WPF

Support de cours

Auteur : Yannick BAZAN
Version : 1.0 (Avril 2019)

TABLE DES MATIERES

PRESENTATION DE WPF

Sorti en 2006, avec la version 3.0 du framework .NET, **W**indows **P**resentation **F**oundation se veut le successeur de Windows Forms.

Les avancées par rapport à Windows Forms sont nombreuses :

- Possibilité de séparer le « code » de l'interface du code de l'application (via le langage créé pour l'occasion, XAML)
- Rendu de l'interface en vectoriel (rendu cohérent et indépendant de la résolution de l'affichage)
- Utilisation de DirectX (c'est la carte graphique qui travaille au rendu, par le processeur)
- Manipulation unifiée de contenus auparavant difficiles à intégrer : 3D, multimédia, documents riches
- Possibilité de modifier complétement le look&feel de l'interface

Le framework évoluera fréquemment jusqu'à la version 4.5 du framework .NET. Bien qu'aujourd'hui WPF ne soit plus forcément mis en avant, il est toujours maintenu. Et l'ouverture de son code source et son intégration avec .NET Core 3.0 (sortie en 2019) pourrait lui donner un nouvel élan. Wait and see…

Reste que l'apprentissage de cette technologie n'est pas aisé, vu tous les « nouveaux » concepts mis en œuvre, que le développeur doit au minimum comprendre pour en tirer toute sa puissance.

Au cours de cette formation, les concepts suivants seront abordés :

- Les Routed Events
- Le DataBinding
- Les Templates
- Les styles
- Les Behaviors
- Les Dependency Properties

PRESENTATION DU LANGAGE XAML

WPF ne va pas sans ce nouveau langage, XAML pour e**X**tensible **A**pplication **M**arkup **L**anguage (prononcé « zammeul »).

C'est le langage pour décrire les interfaces graphiques : les fenêtres et leurs contenus.

Derrière ce nom « barbare », il faut voir XAML comme un langage XML. En effet, un fichier XAML (avec pour extension .xaml), n'est ni plus ni moins qu'un fichier XML.

Hello World !

Par exemple dans le code suivant (une fenêtre avec le texte Hello World ! à l'intérieur), on retrouve les caractéristiques du langage XML.

```
<Window x:Class="M01_HelloWorld.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Première application WPF"
        Height="300"
        Width="300">
    <Grid>
        <TextBlock Text="Hello World!"/>
    </Grid>
</Window>
```

En rouge : les balises (avec une balise ouvrante et fermante, ou une balise auto-fermante)
En vert : les attributs (clé – valeur)
En souligné bleu : les espaces de nom (qui permettent de restreindre les noms de balises/attributs à un référentiel défini).

Visual Studio offre un designer qui permet de visualiser en temps réel le rendu du code XAML. Il offre aussi de l'Intellisense, une assistance lors de la frappe avec les balises/attributs valides.

Aller plus loin

Que se passe-t-il à la compilation ?

Il faut savoir que tout ce qu'on écrit en XAML peut être écrit en code application (C# ou VB.Net). En effet, les balises utilisées correspondent à des classes et les attributs à des propriétés de ces classes.

On pourrait penser qu'à la compilation, Visual Studio transforme ce code XAML en code C# par exemple. Mais il n'en est rien !

En fait, le code XAML est stocké en tant que ressource embarquée dans l'exécutable. Pour des soucis de taille de fichier et de performance de chargement, ce code est sérialisé (on parle alors de BAML).
Quand l'application charge une fenêtre par exemple, elle récupère le code BAML correspondant à cette fenêtre, dans les ressources, le désérialise puis l'exécute à la volée.

D'où l'importance de veiller à la performance de son code, surtout pour les composants complexes !

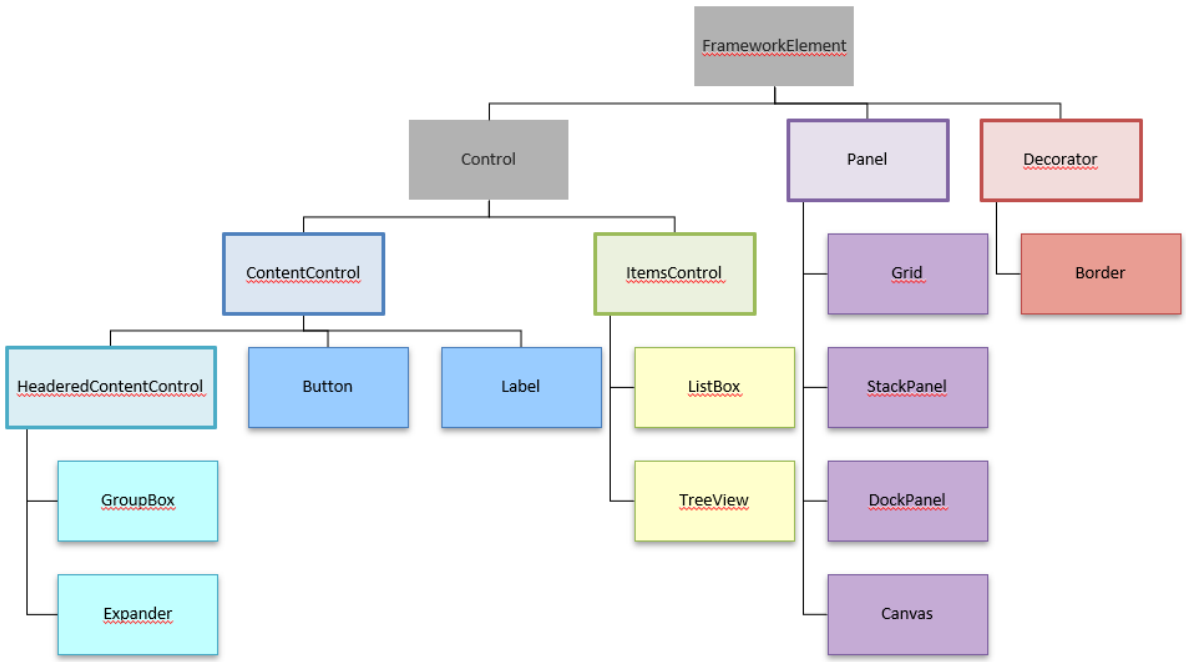
REALISATION D'INTERFACES

Contrôles

La première étape pour réaliser des interfaces, est de choisir les composants qu'on veut avoir pour chaque « écran ». Et le framework WPF en fournit par défaut des dizaines. On les appelle contrôles (zones de texte, listes déroulantes, boutons, menus…).

WPF fournit aussi des conteneurs. Ce sont des contrôles qui permettent de définir le « layout », c'est-à-dire la mise en page des écrans.

Pour offrir une vision plus haut niveau, on obtient le schéma suivant :



Les contrôles « conteneurs » héritent de la classe **Panel**.
Parmi les conteneurs les plus utilisés, on retrouve :

- **Canvas :** les contrôles sont positionnés depuis le point origine du conteneur (haut-gauche)
- **StackPanel :** les contrôles sont affichés à la suite (verticalement ou horizontalement)
- **DockPanel :** les contrôles sont positionnés sur les côtés (haut, bas, gauche, droite, milieu)
- **Grid :** l'écran est « découpé » en lignes et en colonnes, et les contrôles sont positionnés sur cette grille.

Les contrôles héritant de la classe **ContentControl** ont une propriété **Content**. C'est la propriété par défaut, ce qui veut dire qu'en XAML, les 3 codes suivants sont équivalents :

```
<Button Content="Valider"/>

<Button>
   <Button.Content>Valider</Button.Content>
</Button>

<Button>Valider</Button>
```

Les contrôles héritant de la classe **ItemsControl** correspondent à des contrôles de liste. Ils ont une propriété **Items** qui contiennent la liste des éléments enfants. En XAML, il suffit de mettre les éléments à la suite à l'intérieur de la balise.

Les codes suivants sont équivalents :

```
<ListBox>
  <ListItem Content="Enfant 1"/>
  <ListItem Content="Enfant 2"/>
  <ListItem Content="Enfant 3"/>
</ListBox>

<ListBox>
  <ListBox.Items>
    <ListItem Content="Enfant 1"/>
    <ListItem Content="Enfant 2"/>
    <ListItem Content="Enfant 3"/>
  </ListBox.Items>
</ListBox>
```

Les contrôles héritant de la classe HeaderedContentControl proposent les propriétés :

- **Content** (puisque la classe HeaderedContentControl hérite de la classe ContentControl)
- **Header :** Contenu de l'en-tête

Positionnement des contrôles

En plus du positionnement défini par le type de conteneur, il est possible de maîtriser l'espacement externe (avec les contrôles voisins) et interne (avec les contrôles enfants) des contrôles.

Pour l'espacement externe, on utilise la propriété **Margin**. Pour l'espacement interne, il s'agit de la propriété **Padding**.

Ces propriétés acceptent plusieurs valeurs :

- Un entier : l'espace sera appliqué aux 4 côtés

- 2 entiers (ex : 8,5) : l'espacement sera de 8 pour la gauche/droite et 5 pour le haut/bas
- 4 entiers (ex : 8,5,4,2) : l'espacement sera appliqué respectivement à gauche, haut, droite et bas

Aller plus loin

Les valeurs pour le Padding et le Margin sont-elles exprimées en pixel ?

Ce serait tentant de le penser. Et plus d'un développeur ferait l'erreur.

Les valeurs sont en fait des DIP (**D**evice **I**ndependant **P**ixel). Le calcul est le suivant : 1 pouce = 2,54 cm = 96 dip

Pourquoi cette mesure ?
Comme vu dans l'introduction, WPF a été pensé pour offrir un affichage dit vectoriel, qui reste le même quelle que soit la résolution.

Et pour s'affranchir des résolutions différentes (pensez aussi imprimantes), il fallait une unité indépendante !

Il est aussi possible de définir un alignement par rapport au parent. Ce peut être verticalement (propriété **VerticalAlignment**) et horizontalement (propriété **HorizontalAlignment**).

Parmi les différentes valeurs, celle par défaut est Stretch, ce qui signifie que le contrôle prend toute la place qu'on lui offre 😊.

Affichage (ou pas) d'un contrôle

Le tour d'horizon ne serait pas complet sans parler de la propriété Visibility. Comme son nom l'indique, elle permet d'indiquer si un contrôle doit être affiché ou pas. Mais ce n'est pas un booléen ! C'est une énumération avec les valeurs suivantes :

- **Visible**
- **Hidden :** le contrôle est juste masqué et la place qu'il prend est conservée
- **Collapsed :** le contrôle n'apparaît pas et la place qu'il prend est récupérée

CONCEPT 1 : ROUTED EVENTS

Les événements en C# correspondent à une implémentation du design pattern Observer. L'idée est de pouvoir notifier des classes de changements, sans qu'il y ait de dépendance directe entre la classe qui notifie (dit observable) et celle(s) qui reçoivent les notifications (dite(s) observateur(s)).

Les contrôles WPF définissent des événements (Click, MouseEnter…). Cela permet d'écrire du code qui sera exécuté quand l'événement en question sera déclenché.

Mais les événements des contrôles sont plus élaborés que les événements « classiques » du framework. Ils sont dits « routés » parce que l'événement ne se cantonne pas à son contrôle. Il parcoure l'arbre visuel jusqu'à ce qu'un code associé à cet événement marque l'événement comme traité (propriété Handled) ou qu'il n'y a plus d'élément dans l'arbre.

Aller plus loin

Qu'est-ce que l'arbre visuel ?

On comprend bien que la description des interfaces est hiérarchique. Une fenêtre va contenir un conteneur de type StackPanel. Ce dernier contient 3 autres contrôles qui à leur tour, contiennent… Et ce à l'infini…

Cette hiérarchie est appelée arbre logique.

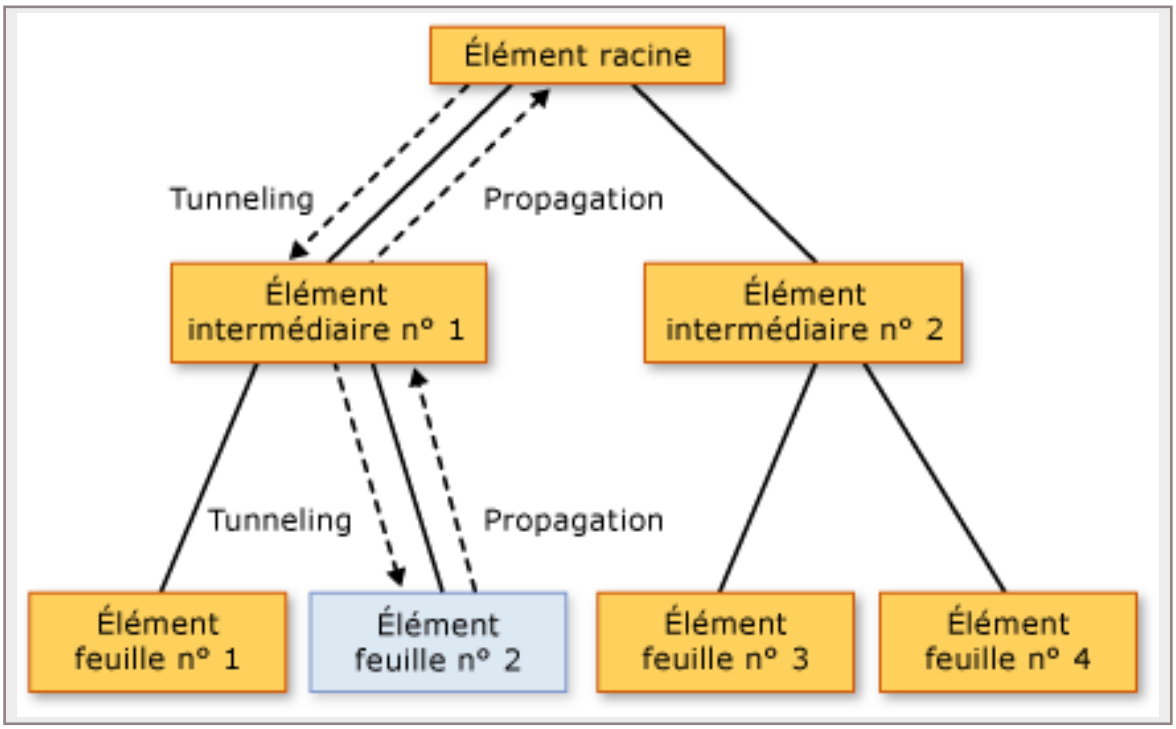
Mais en réalité, quand le moteur WPF affiche notre fenêtre, il y a beaucoup plus d'éléments dans l'arbre. Les contrôles WPF sont en eux-mêmes une composition d'autres éléments visuels (des contrôles conteneurs, ou même des éléments « bas-niveau » comme des Border, Rectangle…).
L'arbre visuel correspond à l'arbre lors de l'exécution.

Il est par ailleurs possible de naviguer dans l'arbre visuel lors de l'exécution. Mais ceci est une autre histoire…

Il y a 2 types d'événements routés :

- **Bubbling** (bulles) : l'événement part du contrôle qui l'a déclenché et « monte » l'arbre visuel jusqu'à la racine (ou si un code l'a marqué comme traité avec la propriété Handled)
- **Tunneling** (tunnels) : l'événement part du contrôle racine et « descend » l'arbre visuel jusqu'au contrôle qui l'a déclenché (sauf si un code avant l'a marqué comme traité). Par convention, ces événements sont préfixés Preview

L'image suivante illustre ces 2 stratégies :



Source : Microsoft

Pourquoi cette distinction ?
Les événements tunneling permettent de « d'empêcher » l'exécution de l'événement bubbling.

Par exemple, on imagine une application avec plusieurs documents ouverts, dont un est en cours de modification. Puis l'utilisateur clique sur la croix en haut à droite pour fermer l'application.
Comment prévenir l'utilisateur qu'il y a des modifications non sauvegardées ?
Comment empêcher la fermeture si une opération importante est en cours ?

Ressources annexes

- Documentation Microsoft : https://docs.microsoft.com/fr-fr/dotnet/framework/wpf/advanced/events-wpf

CONCEPT 2 : DATA BINDING

On touche ici à un des concepts les plus puissants et géniaux de WPF : le Data Binding (ou liaison de données).

Au lieu de mettre une valeur dans la propriété d'un contrôle et de mettre à jour explicitement cette valeur à chaque changement, on « attache » (ou « bind ») la propriété du contrôle à la propriété de l'objet. A chaque modification, le contrôle est mis à jour sans qu'on ait à s'en préoccuper.

Ce mécanisme permet une isolation entre le contrôle et l'objet auquel il s'attache.

Contexte de données (Data context)

Presque tous les contrôles (fenêtres y compris), possèdent la propriété DataContext. Cette propriété permet de spécifier un objet (quelconque) qui va servir de donnée attachée au contrôle.

Le contexte de données s'applique à tous les contrôles enfants jusqu'à ce qu'un contrôle définisse son propre contexte de données.

Généralement, on renseigne la propriété DataContext du contrôle le plus haut dans l'arbre logique.

Liaison de données (Data binding)

Concrètement, la liaison de données se fait par la classe Binding. Mais dans le XAML, cette classe est souvent « masquée » parce qu'il y a une syntaxe qui permet de simplifier la déclaration de la liaison.

Imaginons une classe MainWindow, avec une propriété Nom. Je définis dans l'interface (en XAML) un champ texte (TextBox) et une zone de texte (TextBlock).

Comment « binder » mes 2 contrôles à la propriété Nom ?

```
<Window x:Class="MonProjet.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:MonProjet"
        …>
    <Window.DataContext>
        <local:MainWindow/>
    </Window.DataContext>
    <StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Champ texte"/>
            <TextBox Text="{Binding Path=Nom}"/>
```

```
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Zone de texte"/>
            <TextBlock Text="{Binding Path=Nom}"/>
        </StackPanel>
    </StackPanel>
</Window>
```

Déjà, on note comment renseigner le contexte de données de la fenêtre (en rouge). Quand on veut utiliser des classes qui ne sont pas du framework (gérées dans l'espace de nom par défaut), il faut « importer » l'espace de nom. Ici, je l'ai importé dans la « variable » local.

Ensuite, au niveau des contrôles, sur la propriété Text, je déclare la liaison avec la valeur {Binding Path=…}.

Notez que le code ci-dessous est identique :

```
<TextBox>
    <TextBox.Text>
        <Binding Path="Nom"/>
    </TextBox.Text>
</TextBox>
```

On voit de suite l'avantage d'utiliser la déclaration simplifiée. Cela dit, il est bien de connaître la syntaxe officielle, car il y aura des cas où ce sera la seule manière possible d'implémenter un binding complexe.

Modes de liaison de données

En plus d'offrir un moyen de lier les données, WPF permet d'être plus fin dans la définition.

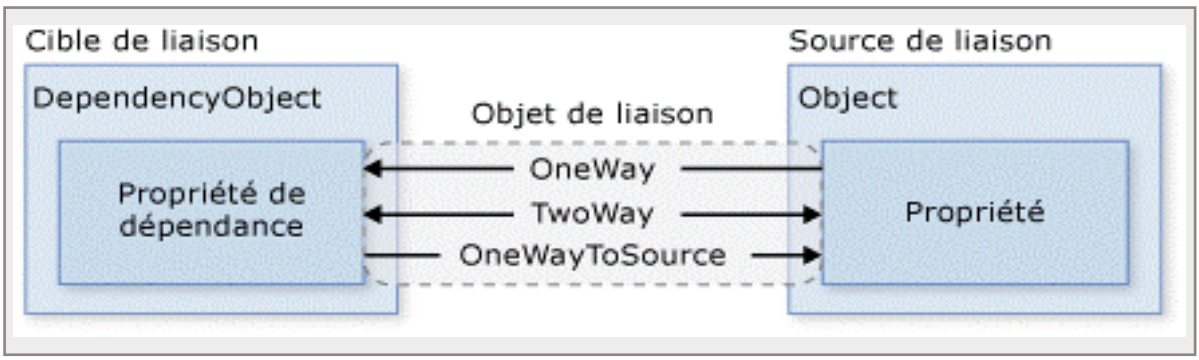
Il y a 4 modes de liaisons (propriété **Mode** de la classe Binding) :

- **OneWay :** seul le contrôle est mis à jour quand l'objet attaché est modifié
- **TwoWay :** la mise à jour se fait dans les 2 sens

- **OneTime :** le contrôle prend la valeur de l'objet attaché au chargement (puis ne bouge plus)
- **OneWayToSource :** l'objet attaché est mis à jour lors du chargement du contrôle (et plus après)

Il faut garder à l'esprit qu'une liaison TwoWay est plus coûteuse en termes de ressources qu'une liaison OneWay, qui elle est plus coûteuse qu'une liaison OneTime.

Voici le schéma de la documentation Microsoft :



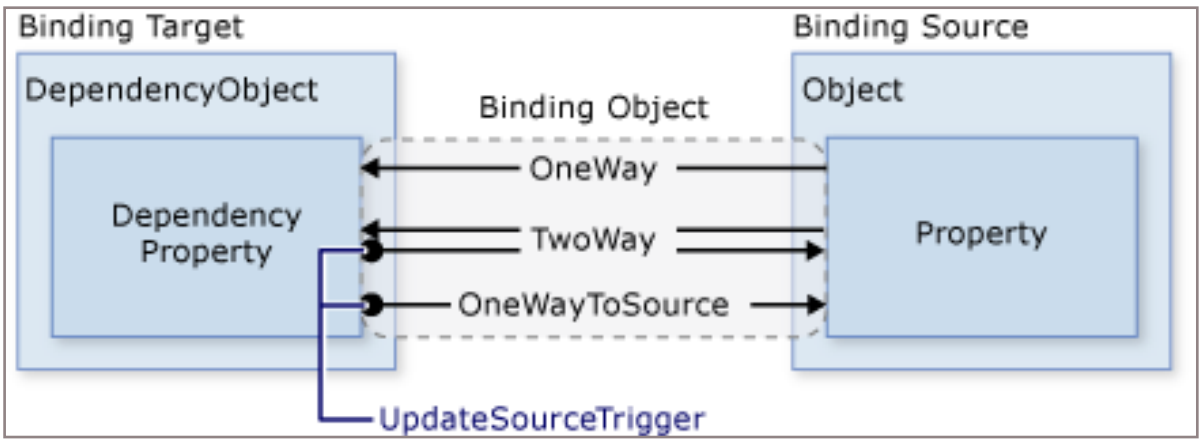
Déclenchement de la mise à jour

Et oui, il est aussi possible de paramétrer à quel moment la mise à jour de l'objet attaché doit se faire.

Il s'agit de la propriété **UpdateSourceTrigger** de la classe Binding. Les valeurs possibles sont :

- **PropertyChanged:** dès qu'une modification est apportée (par défaut)
- **LostFocus:** quand le contrôle perd le focus
- **Explicit:** Fait de manière manuelle

Voici le schéma de la documentation Microsoft :



Notification des changements

Les contrôles WPF sont implémentés de telle sorte que la modification d'une propriété déclenche automatiquement une notification d'un changement. Notification à laquelle on peut s'abonner pour exécuter du code.

Mais voilà, nos classes C# ne le font pas !! Si je déclare une simple propriété Nom de type string, je lui mets une valeur, rien ne se passe !

Il faut donc faire quelque chose de plus, et ce quelque chose, c'est d'implémenter l'interface **INotifyPropertyChanged**.

```
public class MaClasse: INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string _nom;

    public string Nom
    {
        get { return _nom; }
        set
        {
            if (_nom != value)
            {
                _nom = value;
                PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(nameof(Nom)));
            }
        }
    }
}
```

Conversion de données

Mais il arrive souvent que le type d'une propriété d'un contrôle ne corresponde pas au type de la propriété de l'objet attaché.

Par exemple, d'un côté, j'ai une propriété booléenne IsVisible et je veux le « binder » à la propriété Visibility d'un contrôle. Faire classiquement

```
<TextBox Visibility="{Binding Path=IsVisible}"/>
```

Cela ne marche pas. J'ai besoin de « convertir » la valeur lors de la définition de la liaison.

Le framework fournit la solution, avec les convertisseurs (Converters).

Il y en a 2 types :

- A valeur unique
- A valeur multiple

Convertisseur à valeur unique

La première étape consiste à créer une classe, qui implémente l'interface IValueConverter. Par convention, on suffixe le nom des classes par Converter.

Notre convertisseur booléen <-> Visibility donnerait :

```
[ValueConversion(typeof(bool), typeof(Visibility))]
public class BoolVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
    {
        return (bool?)value == true
                    ? Visibility.Visible
                    : Visibility.Collapsed);
    }

    public object ConvertBack(object value, Type targetType, object
parameter, CultureInfo culture)
    {
        return null;
    }
}
```

Pour en savoir plus

Ressources annexes

- Documentation Microsoft : https://docs.microsoft.com/fr-fr/dotnet/framework/wpf/data/data-binding-wpf