDataTemplate`, qui est un type d'objet dérivé de la classe `DataTemplate`. La mise en œuvre de son utilisation est ici détaillée pas à pas.

Tout d'abord, il faut créer le `TreeView` ainsi que la structure de données à partir de laquelle sera peuplé le `TreeView`.

- `MainWindow.xaml.cs` :

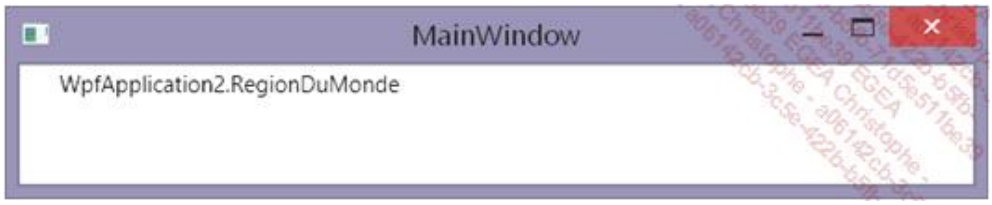
```
public class RegionDuMonde
{
    public string NomRegion { get; set; }
    public List<Pays> ListePays { get; set; }
}
```

```
public class Pays
{
    public string Nom { get; set; }
}

public partial class MainWindow : Window
{
    public List<RegionDuMonde> Regions { get; set; }

    public MainWindow()
    {
        InitializeComponent();

        Regions = new List<RegionDuMonde>
        {
            new RegionDuMonde
            {
                NomRegion = "Europe",
                ListePays = new List<Pays>
                {
                    new Pays { Nom = "Royaume-Uni" },
                    new Pays { Nom = "Danemark" },
                    new Pays { Nom = "Suède" }
                }
            }
        };


        this.DataContext = this;
    }
}
```

- Définition du `TreeView` dans `MainWindow.xaml` :

```
<TreeView ItemsSource="{Binding Regions}">
</TreeView>
```

Ce qui donne à ce stade le résultat suivant :



La classe `HierarchicalDataTemplate` héritant de `DataTemplate`, il est possible de spécifier l'`ItemTemplate` du `TreeView` comme étant de ce type.

```
<TreeView ItemsSource="{Binding Regions}">
    <TreeView.ItemTemplate>
        <HierarchicalDataTemplate>
            <TextBlock Text="{Binding NomRegion}" />
```

```
        </HierarchicalDataTemplate>
    </TreeView.ItemTemplate>
</TreeView>
```

Le `TreeView` a maintenant l'aspect suivant :

La classe `HierarchicalDataTemplate` possède une propriété `ItemsSource` qu'il faut ici lier à la liste de pays :

```
...
<HierarchicalDataTemplate ItemsSource="{Binding ListePays}">
...
```

Comme pour tous les `ItemsControl`, pour obtenir le résultat obtenu, il est nécessaire de fournir un `DataTemplate` pour les objets de type `Pays`. La classe `HierarchicalDataTemplate` a une propriété `ItemTemplate` permettant de spécifier le modèle de données à appliquer pour ces objets.

```
<TreeView ItemsSource="{Binding Regions}">
    <TreeView.ItemTemplate>
        <HierarchicalDataTemplate ItemsSource="{Binding ListePays}">
            <TextBlock Text="{Binding NomRegion}" />
            <HierarchicalDataTemplate.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding NomPays}" />
                </DataTemplate>
            </HierarchicalDataTemplate.ItemTemplate>
        </HierarchicalDataTemplate>
    </TreeView.ItemTemplate>
</TreeView>
```

Le résultat correspond à ce qui était attendu :

g. Slider

Ce contrôle permet la sélection d'une valeur numérique comprise entre deux bornes. L'utilisateur effectue son choix en faisant glisser un curseur le long d'une piste. Ce type de contrôle est souvent utilisé dans les applications multimédias pour gérer le volume sonore ou l'emplacement de lecture actuel d'un fichier audio ou vidéo.

Les bornes minimales et maximales sont définies par la valorisation des propriétés `Minimum` et `Maximum`, tandis que l'intervalle entre les valeurs sélectionnables est défini par la valeur de la propriété `SmallChange`. Si cette dernière propriété n'est pas valorisée, il est possible de sélectionner n'importe quelle valeur décimale supportée par le type `double`. La propriété `LargeChange` permet quant à elle de définir le nombre d'unités que le curseur doit parcourir vers l'avant ou l'arrière lorsque l'utilisateur clique sur la piste mais ne fait pas glisser le curseur.

h. Calendar

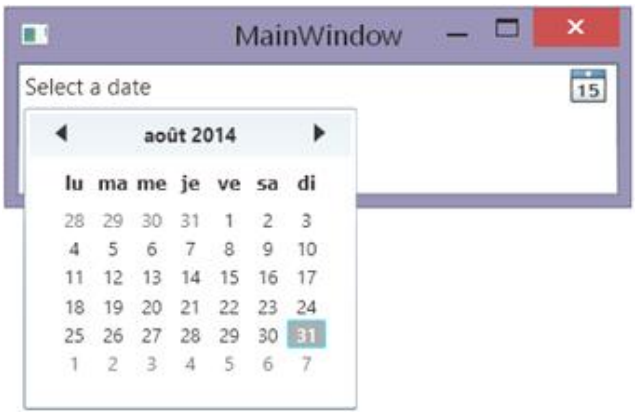
Ce contrôle permet de naviguer visuellement dans un calendrier afin de sélectionner une date, à la manière du calendrier intégré à la barre des tâches de Windows.

La date sélectionnée est représentée par la propriété `SelectedDate`.

i. DatePicker

Les saisies de date peuvent s'avérer particulièrement compliquées à implémenter, c'est pourquoi le framework .NET fournit le contrôle `DatePicker`. Celui-ci est composé de trois parties : un champ de saisie, un calendrier pouvant être affiché ou masqué, et enfin un bouton permettant d'afficher le calendrier. Il permet donc la saisie directe et valide que les données qu'il reçoit correspondent à des dates. La sélection de la date à partir du calendrier permet d'éviter tout problème concernant le format des données et est donc souvent privilégiée.

La propriété `SelectedDate` donne accès en lecture et en écriture à la valeur sélectionnée ou saisie dans le contrôle.



6. Contrôles d'action

À la différence des applications consoles dont le cycle de vie est très simple puisqu'elles utilisent un mode d'exécution séquentiel, les applications graphiques sont souvent composées d'une multitude de modules exécutant des tâches spécifiques lorsque l'utilisateur le souhaite. Pour exécuter ces tâches spécifiques, un des moyens existants est l'utilisation de contrôles d'action.

a. Button

Le contrôle `Button` est sans doute le plus utilisé des contrôles d'action. Son mode de fonctionnement est simple et intuitif : il s'agit d'un contrôle dont l'apparence est celle d'un bouton sur lequel un texte est écrit, ce qui indique son utilité.

Avec WPF, ce contrôle ne permet pas seulement d'afficher du texte. En effet, sa propriété `Content` est de type `Object`, il est donc possible de lui assigner tout contrôle WPF : `Image`, `Grid` ou `StackPanel` avec du contenu, du texte ou même un contrôle personnalisé.

Ce contrôle dispose également d'un événement `Click` permettant d'exécuter une portion de code en réaction au clic d'un utilisateur sur le contrôle.

```
<Button Content="Cliquez !" Click="Button_Click" />
```

b. Menu

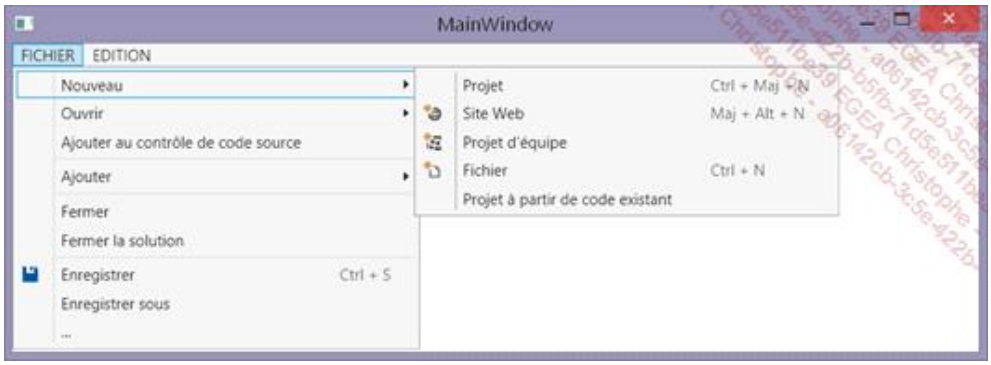
Le contrôle `Menu` est principalement utilisé pour présenter des actions communes à toute l'application. Il est représenté visuellement par une barre située généralement en haut de la fenêtre principale d'une application. Il est évidemment possible de placer différents contrôles `Menu` sur les différentes fenêtres composant un programme.

Chacun des éléments composant cette barre peut à son tour contenir d'autres éléments. Cette structure est similaire à celle d'un `TreeView`, et le code permettant sa création est par conséquent très similaire à celui permettant de définir un `TreeView` et son contenu. Le contrôle `Menu` contient des éléments `MenuItem` pouvant eux-mêmes contenir d'autres éléments de type `MenuItem`. Le texte indiquant l'utilité de chacun des boutons est valorisé par l'intermédiaire de la propriété `Header` de chaque objet de type `MenuItem`. Une image peut être accolée au texte en passant un objet `Image` à la propriété `Icon`, tandis qu'un texte indiquant le raccourci-clavier associé au `MenuItem` est spécifié à l'aide de la propriété `InputGestureText`.

Attention, `InputGestureText` est uniquement une propriété d'affichage. Le fait de la valoriser n'associe pas la

- combinaison de touches et l'exécution de l'action relative au `MenuItem`.

Pour associer une action à un `MenuItem`, il convient d'associer un gestionnaire d'événements à sa propriété `Click` de la même manière que pour un contrôle `Button`.



Le menu ci-dessus est généré à partir du code suivant :

```
<Menu Height="20" VerticalAlignment="Top">
    <MenuItem Header="FICHIER">
        <MenuItem Header="Nouveau">
            <MenuItem Header="Projet" InputGestureText="Ctrl + Maj + N" />
            <MenuItem Header="Site Web" InputGestureText="Maj + Alt + N">
                <MenuItem.Icon>
                    <Image Source="NewWebSite_6288.png" />
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Header="Projet d'équipe" >
                <MenuItem.Icon>
                    <Image Source="NewTeamProject_7437.png" />
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Header="Fichier" InputGestureText="Ctrl + N">
                <MenuItem.Icon>
                    <Image Source="NewFile_6276.png" />
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Header="Projet à partir de code existant" />
        </MenuItem>
        <MenuItem Header="Ouvrir">
            <MenuItem Header="Projet/Solution"
InputGestureText="Ctrl + Maj + O" />
            <MenuItem Header="Site Web" InputGestureText="Maj + Alt + O" />
            <Separator />
            <MenuItem Header="Ouvrir depuis le contrôle de code
source" InputGestureText="Maj + Alt + N" />
            <MenuItem Header="Projet d'équipe"  />
            <Separator />
            <MenuItem Header="Fichier" InputGestureText="Ctrl + O" />
            <MenuItem Header="Convertir" />
        </MenuItem>
        <MenuItem Header="Ajouter au contrôle de code source" />
        <Separator />
```

```
        <MenuItem Header="Ajouter">
            <MenuItem Header="Nouveau projet" />
            <MenuItem Header="Nouveau site Web" /
            <Separator />
            <MenuItem Header="Projet existant" />
            <MenuItem Header="Site Web existant" />
        </MenuItem>
        <Separator />
        <MenuItem Header="Fermer" />
        <MenuItem Header="Fermer la solution" />
        <Separator />
        <MenuItem Header="Enregistrer" InputGestureText="Ctrl + S">
            <MenuItem.Icon>
                <Image Source="Save_6530.png" />
            </MenuItem.Icon>
        </MenuItem>
        <MenuItem Header="Enregistrer sous" />

        <MenuItem Header="..." />
    </MenuItem>
    <MenuItem Header="EDITION">
        <MenuItem Header="Annuler" InputGestureText="Ctrl + Z" />
        <MenuItem Header="Rétablir" InputGestureText="Ctrl + Y" />
        <MenuItem Header="Annuler la dernière action globale" />
        <MenuItem Header="Rétablir la dernière action globale" />

        <MenuItem Header="..." />
    </MenuItem>
</Menu>
```

Les images utilisées pour la réalisation de ce menu proviennent de la **Bibliothèque d'images Visual Studio 2013**, disponible à l'adresse suivante : https://www.microsoft.com/en-us/download/details.aspx?id=35825

Comme l'objet `TreeView`, il est possible de construire un `Menu` en valorisant sa propriété `ItemsSource` et en fournissant un `ItemTemplate` de type `HierarchicalDataTemplate`.

c. ContextMenu

La création d'un menu contextuel est réalisée en WPF avec l'aide du contrôle `ContextMenu`. Celui-ci est conçu sur la même base que le contrôle `Menu` : ils héritent tous deux de la classe `MenuBase`. La conception et l'utilisation de ces deux contrôles sont donc très similaires. L'objet `ContextMenu` contient lui aussi des éléments de type `MenuItem` qu'il est possible d'imbriquer. Les différences principales entre ces contrôles sont que le menu contextuel est lié à un contrôle par la propriété `ContextMenu` du contrôle et qu'il apparaît à la demande lorsque l'utilisateur effectue un clic droit.

Le code suivant crée un contrôle `TextBox` et lui assigne un menu contextuel.

```
<TextBox Height="25" VerticalAlignment="Top" Margin="10">
    <TextBox.ContextMenu>
        <ContextMenu>
            <MenuItem Header="Copier" InputGestureText="Ctrl + C" />
            <MenuItem Header="Coller" InputGestureText="Ctrl + V" />
```

```
            <MenuItem Header="Supprimer" InputGestureText="Suppr" />
        </ContextMenu>
    </TextBox.ContextMenu>
</TextBox>
```

d. ToolBar

Les contrôles `ToolBar` et `ToolBarTray` permettent de créer une barre d'outils complète pour une fenêtre.

Chaque contrôle `ToolBar` constitue un groupement de contrôles fournissant des fonctionnalités similaires ou liées entre elles. Le conteneur dédié aux contrôles de type `ToolBar` est représenté par le type `ToolBarTray`. Celui-ci gère les opérations de placement, de redimensionnement et de glisser-déposer des éléments `ToolBar`. Les différentes `Toolbar` sont positionnées dans leur conteneur à l'aide des propriétés `Band` et `BandIndex`, représentant respectivement l'index de la bande dans laquelle le contrôle doit être placé et l'emplacement dans la bande par rapport aux autres `Toolbar`.

Un contrôle `ToolBar` peut contenir tout type de contrôles : boutons, listes déroulantes, cases à cocher, etc.

Le code suivant permet de générer une barre d'outils complète répartie sur deux bandes.

```
<ToolBarTray VerticalAlignment="Top">
    <ToolBar Band="1" BandIndex="1">
        <Button>
            <Image Source="Cut_6523.png" />
        </Button>
        <Button>
            <Image Source="Copy_6524.png" />
        </Button>
        <Button>
            <Image Source="Paste_6520.png" />
        </Button>
        <Separator />
        <Button>
            <Image Source="Undo_16x.png" />
        </Button>
        <Button>
            <Image Source="Redo_16x.png" />
        </Button>
    </ToolBar>
    <ToolBar Band="2" BandIndex="1">
        <Button>
            <Image Source="Save_6530.png" />
        </Button>
```

```
    </ToolBar>
</ToolBarTray>
```