Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Le Data Binding

Version 1.1

Sommaire

1 Introduction

Dans les chapitres précédents on a pu parler à plusieurs reprises de binding sans détailler plus, on y arrive enfin.

Le but du data binding c'est de permettre de lier un contrôle avec une ou plusieurs sources de données. C'est à dire que, comme on le verra, via plusieurs "mode" on va pouvoir mettre à jour automatiquement un contrôle en fonction d'une ou plusieurs sources.

Avant WPF, on n'avait pas la possibilité d'utiliser cette technique alors comment cela fonctionnait-il ? Concrètement, on devait faire un check de notre source de donnée éventuellement utiliser des événements et faire la liaison à la main. On verra donc tout au long de ce chapitre comment utiliser le data binding, les différents modes d'utilisation. Mais également comment gérer le comportement et l'apparence de ce binding et pour finir les différentes sources de données qui vont pouvoir être bindées vers un contrôle.

2 Initiation au Data Binding

2.1 Les bases du Data Binding

Nous allons commencer notre étude sur le Data Binding en énonçant un vocabulaire correct en ce qui concerne ce Data Binding.

Lorsque qu'on parle de Data Binding, comme on l'a vu, on va lier un contrôle à une source de données, ces sources de données seront détaillés par la suite, on appelle le contrôle qui va définir le binding la **cible** et notre source de données va être tout simplement nommé **source**.

Il est important de se souvenir de ces termes car ils seront utilisés tout au long de ce cours.

La mise en place du Data Binding est vraiment facile, tout va se faire en XAML. En effet, il va nous être proposé différentes propriétés pour les contrôles qui vont nous permettre de définir notre DataBinding.

Notamment des propriétés comme *ElementName* qui vont nous servir à spécifier le nom de l'élément qui sera utilisé comme source de notre Data Binding. Ou encore la propriété *Mode* qui va, comme on pourra le voir juste après, nous permettre de définir plusieurs mode de Binding différents, cela sera extrêmement utile. Pour finir on pourra également se pencher sur la propriété *UpdateSourceTrigger* qui va nous permettre de spécifier quel est l'événement qui va déclencher la mise à jour des données vers la source.

Toutes ces propriétés seront bien sûr détaillées tout au long de ce chapitre.

2.2 Les différents modes de Data Binding

Comme on a pu le voir précédemment dans notre étude des bases du Data Binding, on peut donner plusieurs modes à notre Binding. Concrètement en quoi cela consiste, pourquoi proposer des modes de Binding différents ?

En pratique, on se rend très vite compte qu'on aimerait configurer notre Binding de façon, par exemple, à mettre à jour notre contrôle ET notre source de donnée dés qu'un des deux est modifié, ou seulement un seul, etc.

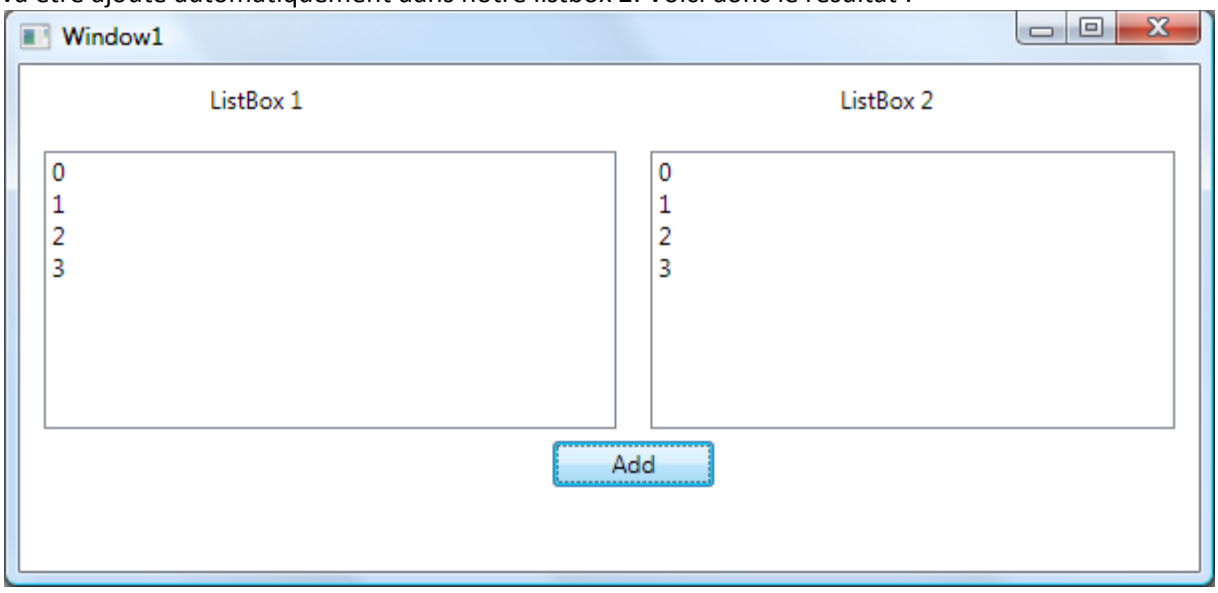
C'est à cela que va nous servir le mode qu'on va donner à notre Binding. En effet, nous verrons que nous avons la possibilité de contrôler tout ça grâce à la seule propriété *Mode*.

Nous allons commencer par voir la méthode dîtes *OneWay* qui consiste en fait tout simplement à mettre à jour automatiquement les données **de la source vers la cible**.

Voici comment procéder :

```
<!--XAML-->
<Window x:Class="DFWpfApplication.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="289" Width="607">
    <Grid>
        <ListBox Margin="12,43,292,71" Name="listBox1"
          SelectionMode="Multiple" />
        <ListBox HorizontalAlignment="Right" Margin="0,43,12,71"
          Name="listBox2" Width="263" ItemsSource="{Binding
          ElementName=listBox1, Path=Items, Mode=OneWay}"
          SelectionMode="Multiple" />

        <TextBlock Height="28" HorizontalAlignment="Left"
          Margin="95,9,0,0" Name="label1" VerticalAlignment="Top"
          Width="120">ListBox 1</TextBlock>
        <TextBlock Height="28" HorizontalAlignment="Right"
          Margin="0,9,60,0" Name="label2" VerticalAlignment="Top"
          Width="120">ListBox 2</TextBlock>

        <Button Height="23" Margin="267,0,243,42" Name="add"
          VerticalAlignment="Bottom" Click="add_Click">Add</Button>
    </Grid>
</Window>
```

Comme vous pouvez le voir, ce code n'est vraiment pas complexe. On a créé deux ListBox qu'on a lié ensemble avec un Binding OneWay, ce qui implique que ce qu'on ajoute dans la listbox 1 va être ajouté automatiquement dans notre listbox 2. Voici donc le résultat :



Il existe d'autres modes de Binding comme le Binding OneTime qui est similaire au Binding OneWay, c'est-à-dire qu'il va mettre à jour les données de la source vers la cible toujours mais uniquement lors de l'initialisation des contrôles, les modifications faites après ne seront pas effective.

On a également l'inverse du OneWay qui est le OneWayToSource qui consiste donc tout simplement à mettre à jour les données de la cible vers la source.

On peut citer aussi le Binding TwoWay qui quant à lui consiste à faire deux OneWay c'est-à-dire mettre à jour de la source de donnée vers la cible et de la cible vers la source.

3 Gestion du Data Binding

3.1 Gestion du comportement

Nous avons vu dans le chapitre précédent comme utiliser les Triggers. Cela nous a permit de voir comment modifier le style d'un contrôle en fonction de propriétés. Mais nous ne pouvions pas surveiller les propriétés de nos propres classes. Avec les Data Triggers, nous allons enfin pouvoir le faire.

Pour comprendre comment fonctionne les data triggers nous allons voir un exemple. Dans cet exemple, nous allons créer une collection « Garage » remplit d'objets de type « Voiture ». Ensuite, nous allons remplir une ListBox avec cette collection, et modifier la couleur et les états des items selon la valeur de leur propriété « Vitesse ».

Voici tout d'abord notre collection et notre classe Voiture :

```
// C#
class Voiture
{
    private string _Vitesse;
    private string _Modele;

    public Voiture(string vitesse, string modele)
    {
        _Vitesse = vitesse;
        _Modele = modele;
    }

    public string Vitesse
    {
        get { return _Vitesse; }
    }

    public string Modele
    {
        get { return _Modele; }
    }
}
```

Rien de compliqué dans ce code, lorsque nous créons un nouvel objet de type Voiture, nous lui donnons une vitesse et le nom de son modèle. Puis nous créons deux propriétés en lecture seule afin de récupérer ces valeurs.

```
// C#
class Garage : ObservableCollection<Voiture>
{
    public Garage()
    {
        Add(new Voiture("Rapide", "Ferrari F430"));
        Add(new Voiture("Lente", "Peugeot 106"));
        Add(new Voiture("Rapide", "Subaru Impreza"));
        Add(new Voiture("Rapide", "Nissan Skyline"));
        Add(new Voiture("Lente", "Renault Clio"));
    }
}
```

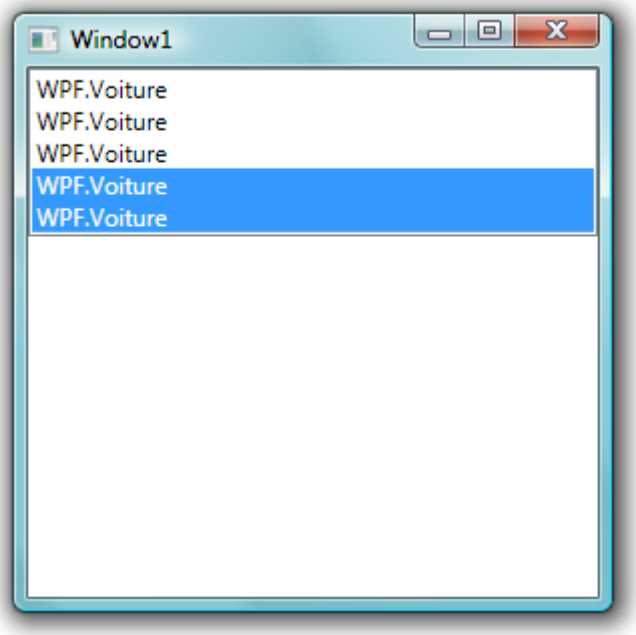
Ici non plus, rien de bien compliqué, nous créons une collection qui hérite de ObservableCollection<>. ObservableCollection est une collection dynamique qui notifie toute modification, par exemple lorsque des éléments sont ajoutés, supprimés, ou que la collection entière est actualisée. Elle parait donc tout à ait approprié pour l'utilisation du Binding. Pour en savoir plus, rendez vous sur MSDN.

Nous avons maintenant nos deux classes. Nous allons maintenant instancier une nouvelle collection dans notre fichier XAML afin de pouvoir l'utiliser :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      xmlns:local="clr-namespace:WPF"
      Width="300" Height="300">
    <Window.Resources>
        <local:Garage x:Key="Garage" />
    </Window.Resources>
    <StackPanel>
        <ListBox x:Name="MaListe" ItemsSource="{StaticResource Garage}"
          SelectionMode="Multiple" />
    </StackPanel>
</Window>
```

Nous ajoutons donc la classe Garage en tant que ressource, sans avoir oublié d'inclure le namespace du projet dans notre balise Window (ici le namespace du projet est WPF). Ensuite nous pouvons remplir une ListBox avec les valeurs de notre collection Garage en passant la ressource de manière statique à notre ListBox grâce à son attribut ItemsSource.

A ce stade, voici ce que nous avons :



Une ListBox multi-sélections remplit d'objets de type Voiture. Ce n'est pas encore ce que nous souhaitons. A partir de là, deux méthodes vont nous permettre d'afficher ce que l'on veut dans notre ListBox, soit en code behind en redéfinissant la méthode ToString dans notre classe Voiture. Soit en utilisant un setter. Dans un premier temps nous allons utiliser le setter, par la suite, nous utiliserons la méthode ToString.

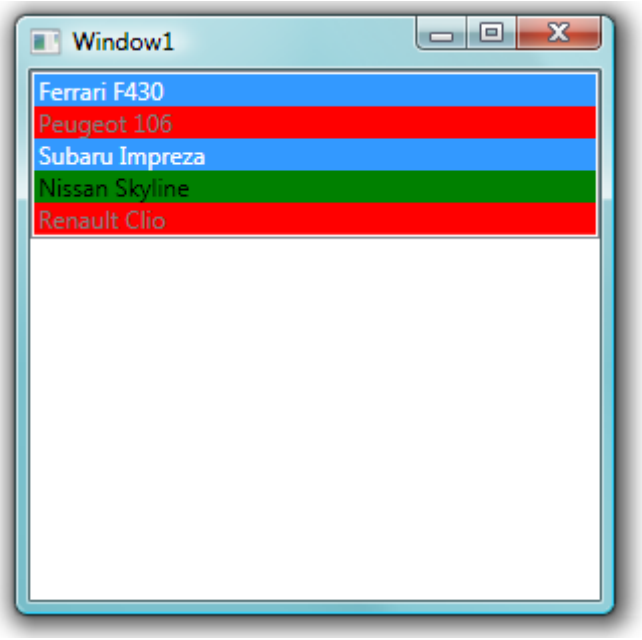
Voici le style utilisé pour cela :

```
<!--XAML-->
<Window.Resources>
    <local:Garage x:Key="Garage" />
    <Style TargetType="ListBoxItem">
        <Style.Triggers>
            <DataTrigger Binding="{Binding Path=Vitesse}" Value="Rapide">
                <Setter Property="IsSelected" Value="True" />
                <Setter Property="Background" Value="Green" />
            </DataTrigger>
            <DataTrigger Binding="{Binding Path=Vitesse}" Value="Lente">
                <Setter Property="IsEnabled" Value="False" />
                <Setter Property="Background" Value="Red" />
            </DataTrigger>
        </Style.Triggers>
        <Setter Property="Content" Value="{Binding Path=Modele}" />
    </Style>
</Window.Resources>
```

Les DataTrigger sont, comme vous pouvez le voir, assez proche des property trigger. La plus grosse différence (au delà du nom des balises) est que nous allons binder le trigger à une propriété, ici, la propriété Vitesse.

Dans le premier cas, si la vitesse a pour valeur « Rapide », alors nous allons sélectionner les cases et leur donner un fond vert, si la propriété a pour valeur « Lente » alors les cases seront rouges et désactivée. Enfin nous définissons un Setter afin de binder pour chaque item la valeur de la propriété Modele de notre classe Voiture, à la propriété Content de notre ListBoxItem.

Voici le nouveau résultat :



Nos véhicule rapide sont effectivement de couleur verte et sélectionné (mit à part Nissan Skyline qui a été désélectionné afin de voir le fond vert), et les véhicule lent sont désactivé et de couleur rouge.

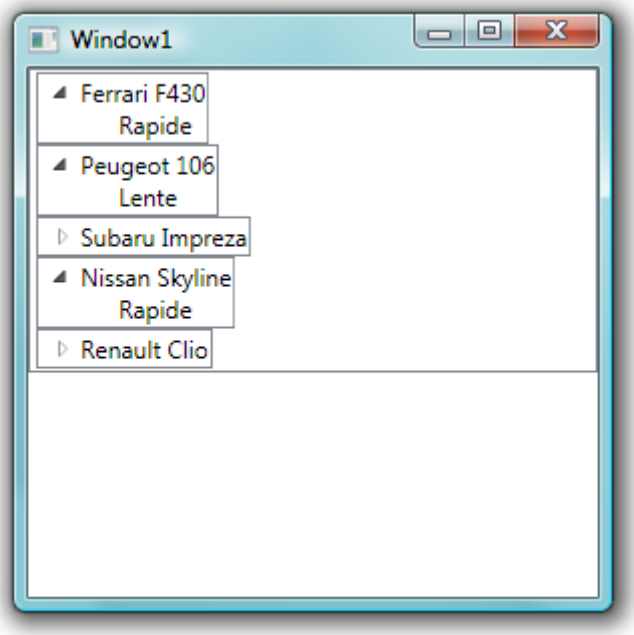
3.2 Gestion du Template

Nous venons de voir comment utiliser les Data Triggers, nous allons maintenant voir comment gérer des Data Template. Les Data Template vont nous permettre de créer des minis Template pour mettre en forme les données. Par exemple, si nous reprenons l'exemple précédent, nous aimerions afficher dans chaque ListBoxItem un TreeView contenant le modèle de la voiture et qui déroule sa vitesse. Voyons comment faire avec un exemple :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      xmlns:local="clr-namespace:WPF"
      Width="300" Height="300">
    <Window.Resources>
        <local:Garage x:Key="Garage" />
    </Window.Resources>
    <StackPanel>
        <ListBox x:Name="MaListe" ItemsSource="{StaticResource Garage}"
         SelectionMode="Multiple" >
           <ListBox.ItemTemplate>
               <DataTemplate>
                   <StackPanel Orientation="Horizontal">
                       <TreeView>
                           <TreeViewItem Header="{Binding Path=Modele}">
                               <TreeViewItem Header="{Binding
                                Path=Vitesse}" />
                           </TreeViewItem>
                       </TreeView>
                   </StackPanel>
               </DataTemplate>
           </ListBox.ItemTemplate>
        </ListBox>
    </StackPanel>
</Window>
```

Pour créer un DataTemplate, nous avons utilisé les balises DataTemplate. Celles-ci sont englobées dans les balises <ListBox.ItemTemplate> qui permettent d'indiquer que nous retouchons le Template de chaque Item.

Ensuite, il suffit de suivre le même procédé que pour les Template classique en gardant en tête que <DataTemplate> est la racine du Template. L'exemple précédent donne comme résultat :



Nous arrivons au résultat attendu de manière très simple. Les Data Template offrent les mêmes possibilités que les Template, laissez courir votre imagination.

Note : Les DataTemplate proposent également de créer des Template avec hiérarchie, les HierarchicalDataTemplate. Ceux-ci s'adaptent parfaitement aux TreeViewsItem et MenuItem lorsque que la structure de vos données devient complexe (notamment si vous avez de multiples imbrications d'item). Leur utilisation étant quasi similaire aux DataTemplate, nous vous invitons à vous documenter sur les HierarchicalDataTemplate sur MSDN.

3.3 Lier un objet avec ObjectDataProvider

Nous avons vu jusqu'à présent, notamment avec les Data Trigger et Data Template comment faire de la liaison de donnée simple. Nous allons maintenant aborder une manière de lier les données un peu plus complexe, mais qui offre beaucoup plus de possibilité.

En effet, jusqu'à présent, nous devions obligatoirement utiliser du code behind pour passer un paramètre au constructeur de notre classe, et pour appeler une méthode. Grâce à l'objet ObjectDataProvider, nous allons pouvoir effectuer les actions précédentes, ainsi que d'autres actions telles que le remplacement de la source de donnée (passer d'un garage par exemple à une écurie est possible avec quelques adaptations) et également de travailler de manière asynchrone.

Nous allons étudier un exemple d'utilisation d'ObjectDataProvider simple, qui devrait vous mettre sur la voie pour des utilisations plus complexes.

Dans notre exemple, nous allons rajouter à notre exemple de Garage une méthode permettant de retourner une Voiture en fonction de sa place dans la collection. Pour cela, nous commençons par rajouter une méthode dans la classe Garage.

```
// C#
class Garage : ObservableCollection<Voiture>
{
    public Garage()
    {
        Add(new Voiture("Rapide", "Ferrari F430"));
        Add(new Voiture("Lente", "Peugeot 106"));
        Add(new Voiture("Rapide", "Subaru Impreza"));
        Add(new Voiture("Rapide", "Nissan Skyline"));
        Add(new Voiture("Lente", "Renault Clio"));
    }

    public string ChoixVoiture(string numero)
    {
        int result;
        int.TryParse(numero, out result);
        return this.ElementAt(result).ToString();
    }
}
```

Notre méthode retourne un objet Voiture sous forme de chaine de caractère grâce à ToString. Nous avons convertit le paramètre en Int pour une raison pratique, cependant, ObjectDataProvider offre une fonctionnalité permettant de convertir à la volée les paramètres, nous le verrons dans un deuxième temps.

Ensuite nous modifions notre XAML de la manière suivante :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      xmlns:local="clr-namespace:WPF"
      xmlns:system="clr-namespace:System;assembly=mscorlib"
      Width="300" Height="300">
    <Window.Resources>
        <ObjectDataProvider x:Key="Garage" ObjectType="{x:Type
          local:Garage}" />
        <ObjectDataProvider ObjectInstance="{StaticResource Garage}"
          x:Key="ChoixVoiture" MethodName="ChoixVoiture">
            <ObjectDataProvider.MethodParameters>
                <system:String>2</system:String>
            </ObjectDataProvider.MethodParameters>
        </ObjectDataProvider>
    </Window.Resources>
    <StackPanel>
        <ListBox x:Name="MaListe" ItemsSource="{Binding
          Source={StaticResource Garage}}" SelectionMode="Multiple" />
        <TextBox>
            <TextBox.Text>
                <Binding Source="{StaticResource ChoixVoiture}"
                 Path="MethodParameters[0]" BindsDirectlyToSource="true"
                 UpdateSourceTrigger="PropertyChanged" />
            </TextBox.Text>
        </TextBox>
        <Label Content="{Binding Source={StaticResource ChoixVoiture}}" />
    </StackPanel>
</Window>
```

Tout d'abord, nous avons inclus un nouveau NameSpace qui va nous permettre d'accéder aux objets de bases du Framework, afin de pouvoir utiliser plus tard le type String.

Dans les ressources de la fenêtre, nous avons deux ObjectDataProvider.

- Le premier va se charger de créer une instance de « Garage »
- Le second va nous permettre de pointer sur une méthode de l'instance, à savoir la méthode ChoixVoiture. Pour indiquer que nous allons utiliser l'instance précédemment crée, nous utilisons l'attribut ObjectInstance.

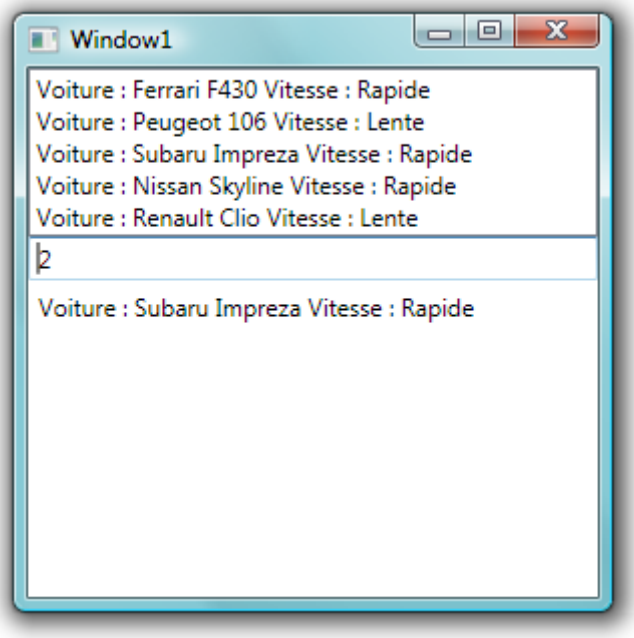
Les balises <ObjectDataProvider.MethodParameters> vont nous permettre de définir quel type de donnée nous allons passer par défaut à la méthode, ici, nous passons le caractère 2 sous forme de string.

Ensuite, dans le corps de notre fenêtre, nous avons notre ListBox précédente, qui va afficher strictement la même chose que dans les précédents exemples suivit par une textbox et un Label.

La Textbox va binder sur la méthode ChoixVoiture en Two-Ways, en effet nous allons envoyer et recevoir des informations. Rassurez vous, par défaut une TextBox à un comportement Two-Ways. Ensuite elle va cibler le premier paramètre de celle-ci (paramètre indice 0). A chaque modification, elle avertira que sa valeur a changé car nous avons définit UpdateSourceTrigger à PropertyChanged. Enfin BindsDirectlyToSource à true nous permet d'indiquer que la TextBox doit se lier à la propriété MethodParameters de notre ObjectDataProvider, et non pas avec les propriétés de l'objet encapsulé (Garage).

Enfin, Le Label se contente de récupérer la valeur émise par la méthode ChoixVoiture.

Si vous avez bien suivit les instructions, vous devriez obtenir cette fenêtre :



Si vous changez la valeur dans la textbox, le modèle de voiture changera en conséquence.

ObjectDataProvider va également nous permettre d'ajouter des vérifications et des Converters à nos bindings. Nous avons vu ici que nous devions convertir un string en Int pour que la méthode ChoixVoiture fonctionne, de même si nous entrons une lettre à la place d'un chiffre, nous ne pourrons retirer aucun résultat. Nous allons donc tâcher de résoudre ces deux problèmes :

Tout d'abord, nous allons créer notre Converter. Celui-ci va se charger de convertir un Int en String et vice versa.

Voici le code behind du Converter :

```
// C#
using System.Windows.Data;
public class IntToString : IValueConverter
{
    public object Convert(object value, Type targetType, object
parameter, System.Globalization.CultureInfo culture)
    {
        if (value != null)
        {
            return value.ToString();
        }
        else
            return null;
    }

    public object ConvertBack(object value, Type targetType, object
parameter, System.Globalization.CultureInfo culture)
    {
        string chaine = value as string;
        if (chaine != null)
        {
            int resultat;
            if (int.TryParse(chaine, out resultat))
            {
                return resultat;
            }
        }
        return null;
    }
}
```

Comme vous pouvez le constater, c'est une classe dérivant de System.Windows.Data.IValueConverter, contenant deux méthodes obligatoires. Dans la première nous faisons la conversion principale : Int en String, et dans la seconde méthode, nous faisons l'inverse String vers Int.

Nous pouvons maintenant utiliser notre Converter, tout d'abord nous modifions la méthode ChoixVoiture :

```
// C#
class Garage : ObservableCollection<Voiture>
{
    public string ChoixVoiture(int numero)
    {
        return this.ElementAt(numero).ToString();
    }
}
```

Ensuite dans le code XAML :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      xmlns:local="clr-namespace:WPF"
      xmlns:system="clr-namespace:System;assembly=mscorlib"
      Width="300" Height="300">
    <Window.Resources>
        […]
            <ObjectDataProvider.MethodParameters>
                <system:Int32>2</system:Int32>
            </ObjectDataProvider.MethodParameters>
        […]
        <local:IntToString x:Key="IntToString" />
    </Window.Resources>
    <StackPanel>
        […]
        <TextBox>
            <TextBox.Text>
                <Binding Source="{StaticResource ChoixVoiture}"
                 Path="MethodParameters[0]" BindsDirectlyToSource="true"
                 UpdateSourceTrigger="PropertyChanged"
                 Converter="{StaticResource IntToString}">
                </Binding>
            </TextBox.Text>
```

Nous ajoutons dans les ressources une instance de la classe IntToString, associé à la clé IntToString. Ensuite nous changeons le type du paramètre passé à la méthode (de string à Int32). Puis nous "bindons" la ressource à l'attribut Converter du Binding de la méthode.

Maintenant, l'application gère elle-même les conversions entre la textbox et le code behind.

Nous pouvons enfin appliquer notre validateur, pour cela nous allons d'abord en écrire le code behind :

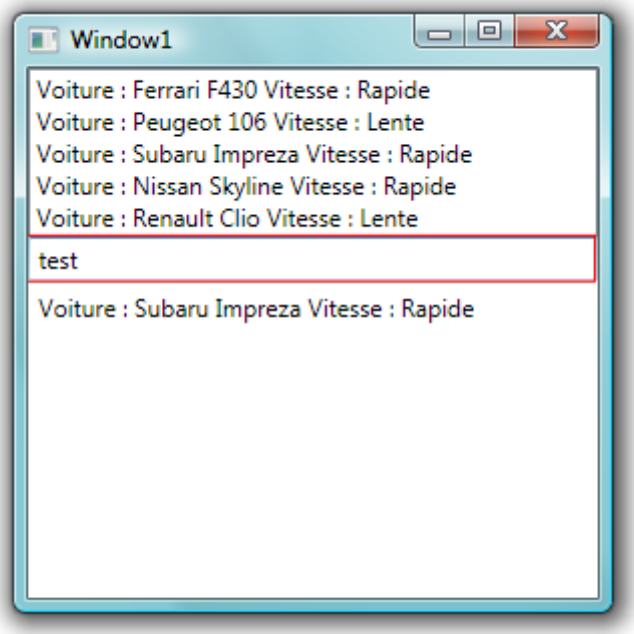
```
// C#
using System.Windows.Controls;
public class ChoixValidationRule : ValidationRule
{
    public override ValidationResult Validate(object value,
System.Globalization.CultureInfo cultureInfo)
    {
        int resultat;
        if (!int.TryParse(value as string, out resultat))
        {
            return new ValidationResult(false, "Erreur, conversion en Int
            impossible");
        }

        if ((resultat < 0) || (resultat > 4))
        {
            return new ValidationResult(false, "Erreur, ce chiffre n'est
            pas compris entre 0 et 4");
        }
        return ValidationResult.ValidResult;
    }
}
```

Nous créons une classe qui dérive de ValidationRule, celle-ci comporte une méthode surchargée retournant un ValidationResult : Validate.

A l'intérieur nous faisons nos tests, nous vérifions d'abord que nous pouvons convertir en Int la valeur inscrite, puis si elle est comprise entre 0 et 4. Sinon, nous retournons un ValidationResult à false, suivit du message d'erreur (dans le cas ou vous voudriez l'afficher par exemple).

Si tout va bien, alors nous retournons la valeur ValidationResult.ValidResult.

Si nous entrons maintenant tout caractère interdit, la textbox va voir sa bordure devenir rouge, essayons :



4 Conclusion

Comme on a pu le voir, le DataBinding est un mécanisme permettant de lier un ou plusieurs contrôles à une ou plusieurs sources de données. Plus précisément, on lie une source de données multiples : contrôles (Listbox, textbox…), objets, XML, base de données… à une propriété d'un contrôle.

On a également vu qu'il existe différent modes de binding permettant de spécifier dans quel direction le binding va s'effectuer et à quel moment : OneWay, OneTime, TwoWay, OneWayToSource…

Mais aussi qu'il était également possible de spécifier comment la source de donnée doit être mise à jour lorsque les données de la cible changent : UpdateSourceTrigger.

Et pour finir qu'il était possible de mettre en forme les données binder avec l'utilisation des DataTemplate.

Le Databinding et toutes ses fonctionnalités permettent de lier ce que l'on veut, de la manière que l'on veut et suivant la forme que l'on souhaite. Cela permet de gagner du temps, des lignes de codes et par conséquent, d'augmenter votre productivité.