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 événements routés
- Les liaisons de données
- Les styles et les modèles

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.

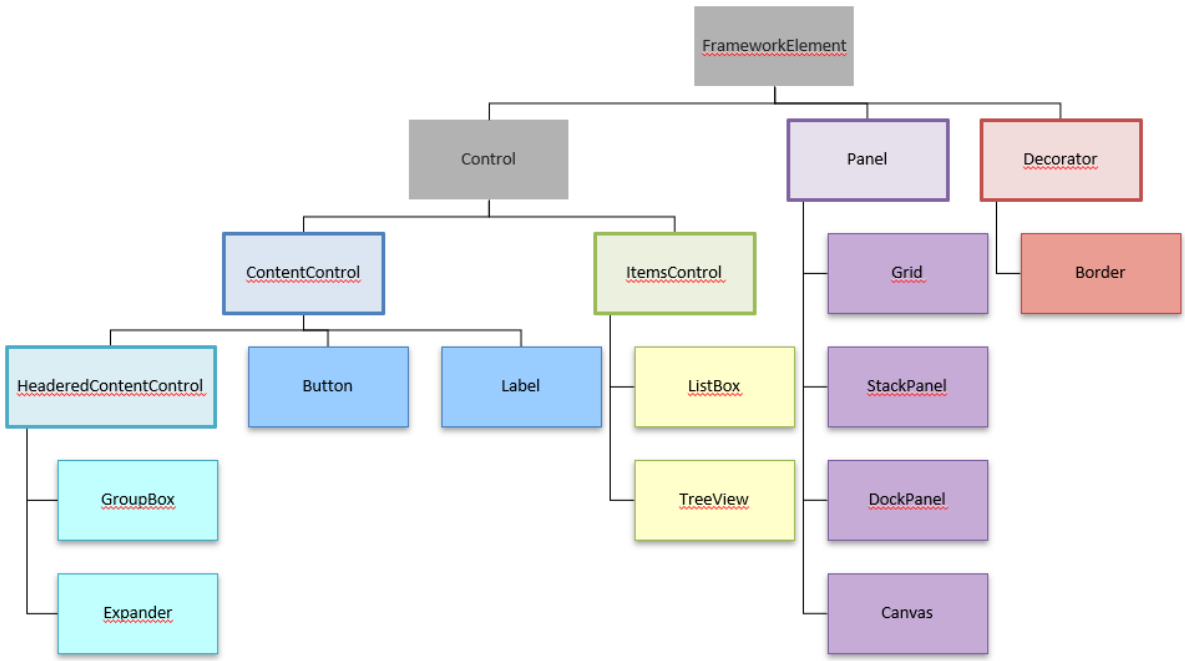
D'où l'importance de veiller à la performance de son code, surtout pour les composants complexes !

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…).

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



Contrôles conteneurs

Un conteneur est un contrôle qui permettent de définir le « layout », c'est-à-dire la mise en page d'un écran.

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)
- **Grid :** l'écran est « découpé » en lignes et en colonnes, et les contrôles sont positionnés sur cette grille.
- **DockPanel :** les contrôles sont positionnés sur les côtés (haut, bas, gauche, droite, milieu)
- **StackPanel :** les contrôles sont affichés à la suite (verticalement ou horizontalement)
- **WrapPanel :** les contrôles sont ajoutés horizontalement à la suite, dans la largeur du conteneur. Les contrôles sont renvoyés à la ligne s'il n'y a pas assez de place pour les afficher.

Contrôles de contenu

Les contrôles de contenu hérite de la classe **ContentControl**. Ils 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>
```

Remarque : la propriété Content peut contenir tout type, pas uniquement des chaînes de caractères.

Prenons le cas d'un bouton qui affiche une image avec du texte :

```
<Button>
    <Button.Content>
        <StackPanel Orientation="Horizontal">
            <Image Source="[chemin de l'image]"/>
            <TextBlock>Valider</TextBlock>
        </StackPanel>
    </Button.Content>
</Button>
```

Dans cette famille, ajoutons les contrôles avec un en-tête (header en anglais). Ils héritent de la classe HeaderedContentControl et proposent, en plus de la propriété Content, la propriété **Header** (qui n'est pas obligatoirement une chaîne de caractères).

Contrôles de liste

Les contrôles de liste héritent de la classe **ItemsControl**. 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>
  <ListBoxItem Content="Enfant 1"/>
  <ListBoxItem Content="Enfant 2"/>
  <ListBoxItem Content="Enfant 3"/>
</ListBox>
```

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

Positionnement des contrôles

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

Pour l'espacement extérieur, on utilise la propriété **Margin**. Pour l'espacement intérieur, 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

EVENEMENTS ROUTES (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.

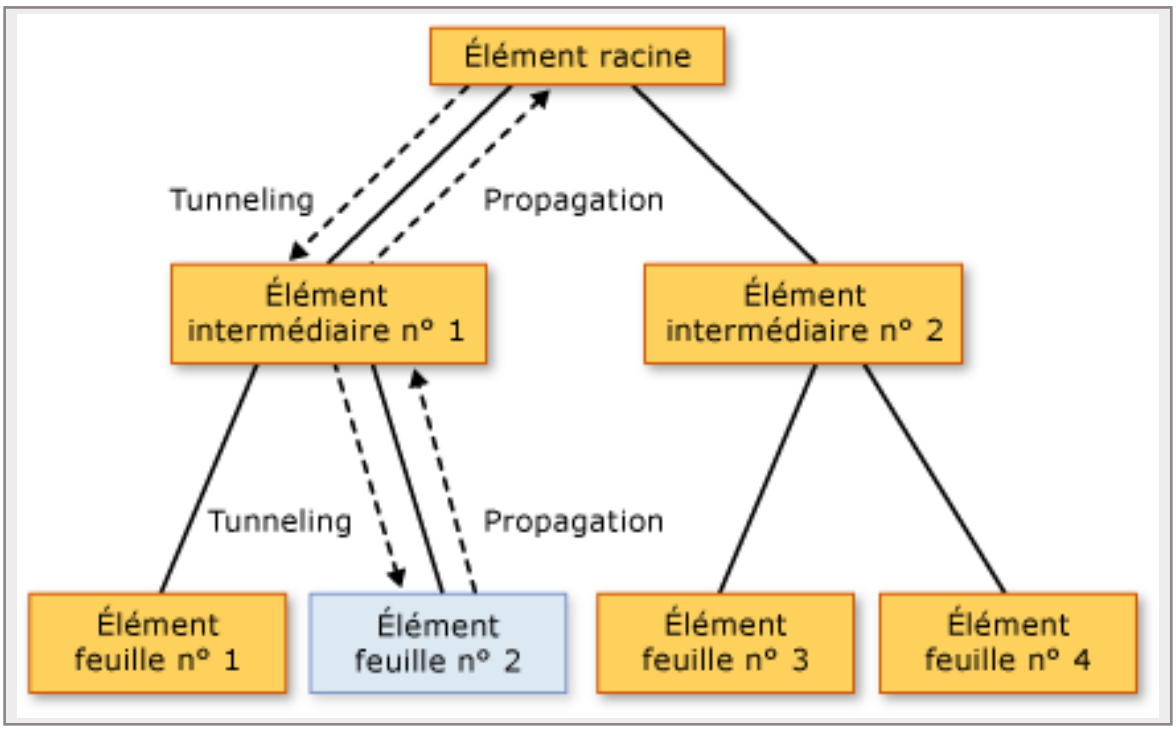
Mais en réalité, quand le moteur WPF affiche notre fenêtre, il y a beaucoup plus d'éléments dans l'arbre. Les contrôles WPF sont en eux-mêmes une composition d'autres éléments visuels (des contrôles conteneurs, ou même des éléments « bas-niveau » comme des Border, Rectangle…).
L'arbre visuel correspond 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 3 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
- **Direct** : événement « classique »

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 ?

Pour en savoir plus

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

LES RESSOURCES

Pour ceux qui ont déjà réalisé des applications .NET, de type ASP.NET ou encore Windows Forms, la notion de ressources n'est pas inconnue.

Un fichier de ressources a pour extension .resx et il sert à stocker des données (texte, images…). Ce(s) fichiers de ressources sont embarqués dans le code généré.
Une des utilisations les plus fréquentes est la localisation (rien à voir avec la localisation géographique, quoique 😊), c'est-à-dire la gestion de ressources en fonction d'une langue.

WPF prend très bien en charge ces fichiers de ressources. Mais il comprend aussi une notion qui lui est propre.

Une ressource est fondamentalement un objet que l'on va stocker à un endroit, pour l'utiliser par la suite. Ce peut être n'importe quel type d'objet : un style, une couleur, un modèle (templates).

C'est qui importe, ce n'est pas vraiment le type de l'élément, mais l'endroit où il est placé. Cet endroit déterminera sa portée, c'est-à-dire les endroits où les éléments définis seront utilisables (parce qu'accessibles).

Portée des ressources

Tout élément héritant la classe FrameworkElement (soit la grande majorité des éléments qu'on utilise en XAML) possède la propriété Ressources.
Le type de cette propriété est ResourceDictionary, donc une liste de clé/valeur.

La portée d'une ressource correspond donc à la visibilité de l'élément sur lequel elle est définie.

Dans l'exemple ci-dessous :

```
<Window …>
    <Window.Resources>
        <Style TargetType="TextBlock">…</Style>
    </Window.Resources>
    <StackPanel>
        <TextBlock x:Name="TextBlock1" … />
        <Grid>
            <Grid.Resources>
                <Style TargetType="TextBlock">…</Style>
            </Grid.Resources>
            <TextBlock x:Name="TextBlock2" … />
        </Grid>
…
</Window>
```

Nous avons ici 2 styles (voir le chapitre dédié aux style pour comprendre de quoi il s'agit) :

- Un est défini au niveau de la fenêtre
- Un autre est défini au niveau de la grille

Concrètement, quel style s'appliquera au TextBlock au-dessus de la grille ?
Celui défini au niveau de la fenêtre.
Vous devinez la conclusion pour le TextBlock à l'intérieur de la grille…

La portée de la ressource est descendante, elle concerne tous les enfants de l'élément sur lequel la ressource est défini.

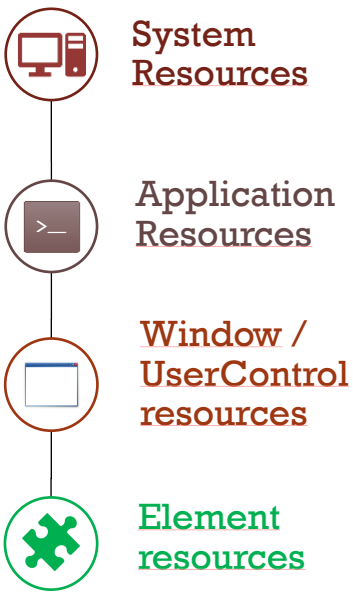
Il est possible de placer les ressources au niveau de **l'application**, donc accessibles par tous les éléments de l'application (les fenêtres par exemple).

Il suffit de placer les ressources dans le fichier App.xaml.

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

Et il y a des ressources plus haut niveau encore ! Si, si ! On les appelle **ressources système,** elles font partie du framework. C'est là que se trouvent le rendu de tous les contrôles du framework.

Voici un schéma pour résumer ces portées :



Fichier de ressources

La propriété Resources de la balise Application peut devenir très grande. Pour la lisibilité du code, et aussi pour pouvoir travailler avec des équipes dédiées au design par exemple, il est avantageux de regrouper les ressources dans des fichiers séparés.

Ces fichiers sont des fichiers xaml définis ainsi :

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

… Ressources

</ResourceDictionary>
```

Dans la propriété Resources de la classe Application, on va faire une référence à ces fichiers créés.

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>

            <ResourceDictionary Source="Chemin du fichier de ressources"/>

        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

Notez la présence de la balise ResourceDictionary.MergedDictionaries. En effet, la propriété Resources n'accepte qu'un ResourceDictionary. La classe MergedDictionaries permet de fusionner plusieurs ResourceDictionary.

Référencer des ressources d'un autre assembly

Il est tout à fait possible de référencer des fichiers de ressources qui sont définis dans un **autre projet ou une autre bibliothèque** référencée par le projet.

Dans ce cas-là, le chemin doit comporter le nom de l'assembly suivi du chemin du fichier de ressources dans cet assembly.

Voici la syntaxe :

```
<ResourceDictionary Source="pack://application:,,,/[Nom de
l'assembly];component/[Chemin du fichier de ressource dans l'assembly]" />
```

Pour en savoir plus

- Documentation Microsoft : https://docs.microsoft.com/fr-fr/dotnet/framework/wpf/advanced/xaml-resources
- Documentation Microsoft : https://docs.microsoft.com/fr-fr/dotnet/framework/wpf/app-development/pack-uris-in-wpf

LIAISON DE DONNEES (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"…>
    <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>
```

On peut renseigner le DataContext de la fenêtre depuis le code.

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 tout 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.

Liaison sur d'autres sources que le contexte de données

Le mécanisme ne se limite pas à s'attacher à des objets du contexte de données. On peut aussi s'attacher à des propriétés de contrôles !

Pour cela, il faut renseigner l'attribut ElementName dans le Binding, la valeur de l'attribut Path étant la propriété du contrôle qu'on souhaite gérer.

```
<TextBox x:Name="textBox1" Text="Coucou"/>
<TextBlock Text="{Binding ElementName=textBox1, Path=Text}"/>
```

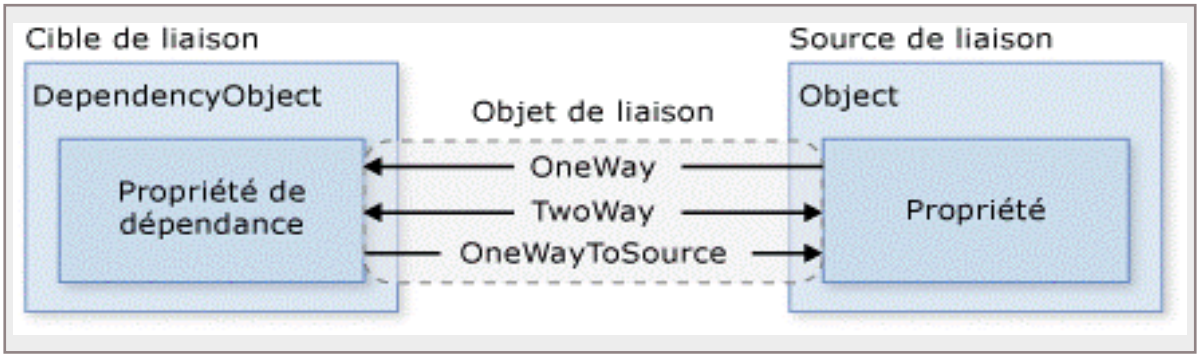
Modes de liaison de données

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 :



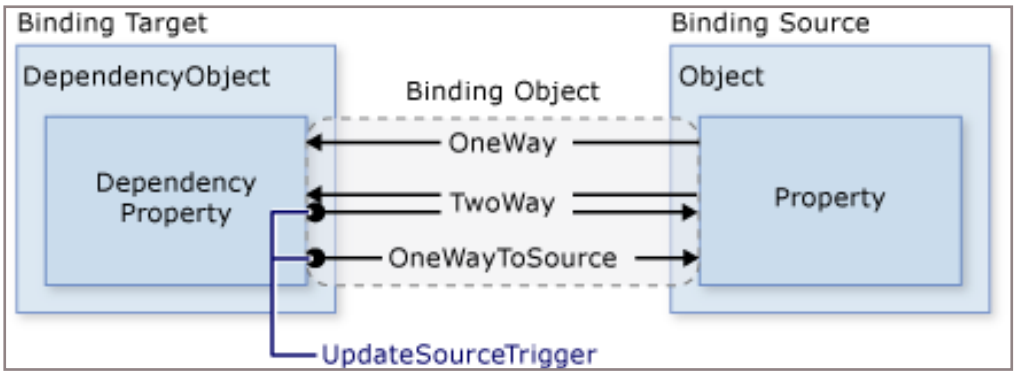
Déclenchement de la mise à jour

Il est même possible de paramétrer le moment où l'objet attaché est mis à jour.

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 :



Sur un TextBox par exemple, le déclenchement se fait quand le contrôle perd le focus (LostFocus).

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 que la classe Binding gère toute seule et qui lui permet de mettre à jour l'objet attaché.

Mais voilà, nos classes C# ne déclenchent aucune notification lors d'un changement !! 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;
    }
}
```

La méthode Convert prend un entrée une valeur de type booléen et en sortie renvoie une valeur d'énumération Visibility.

La signature de la méthode Convert est imposée par l'interface. On a donc des type object en paramètre et en sortie. Il faut donc penser à transtyper ces valeurs dans le code.

La méthode ConvertBack permet de faire l'opération dans l'autre sens. Elle est utile si le binding est en mode TwoWay ou OneWayToSource. Autrement retourner null est suffisant.

Côté XAML, ça donne :

Syntaxe complète :

```
<!-- Ne pas oublier d'instancier le convertisseur dans les ressources -->

<TextBlock>
    <TextBlock.Visibility>
        <Binding Path="IsVisible" Converter="{StaticResource
BoolVisibilityConverter}" />
    </TextBlock.Visibility>
</TextBlock>
```

Syntaxe simplifiée :

```
<!-- Ne pas oublier d'instancier le convertisseur dans les ressources -->

<TextBlock Visibility="{Binding IsVisible, Converter="{StaticResource
BoolVisibilityConverter}}"/>
```

Convertisseur à valeur multiple

Il peut arriver qu'on ait en entrée plusieurs valeurs et qu'on doit gérer un seul type en sortie.

C'est le cas quand on utilise du binding multiple par exemple. On a donc un tableau de valeurs en entrée.

Pour implémenter un convertisseur multiple, on crée une classe qui implémente l'interface **IMultiValueConverter**.

Imaginons la classe Personne avec les propriétés Genre (Homme, Femme), Nom, Prénom.

Je veux afficher dans un TextBlock le nom complet : M. [Prénom] [NOM] pour un homme, Mme [Prénom] [NOM] pour une femme.

Côté binding, j'aurai quelque chose du genre :

```
<TextBlock>
    <TextBlock.Text>
        <MultiBinding>
            <Binding Path="Genre"/>
            <Binding Path="Prenom"/>
            <Binding Path="Nom"/>
        </MultiBinding>
    </TextBlock.Text>
</TextBlock>
```

Remarque: pour utiliser du binding multiple, il faut obligatoirement utiliser la syntaxe complète.

Pour gérer mon affichage, je dois donc implémenter un convertisseur :

```
public class PersonneNomCompletMultiValueConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType, object
parameter, System.Globalization.CultureInfo culture)
    {
        if (values == null || values.Length < 3)
            return null;
```

```
        var genre = (Genre)values[0] ;
        var renduGenre = genre == Genre.Homme ? "M." : "Mme" ;

            return $"{renduGenre} {values[1].ToString()}
{values[2].ToString().ToUpperCase()}";
    }
}
```

On comprend vite que l'ordre d'apparition du Binding dans le XAML détermine l'ordre des valeurs passés en paramètres.

Il ne reste plus qu'à modifier le XAML pour indiquer le convertisseur à utiliser sur le binding multiple.

```
<!-- Ne pas oublier d'instancier le convertisseur dans les ressources -->

<TextBlock>
    <TextBlock.Text>
        <MultiBinding Converter="{StaticResource PersonneConverter}">
            <Binding Path="Genre"/>
            <Binding Path="Prenom"/>
            <Binding Path="Nom"/>
        </MultiBinding>
    </TextBlock.Text>
</TextBlock>
```

Validation des données

Tout bon logiciel qui contient des formulaires de saisie se doit de gérer la validation.

Il y a plusieurs mécanismes de validation que WPF prend en charge.

On va ici aborder les règles de validation.

Validation Rules

Une règle de validation est une classe, héritant de la classe **ValidationRule**, que l'on place sur la déclaration d'une liaison de données (data binding).

A chaque modification du contrôle attaché, les règles de validation sont évaluées.

Implémentons une règle de validation qui vérifie qu'une chaîne de caractères respecte une expression régulière (ou regex) définie.

```
public class RegexValidation : ValidationRule
{
    public string Pattern { get ; set; }

    public override ValidationResult Validate(object value, CultureInfo
cultureInfo)
    {
        Regex regex = new Regex(Pattern);
        Match match = regex.Match(value.ToString());
        if (match == null || match == Match.Empty)
        {
            return new ValidationResult(false, "Invalid format");
        }
        else
        {
            return ValidationResult.ValidResult;
        }
    }
}
```

Puis, côté XAML, utilisons cette règle de validation sur une zone de texte pour s'assurer que la valeur est une adresse email valide.

```
<TextBox>
    <TextBox.Text>
        <Binding Path="Email">
           <Binding.ValidationRules>
               <local:RegexValidationRule Pattern="^[\w-\.]+@([\w-
]+\.)+[\w-]{2,4}$" />
           </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Remarque 1 : il est nécessaire d'utiliser la syntaxe de binding complète si on veut appliquer des règles de validation.

Remarque 2 : il est tout à fait possible de mettre plusieurs règles de validation.

Pour en savoir plus

- Documentation Microsoft : https://docs.microsoft.com/fr-fr/dotnet/framework/wpf/data/data-binding-wpf
- Blog : https://blog.magnusmontin.net/2013/08/26/data-validation-in-wpf/

STYLES ET MODELES (TEMPLATES)

Une des forces de WPF, c'est la séparation entre la description de l'interface graphique du code de l'application. Avec les fichiers de ressources, il est possible de grouper des éléments et de les partager. Parmi ces éléments, on trouve les styles et les modèles (templates).

Les styles

Un style permet de regrouper des caractéristiques visuelles d'un contrôle.

Déclaration XAML

Un style se définit par la balise Style, qui contient des balises Setter.

Exemple d'un style pour des TextBlock pour que le texte soit en gras, bleu foncé d'une taille de 15.

```
<Style TargetType="TextBlock">
    <Setter Property="FontSize" Value="15" />
    <Setter Property="FontWeight" Value="Bold" />
    <Setter Property="Foreground" Value="DarkBlue" />
</Style>
```

Remarque : ce style s'appliquera à tous les TextBlock dans le scope de sa déclaration.

Si on ne veut pas un style global, on peut lui donner une clé avec l'attribut x:Key (la valeur est une chaîne de caractères).

Modifions le style précédent en lui donnant la clé « TitleTextBlockStyle »

```
<!-- On peut aussi utiliser TargetType="{x:Type TextBlock}" -->
<Style x:Key="TitleTextBlockStyle" TargetType="TextBlock">
    <Setter Property="FontSize" Value="20" />
    <Setter Property="Style" Value="Italic" />
</Style>
```

Ici le style a la clé « TitleTextBlockStyle », ce qui implique qu'il faudra le référencer explicitement pour l'utiliser.

Pour qu'un TextBlock applique ce style, on renseigne sa propriété Style :

```
<TextBlock Style="{StaticResource TitleTextBlockStyle}" Text="Titre" />
```

La syntaxe {**StaticResource** [clé du style]} ressemble à la syntaxe simplifiée du data binding. Static signifie que le style est évalué une seule fois. Les autres appels à ce style utiliseront la version en cache.

Il y a aussi la syntaxe {**DynamicResource**…}. Dans ce cas-là, le style sera évalué à chaque appel. L'avantage, c'est qu'on peut imaginer un mécanisme de skin (ou thèmes) qu'on pourrait changer à la volée (sans devoir redémarrer l'application). Mais attention, ce référencement est plus coûteux.

Héritage d'un style

WPF va encore plus loin : il est possible de créer des sous-styles !

Imaginons maintenant un style « SubTitleTextBlockStyle » :

```
<Style x:Key="TitleTextBlockStyle"
       TargetType="TextBlock"
       BasedOn="{StaticResource TitleTextBlockStyle}">
    <Setter Property="FontSize" Value="13" />
    <Setter Property="Foreground" Value="LightBlue" />
</Style>
```

Le sous-style récupère toutes les propriétés définies sur le style parent (ou plutôt de tous les styles parents). Il sucharge les propriétés FontSize et Foreground. Les TextBlock qui utilisent ce style afficheront un texte toujours gras, mais maintenant bleu clair et d'une taille de 13.

Pour hériter d'un style défini par défaut (sans clé), voici la syntaxe :

```
<Style x:Key="TitleTextBlockStyle"
       TargetType="TextBlock"
       BasedOn="{StaticResource {x:Type TextBlock}}">
    …
</Style>
```

On récupère le style qui s'applique par défaut au type TextBlock.

Les templates

Les styles concernent les propriétés des contrôles. Il est possible de spécifier **comment** un élément doit être rendu. Par élément, j'entends un contrôle (un bouton par exemple) ou une instance d'une classe C# qu'on a écrite.

C'est là qu'entre en jeu le mécanisme des templates (ou modèles en français).

Modèle de données (Data templates)

On travaille en programmation orientée objet, on a donc de nombreuses classes, et certaines peut être utilisées à différents endroits de l'application. Le modèle de données permet de définir, pour une classe donnée, un rendu.

Imaginons la classe Personne, avec les propriétés Nom, Prenom, Adresse, Pays.

Je voudrais l'afficher comme suit :
Prenom Nom
Adresse
Pays

On peut utiliser des TextBlock, liés aux différentes propriétés de notre classe, le tout dans un StackPanel.

Et si je souhaite afficher des personnes dans plusieurs endroits de mon application ? Je fais du copier-coller ?

Non, je définis un modèle de données et je le mets dans un fichier de ressources référencé au niveau de l'application.

En XAML, ça donne :

```
<!-- Penser à importer le namespace de la classe Personne (ici local) -->
…
<DataTemplate DataType="{x:Type local:Personne">
    <StackPanel>
        <StackPanel Orientiation="Horizontal" TextBlock.FontWeight="Bold">
            <TextBlock Text="{Binding Prenom}" />
            <TextBlock Text="{Binding Nom}" />
        </StackPanel>
        <TextBlock Text="{Binding Adresse}" FontStyle="Italic" />
        <TextBlock Text="{Binding Pays}" />
    </StackPanel>
</DataTemplate>
```

Remarque: quand on renseigne l'attribut DataType, Visual Studio propose les propriétés du type quand on renseigne le binding.

Remarque 2 : ici, on a défini un modèle par défaut pour la classe Personne. Il est possible de créer autant de modèle d'une même classe qu'on souhaite, en donnant des clés (attribut x:Key) différentes.

Data triggers

XAML nous permet de modifier des caractéristiques visuelles quand des propriétés choisies d'un objet attaché reçoivent des valeurs attendues.

Si on reprend l'exemple de la classe Personne, je voudrais que le TextBlock affichant le pays devienne gras si la valeur est « France ».

```
<DataTemplate DataType="{x:Type local:Personne">
    <StackPanel>
        …
        <TextBlock x:Name="tbPays" Text="{Binding Pays}" />
    </StackPanel>
    <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding Pays}" Value="France">
            <Setter TargetName="tbPays"
                    Property="FontWeight"
                    Value="Bold" />
        </DataTrigger>
    </DataTemplate.Triggers>
</DataTemplate>
```

On a donné un nom au TextBlock affichant le pays. Puis, on a déclaré un déclencheur sur la propriété Pays. Si la valeur vaut « France » (ou si elle le devient suite à une modification), on modifie le TextBlock en gras.

Modèle de contrôles (Control templates)

Le modèle de contrôle permet de redéfinir complètement le rendu d'un contrôle.

On peut changer le rendu d'un bouton pour qu'il soit circulaire, ou ce qu'on souhaite d'autre 😊.

Toutefois, ce point ne sera pas abordé dans ce support.

Pour en savoir plus

- Documentation Microsoft : https://docs.microsoft.com/fr-fr/dotnet/framework/wpf/controls/styling-and-templating