eBook Gratuit

APPRENEZ wpf

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#wpf

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: wpf

It is an unofficial and free wpf ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official wpf.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec wpf

Remarques

WPF (Windows Presentation Foundation) est la technologie de présentation recommandée par Microsoft pour les applications de bureau Windows classiques. WPF ne doit pas être confondu avec UWP (Universal Windows Platform) bien que des similitudes existent entre les deux.

WPF encourage les applications basées sur les données en mettant l'accent sur le multimédia, l'animation et la liaison de données. Les interfaces sont créées à l'aide d'un langage appelé XAML (eXtensible Application Markup Language), dérivé du langage XML. XAML aide les programmeurs WPF à maintenir la séparation entre la conception visuelle et la logique d'interface.

Contrairement à son prédécesseur Windows Forms, WPF utilise un modèle de boîte pour mettre en forme tous les éléments de l'interface. Chaque élément a une hauteur, une largeur et des marges et est disposé à l'écran par rapport à son parent.

WPF signifie Windows Presentation Foundation et est également connu sous son nom de code Avalon. C'est un cadre graphique et une partie de Microsoft .NET Framework. WPF est pré-installé dans Windows Vista, 7, 8 et 10 et peut être installé sur Windows XP et Server 2003.

Versions

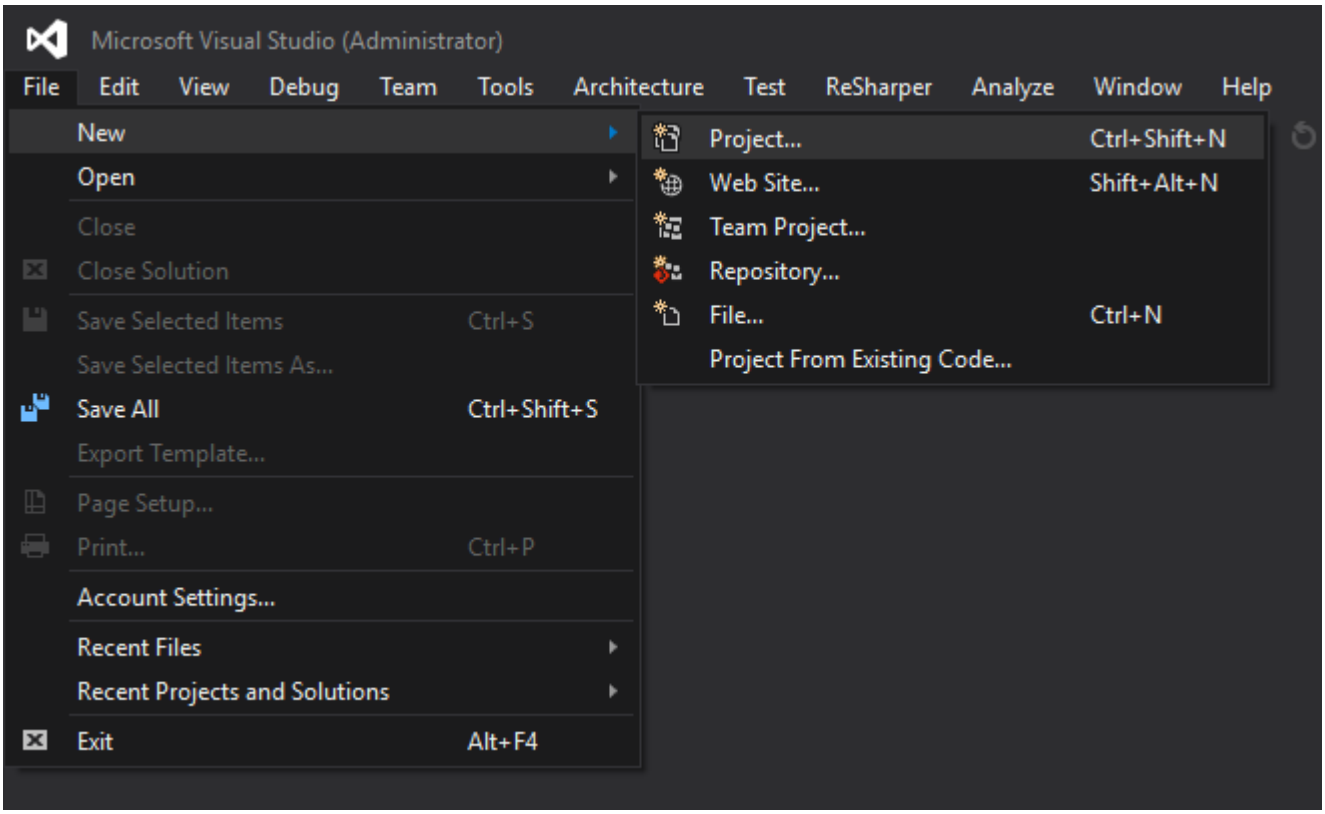
Version 4.6.1 - Décembre 2015

Examples

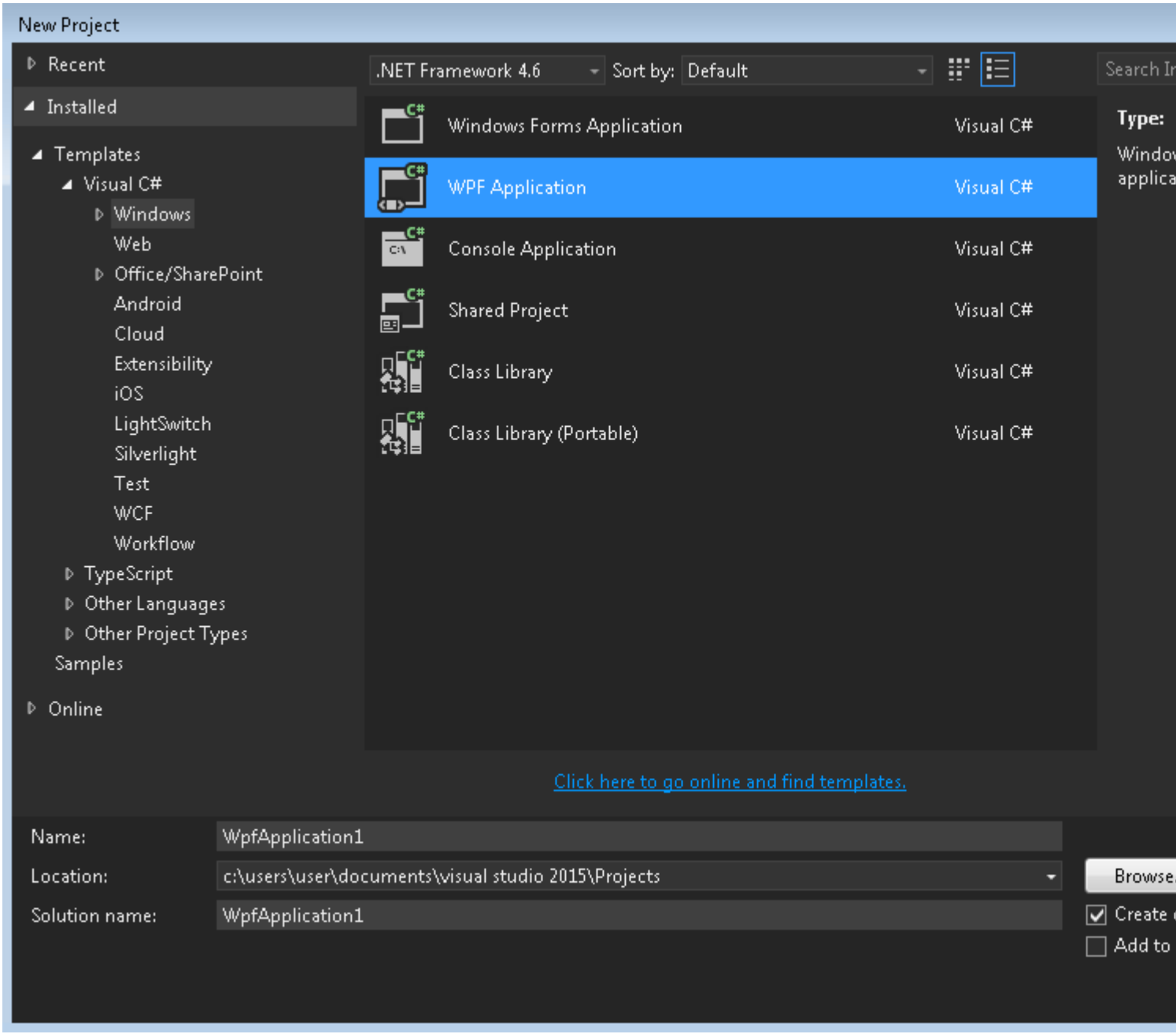
Application Hello World

Pour créer et exécuter un nouveau projet WPF dans Visual Studio:

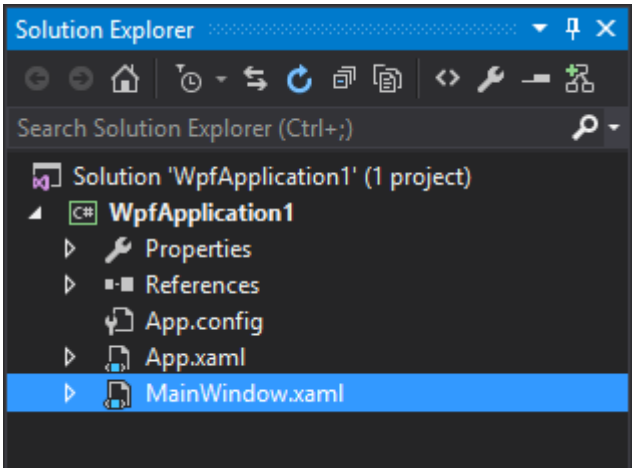
1. Cliquez sur **Fichier → Nouveau → Projet**



2. Sélectionnez un modèle en cliquant sur **Modèles → Visual C # → Windows → Application WPF** et appuyez sur `OK` :



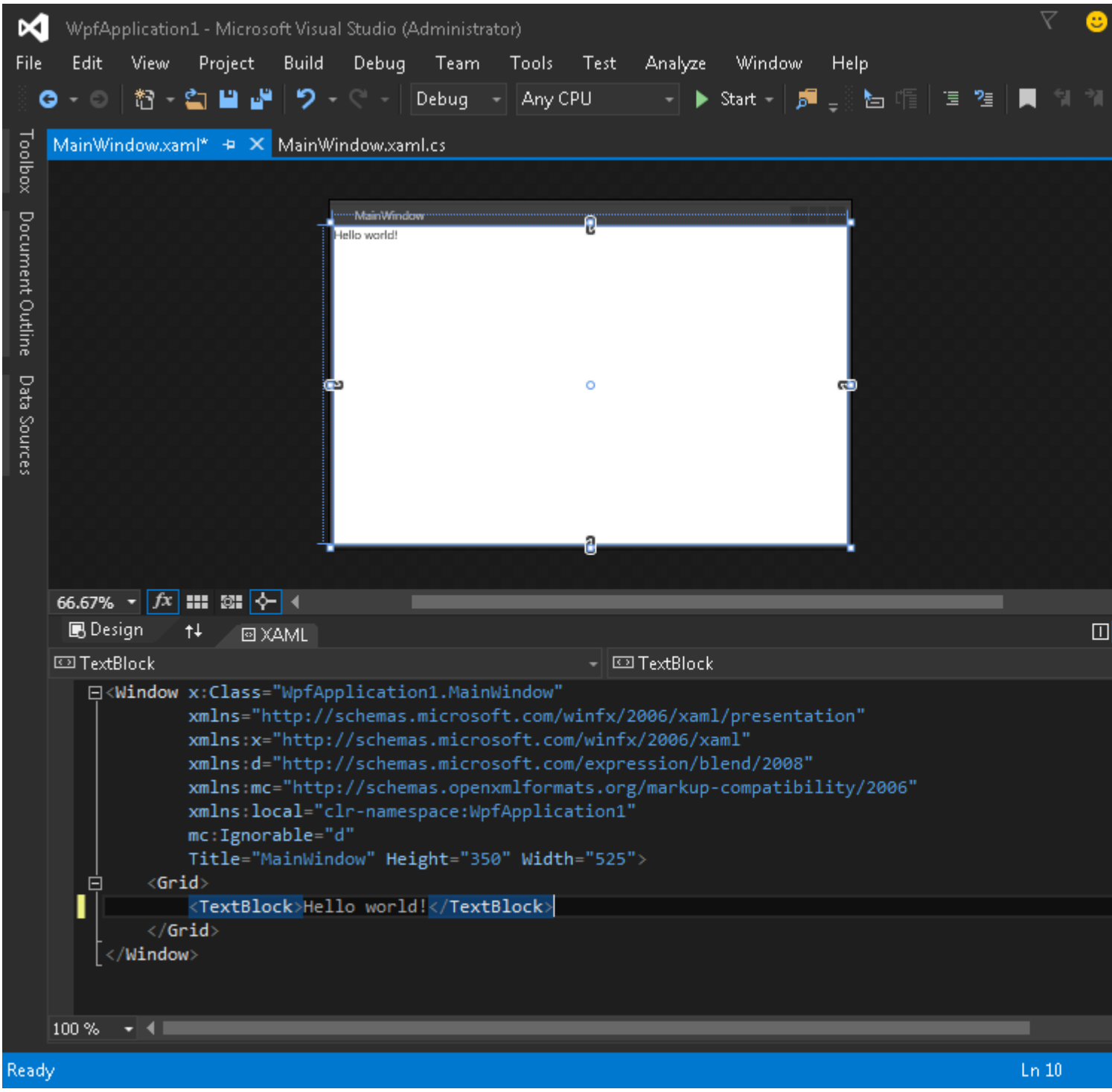
3. Ouvrez le fichier **MainWindow.xaml** dans l' *Explorateur de solutions* (si vous ne voyez pas la fenêtre *Explorateur de solutions* , ouvrez-la en cliquant sur **Affichage → Explorateur de solutions**):



4. Dans la section *XAML* (par défaut sous la section *Conception*) ajoutez ce code

```
<TextBlock>Hello world!</TextBlock>
```

à l'intérieur de la `Grid` :

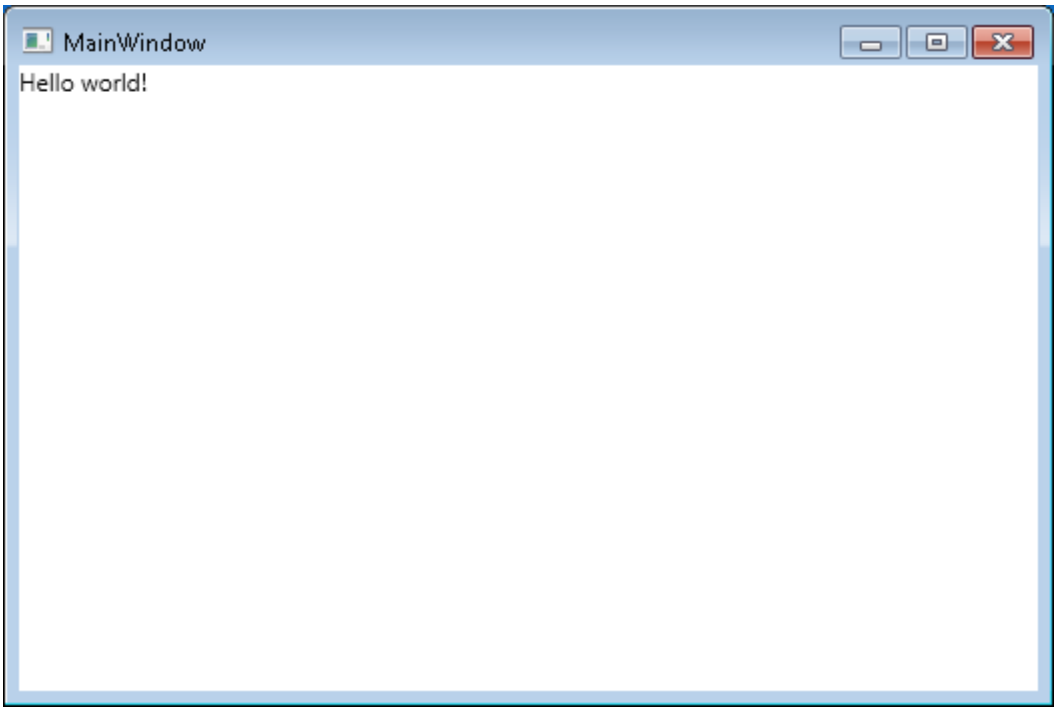


Le code devrait ressembler à:

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

```
        xmlns:local="clr-namespace:WpfApplication1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <TextBlock>Hello world!</TextBlock>
    </Grid>
</Window>
```

5. Exécutez l'application en appuyant sur `F5` ou en cliquant sur le menu **Déboguer → Démarrer le débogage** . Cela devrait ressembler à:



Lire Démarrer avec wpf en ligne: https://riptutorial.com/fr/wpf/topic/820/demarrer-avec-wpf

Chapitre 2: Affinité des threads Accès aux éléments de l'interface utilisateur

Examples

Accès à un élément d'interface utilisateur à partir d'une tâche

Tous les éléments d'interface utilisateur créés et résident dans le thread principal d'un programme. L'accès à ces derniers à partir d'un autre thread est interdit par le runtime du framework .net. Fondamentalement, c'est parce que tous les éléments d'interface utilisateur sont des **ressources sensibles au thread** et que l'accès à une ressource dans un environnement multithread nécessite d'être thread-safe. Si l'accès à cet objet est autorisé, la cohérence sera affectée en premier lieu.

Considérez ce scénario:

Nous avons un calcul dans une tâche. Les tâches sont exécutées dans un autre thread que le thread principal. Pendant que le calcul se poursuit, nous devons mettre à jour une barre de progression. Pour faire ça:

```
//Prepare the action
Action taskAction = new Action( () => {
    int progress = 0;
    Action invokeAction = new Action( () => { progressBar.Value = progress; });
    while (progress <= 100) {
        progress = CalculateSomething();
        progressBar.Dispatcher.Invoke( invokeAction );
    }
} );

//After .net 4.5
Task.Run( taskAction );

//Before .net 4.5
Task.Factory.StartNew( taskAction ,
    CancellationToken.None,
    TaskCreationOptions.DenyChildAttach,
    TaskScheduler.Default);
```

Chaque élément d'interface utilisateur possède un objet Dispatcher provenant de son ancêtre `DispatcherObject` (dans l'espace de noms `System.Windows.Threading`). Dispatcher exécute le délégué spécifié de manière synchrone à la priorité spécifiée sur le thread auquel le Dispatcher est associé. Puisque l'exécution est synchronisée, la tâche de l'appelant doit attendre son résultat. Cela nous donne la possibilité d'utiliser `int progress` également à l'intérieur d'un délégué répartiteur.

Nous souhaitons peut-être mettre à jour un élément d'interface utilisateur de manière asynchrone, puis `invokeAction` modifications de définition d' `invokeAction` :

```
//Prepare the action
Action taskAction = new Action( () => {
    int progress = 0;
    Action<int> invokeAction = new Action<int>( (i) => { progressBar.Value = i; } )
    while (progress <= 100) {
        progress = CalculateSomething();
        progressBar.Dispatcher.BeginInvoke(
            invokeAction,
            progress );
    }
} );

//After .net 4.5
Task.Run( taskAction );

//Before .net 4.5
Task.Factory.StartNew( taskAction ,
    CancellationToken.None,
    TaskCreationOptions.DenyChildAttach,
    TaskScheduler.Default);
```

Cette fois, nous avons emballé le `int progress` et l'avons utilisé comme paramètre pour délégué.

Lire Affinité des threads Accès aux éléments de l'interface utilisateur en ligne: https://riptutorial.com/fr/wpf/topic/6128/affinite-des-threads-acces-aux-elements-de-l-interface-utilisateur

Chapitre 3: Architecture WPF

Examples

DispatcherObject

Dérive de

`Object`

Membres clés

```
public Dispatcher Dispatcher { get; }
```

Résumé

La plupart des objets dans WPF dérivent de `DispatcherObject` , qui fournit les constructions de base pour gérer la concurrence et le threading. Ces objets sont associés à un répartiteur.

Seul le thread sur lequel le répartiteur a été créé peut accéder directement à DispatcherObject. Pour accéder à DispatcherObject à partir d'un thread autre que le thread sur `BeginInvoke` DispatcherObject a été créé, un appel à `Invoke` ou `BeginInvoke` sur le répartiteur `BeginInvoke` l'objet est associé est requis.

DependencyObject

Dérive de

`DispatcherObject`

Membres clés

```
public object GetValue(DependencyProperty dp);
public void   SetValue(DependencyProperty dp, object value);
```

Résumé

Les classes dérivées de `DependencyObject` participent au système de propriété de dépendance , qui inclut l'enregistrement des propriétés de dépendance et la fourniture d'une identification et d'informations sur ces propriétés. Les propriétés de dépendance étant la pierre angulaire du développement WPF, tous les contrôles WPF dérivent finalement de `DependencyObject` .

Lire Architecture WPF en ligne: https://riptutorial.com/fr/wpf/topic/3571/architecture-wpf

Chapitre 4: Comportements WPF

Introduction

Les comportements WPF permettent à un développeur de modifier la manière dont les contrôles WPF agissent en réponse aux événements système et utilisateur. Les comportements héritent de la classe `Behavior` de l'espace de noms `System.Windows.Interactity` . Cet espace de noms fait partie du SDK Expression Blend global, mais une version plus légère, adaptée aux bibliothèques de comportement, est disponible sous la forme [package nuget] [1]. [1]: https://www.nuget.org/packages/System.Windows.Interactivity.WPF/

Examples

Comportement simple pour intercepter les événements de la molette de la souris

Mise en œuvre du comportement

Ce comportement entraînera une `ScrollViewer` événements de la molette de la souris depuis un `ScrollViewer` interne vers le `ScrollViewer` parent lorsque celui-ci se trouve à sa limite supérieure ou inférieure. Sans ce comportement, les événements ne pourront jamais sortir du `ScrollViewer` interne.

```
public class BubbleMouseWheelEvents : Behavior<UIElement>
{
    protected override void OnAttached()
    {
        base.OnAttached();
        this.AssociatedObject.PreviewMouseWheel += PreviewMouseWheel;
    }

    protected override void OnDetaching()
    {
        this.AssociatedObject.PreviewMouseWheel -= PreviewMouseWheel;
        base.OnDetaching();
    }

    private void PreviewMouseWheel(object sender, MouseWheelEventArgs e)
    {
        var scrollViewer = AssociatedObject.GetChildOf<ScrollViewer>(includeSelf: true);
        var scrollPos = scrollViewer.ContentVerticalOffset;
        if ((scrollPos == scrollViewer.ScrollableHeight && e.Delta < 0) || (scrollPos == 0 &&
e.Delta > 0))
        {
            UIElement rerouteTo = AssociatedObject;
            if (ReferenceEquals(scrollViewer, AssociatedObject))
            {
                rerouteTo = (UIElement) VisualTreeHelper.GetParent(AssociatedObject);
            }

            e.Handled = true;
```

```
            var e2 = new MouseWheelEventArgs(e.MouseDevice, e.Timestamp, e.Delta);
            e2.RoutedEvent = UIElement.MouseWheelEvent;
            rerouteTo.RaiseEvent(e2);
        }
    }
}
```

Comportements sous-classe de la classe de base `Behavior<T>` , `T` étant le type de contrôle auquel il peut être associé, dans ce cas `UIElement` . Lorsque le `Behavior` est instancié à partir de XAML, la méthode `OnAttached` est appelée. Cette méthode permet au comportement de se connecter aux événements à partir du contrôle auquel il est associé (via `AssociatedControl`). Une méthode similaire, `OnDetached` est appelée lorsque le comportement doit être décroché de l'élément associé. Des précautions doivent être prises pour supprimer tous les gestionnaires d'événements ou pour nettoyer les objets afin d'éviter les fuites de mémoire.

Ce comportement s'intègre à l'événement `PreviewMouseWheel` , ce qui lui permet d'intercepter l'événement avant que `ScrollViewer` puisse le voir. Il vérifie la position pour voir s'il doit transférer l'événement dans l'arborescence visuelle à une hiérarchie supérieure de `ScrollViewer` . Si c'est le cas, il définit `e.Handled` sur `true` pour empêcher l'action par défaut de l'événement. Il déclenche ensuite un nouveau `MouseWheelEvent` routé vers `AssociatedObject` . Sinon, l'événement est routé normalement.

Attacher le comportement à un élément dans XAML

Tout d'abord, l'espace de noms XML d' `interactivity` doit être étendu à la portée avant de pouvoir être utilisé dans XAML. Ajoutez la ligne suivante aux espaces de noms de votre XAML.

> xmlns: interactivity = " http://schemas.microsoft.com/expression/2010/interactivity "

Le comportement peut être attaché comme suit:

```
<ScrollViewer>
    <!--...Content...-->
    <ScrollViewer>
        <interactivity:Interaction.Behaviors>
            <behaviors:BubbleMouseWheelEvents />
        </interactivity:Interaction.Behaviors>
        <!--...Content...-->
    </ScrollViewer>
    <!--...Content...-->.
</ScrollViewer>
```

Cela crée une collection de `Behaviors` tant que propriété attachée sur le `ScrollViewer` interne qui contient un comportement `BubbleMouseWheelEvents` .

Ce comportement particulier pourrait également être associé à tout contrôle existant contenant un `ScrollViewer` intégré, tel qu'un `GridView` , et il fonctionnerait toujours correctement.

Lire Comportements WPF en ligne: https://riptutorial.com/fr/wpf/topic/8365/comportements-wpf

Chapitre 5: Contrôle de la grille

Examples

Une grille simple

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Column="1" Text="abc"/>
    <TextBlock Grid.Row="1" Grid.Column="1" Text="def"/>
</Grid>
```

Les lignes et les colonnes sont définies en ajoutant des éléments `RowDefinition` et `ColumnDefinition` aux collections correspondantes.

Il peut y avoir un nombre aberrant d'enfants dans la `Grid` . Pour spécifier la ligne ou la colonne qu'un enfant doit placer dans les propriétés attachées, `Grid.Row` et `Grid.Column` sont utilisées. Les numéros de ligne et de colonne sont basés sur zéro. Si aucune ligne ou colonne n'est définie, la valeur par défaut est `0` .

Les enfants placés dans la même rangée et la même colonne sont dessinés par ordre de définition. Ainsi, l'enfant défini en dernier sera dessiné au-dessus de l'enfant défini auparavant.

Enfants de la grille couvrant plusieurs lignes / colonnes

En utilisant les propriétés jointes `Grid.RowSpan` et `Grid.ColumnSpan` , les enfants d'une `Grid` peuvent s'étendre sur plusieurs lignes ou colonnes. Dans l'exemple suivant, le second `TextBlock` couvrira les deuxième et troisième colonnes de la `Grid` .

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Column="2" Text="abc"/>
    <TextBlock Grid.Column="1" Grid.ColumnSpan="2" Text="def"/>
</Grid>
```

Synchronisation de lignes ou de colonnes de plusieurs grilles

Les hauteurs de ligne ou les largeurs de colonne de plusieurs `Grid` peuvent être synchronisées en

définissant un `SharedSizeGroup` commun sur les lignes ou les colonnes à synchroniser. Ensuite, un contrôle parent situé dans l'arborescence au-dessus de la `Grid` doit avoir la propriété attachée `Grid.IsSharedSizeScope` définie sur `True` .

```
<StackPanel Grid.IsSharedSizeScope="True">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" SharedSizeGroup="MyGroup"/>
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        [...]
    </Grid>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" SharedSizeGroup="MyGroup"/>
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        [...]
    </Grid>
</StackPanel>
```

Dans cet exemple, la première colonne des deux `Grid` aura toujours la même largeur, même si l’une d’entre elles est redimensionnée par son contenu.

Lire Contrôle de la grille en ligne: https://riptutorial.com/fr/wpf/topic/6483/controle-de-la-grille

Chapitre 6: Convertisseurs de valeur et de valeurs multiples

Paramètres

Paramètre	Détails
valeur	La valeur produite par la source de liaison.
valeurs	Le tableau de valeurs, produit par la source de liaison.
targetType	Le type de la propriété cible de liaison.
paramètre	Le paramètre de convertisseur à utiliser.
Culture	La culture à utiliser dans le convertisseur.

Remarques

Qu'est-ce que IValueConverter et IMultiValueConverterthey sont

IValueConverter et IMultiValueConverter - interfaces permettant d'appliquer une logique personnalisée à une liaison.

Qu'est-ce qu'ils sont utiles pour

1. Vous avez une certaine valeur de type mais vous voulez afficher les valeurs nulles d'une manière et les nombres positifs d'une autre manière
2. Vous avez une certaine valeur de type et souhaitez afficher l'élément dans un cas et le cacher dans un autre
3. Vous avez une valeur numérique mais vous voulez le montrer en mots
4. Vous avez une valeur numérique mais vous souhaitez afficher des images différentes pour les numéros différents

Ce sont quelques cas simples, mais il y en a beaucoup plus.

Pour des cas comme celui-ci, vous pouvez utiliser un convertisseur de valeur. Ces petites classes, qui implémentent l'interface IValueConverter ou IMultiValueConverter, agiront comme des intermédiaires et traduiront une valeur entre la source et la destination. Ainsi, dans toute situation où vous devez transformer une valeur avant qu'elle n'atteigne sa destination ou qu'elle ne soit à nouveau source, vous avez probablement besoin d'un convertisseur.

Examples

Build-In BooleanToVisibilityConverter [IValueConverter]

Convertisseur entre booléen et visibilité. Récupère la valeur `bool` en entrée et renvoie la valeur de `Visibility` .

Remarque: ce convertisseur existe déjà dans l'espace de noms `System.Windows.Controls` .

```
public sealed class BooleanToVisibilityConverter : IValueConverter
{
    /// <summary>
    /// Convert bool or Nullable bool to Visibility
    /// </summary>
    /// <param name="value">bool or Nullable bool</param>
    /// <param name="targetType">Visibility</param>
    /// <param name="parameter">null</param>
    /// <param name="culture">null</param>
    /// <returns>Visible or Collapsed</returns>
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        bool bValue = false;
        if (value is bool)
        {
            bValue = (bool)value;
        }
        else if (value is Nullable<bool>)
        {
            Nullable<bool> tmp = (Nullable<bool>)value;
            bValue = tmp.HasValue ? tmp.Value : false;
        }
        return (bValue) ? Visibility.Visible : Visibility.Collapsed;
    }

    /// <summary>
    /// Convert Visibility to boolean
    /// </summary>
    /// <param name="value"></param>
    /// <param name="targetType"></param>
    /// <param name="parameter"></param>
    /// <param name="culture"></param>
    /// <returns></returns>
    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        if (value is Visibility)
        {
            return (Visibility)value == Visibility.Visible;
        }
        else
        {
            return false;
        }
    }
}
```

Utiliser le convertisseur

1. Définir une ressource

```
<BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>
```

3. Utilisez-le dans la liaison

```
<Button Visibility="{Binding AllowEditing,
                             Converter={StaticResource BooleanToVisibilityConverter}}"/>
```

Convertisseur avec propriété [IValueConverter]

Montrer comment créer un convertisseur simple avec un paramètre via une propriété, puis le transmettre dans une déclaration. Convertissez la valeur `bool` en `Visibility` . Autorise l'inversion de la valeur du résultat en définissant la propriété `Inverted` sur `True` .

```
public class BooleanToVisibilityConverter : IValueConverter
{
    public bool Inverted { get; set; }

    /// <summary>
    /// Convert bool or Nullable bool to Visibility
    /// </summary>
    /// <param name="value">bool or Nullable bool</param>
    /// <param name="targetType">Visibility</param>
    /// <param name="parameter">null</param>
    /// <param name="culture">null</param>
    /// <returns>Visible or Collapsed</returns>
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        bool bValue = false;
        if (value is bool)
        {
            bValue = (bool)value;
        }
        else if (value is Nullable<bool>)
        {
            Nullable<bool> tmp = (Nullable<bool>)value;
            bValue = tmp ?? false;
        }

        if (Inverted)
            bValue = !bValue;
        return (bValue) ? Visibility.Visible : Visibility.Collapsed;
    }

    /// <summary>
    /// Convert Visibility to boolean
    /// </summary>
    /// <param name="value"></param>
    /// <param name="targetType"></param>
    /// <param name="parameter"></param>
    /// <param name="culture"></param>
```

```
    /// <returns>True or False</returns>
    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        if (value is Visibility)
        {
            return ((Visibility) value == Visibility.Visible) && !Inverted;
        }

        return false;
    }
}
```

Utiliser le convertisseur

1. Définir un espace de noms

```
xmlns:converters="clr-namespace:MyProject.Converters;assembly=MyProject"
```

2. Définir une ressource

```
<converters:BooleanToVisibilityConverter x:Key="BoolToVisibilityInvertedConverter"
                                         Inverted="False"/>
```

3. Utilisez-le dans la liaison

```
<Button Visibility="{Binding AllowEditing, Converter={StaticResource
BoolToVisibilityConverter}}"/>
```

Simple add converter [IMultiValueConverter]

Montrer comment créer un simple convertisseur `IMultiValueConverter` et utiliser `MultiBinding` dans xaml. Obtient la somme de toutes les valeurs transmises par le tableau de `values` .

```
public class AddConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType, object parameter, CultureInfo
culture)
    {
        decimal sum = 0M;

        foreach (string value in values)
        {
            decimal parseResult;
            if (decimal.TryParse(value, out parseResult))
            {
                sum += parseResult;
            }
        }

        return sum.ToString(culture);
    }

    public object[] ConvertBack(object value, Type[] targetTypes, object parameter,
```

```
CultureInfo culture)
    {
        throw new NotSupportedException();
    }
}
```

Utiliser le convertisseur

1. Définir un espace de noms

```
xmlns:converters="clr-namespace:MyProject.Converters;assembly=MyProject"
```

2. Définir une ressource

```
<converters:AddConverter x:Key="AddConverter"/>
```

3. Utilisez-le dans la liaison

```
<StackPanel Orientation="Vertical">
    <TextBox x:Name="TextBox" />
    <TextBox x:Name="TextBox1" />
    <TextBlock >
        <TextBlock.Text>
            <MultiBinding Converter="{StaticResource AddConverter}">
                <Binding Path="Text" ElementName="TextBox"/>
                <Binding Path="Text" ElementName="TextBox1"/>
            </MultiBinding>
         </TextBlock.Text>
    </TextBlock>
</StackPanel>
```

Convertisseurs d'utilisation avec ConverterParameter

Montrer comment créer un convertisseur simple et utiliser `ConverterParameter` pour passer le paramètre au convertisseur. Multipliez la valeur par le coefficient passé dans ConverterParameter.

```
public class MultiplyConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        if (value == null)
            return 0;

        if (parameter == null)
            parameter = 1;

        double number;
        double coefficient;

        if (double.TryParse(value.ToString(), out number) &&
double.TryParse(parameter.ToString(), out coefficient))
        {
```

```
            return number * coefficient;
        }

        return 0;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        throw new NotSupportedException();
    }
}
```

Utiliser le convertisseur

1. Définir un espace de noms

```
xmlns:converters="clr-namespace:MyProject.Converters;assembly=MyProject"
```

2. Définir une ressource

```
<converters:MultiplyConverter x:Key="MultiplyConverter"/>
```

3. Utilisez-le dans la liaison

```
<StackPanel Orientation="Vertical">
    <TextBox x:Name="TextBox" />
    <TextBlock Text="{Binding Path=Text,
                             ElementName=TextBox,
                             Converter={StaticResource MultiplyConverter},
                             ConverterParameter=10}"/>
</StackPanel>
```

Grouper plusieurs convertisseurs [IValueConverter]

Ce convertisseur enchaînera plusieurs convertisseurs.

```
public class ValueConverterGroup : List<IValueConverter>, IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        return this.Aggregate(value, (current, converter) => converter.Convert(current,
targetType, parameter, culture));
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        throw new NotSupportedException();
    }
}
```

Dans cet exemple, le résultat booléen de `EnumToBooleanConverter` est utilisé comme entrée dans `BooleanToVisibilityConverter` .

```
<local:ValueConverterGroup x:Key="EnumToVisibilityConverter">
    <local:EnumToBooleanConverter/>
    <local:BooleanToVisibilityConverter/>
</local:ValueConverterGroup>
```

Le bouton ne sera visible que lorsque la propriété `CurrentMode` est définie sur `Ready` .

```
<Button Content="Ok" Visibility="{Binding Path=CurrentMode, Converter={StaticResource
EnumToVisibilityConverter}, ConverterParameter={x:Static local:Mode.Ready}"/>
```

Utilisation de MarkupExtension avec des convertisseurs pour ignorer la déclaration de recource

Habituellement, pour utiliser le convertisseur, nous devons le définir comme ressource de la manière suivante:

```
<converters:SomeConverter x:Key="SomeConverter"/>
```

Il est possible d'ignorer cette étape en définissant un convertisseur comme `MarkupExtension` et en implémentant la méthode `ProvideValue` . L'exemple suivant convertit une valeur en son négatif:

```
namespace MyProject.Converters
{
public class Converter_Negative : MarkupExtension, IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        return this.ReturnNegative(value);
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        return this.ReturnNegative(value);
    }

    private object ReturnNegative(object value)
    {
        object result = null;
        var @switch = new Dictionary<Type, Action> {
            { typeof(bool), () => result=!(bool)value },
            { typeof(byte), () => result=-1*(byte)value },
            { typeof(short), () => result=-1*(short)value },
            { typeof(int), () => result=-1*(int)value },
            { typeof(long), () => result=-1*(long)value },
            { typeof(float), () => result=-1f*(float)value },
            { typeof(double), () => result=-1d*(double)value },
            { typeof(decimal), () => result=-1m*(decimal)value }
        };
```

```
        @switch[value.GetType()]();
        if (result == null) throw new NotImplementedException();
        return result;
    }

    public Converter_Negative()
        : base()
    {
    }

    private static Converter_Negative _converter = null;

    public override object ProvideValue(IServiceProvider serviceProvider)
    {
        if (_converter == null) _converter = new Converter_Negative();
        return _converter;
    }
}
}
```

En utilisant le convertisseur:

1. Définir un espace de noms

   xmlns: converters = "espace de noms clr: MyProject.Converters; assembly = MyProject"

2. Exemple d'utilisation de ce convertisseur en liaison

```
<RichTextBox IsReadOnly="{Binding Path=IsChecked, ElementName=toggleIsEnabled,
Converter={converters:Converter_Negative}}"/>
```

Utilisez IMultiValueConverter pour transmettre plusieurs paramètres à une commande

Il est possible de transmettre plusieurs valeurs liées en tant que `CommandParameter` aide de `MultiBinding` avec un `IMultiValueConverter` très simple:

```
namespace MyProject.Converters
{
    public class Converter_MultipleCommandParameters : MarkupExtension, IMultiValueConverter
    {
        public object Convert(object[] values, Type targetType, object parameter, CultureInfo
culture)
        {
            return values.ToArray();
        }
        public object[] ConvertBack(object value, Type[] targetTypes, object parameter,
CultureInfo culture)
        {
            throw new NotSupportedException();
        }

        private static Converter_MultipleCommandParameters _converter = null;

        public override object ProvideValue(IServiceProvider serviceProvider)
```

```
        {
            if (_converter == null) _converter = new Converter_MultipleCommandParameters();
            return _converter;
        }

        public Converter_MultipleCommandParameters()
            : base()
        {
        }
    }
}
```

En utilisant le convertisseur:

1. Exemple d'implémentation - méthode appelée lorsque `SomeCommand` est exécuté (*note: `DelegateCommand` est une implémentation de `ICommand` non fournie dans cet exemple*):

```
 private ICommand _SomeCommand;
 public ICommand SomeCommand
 {
     get { return _SomeCommand ?? (_SomeCommand = new DelegateCommand(a =>
OnSomeCommand(a))); }
 }

 private void OnSomeCommand(object item)
 {
     object[] parameters = item as object[];

     MessageBox.Show(
         string.Format("Execute command: {0}\nParameter 1: {1}\nParamter 2: {2}\nParamter
3: {3}",
         "SomeCommand", parameters[0], parameters[1], parameters[2]));
 }
```

2. Définir un espace de noms

   xmlns: converters = "espace de noms clr: MyProject.Converters; assembly = MyProject"

3. Exemple d'utilisation de ce convertisseur en liaison

```
<Button Width="150" Height="23" Content="Execute some command" Name="btnTestSomeCommand"
     Command="{Binding Path=SomeCommand}" >
     <Button.CommandParameter>
         <MultiBinding Converter="{converters:Converter_MultipleCommandParameters}">
             <Binding RelativeSource="{RelativeSource Self}" Path="IsFocused"/>
             <Binding RelativeSource="{RelativeSource Self}" Path="Name"/>
             <Binding RelativeSource="{RelativeSource Self}" Path="ActualWidth"/>
         </MultiBinding>
     </Button.CommandParameter>
 </Button>
```

Lire Convertisseurs de valeur et de valeurs multiples en ligne:
https://riptutorial.com/fr/wpf/topic/3950/convertisseurs-de-valeur-et-de-valeurs-multiples

Chapitre 7: Création d'un écran de démarrage dans WPF

Introduction

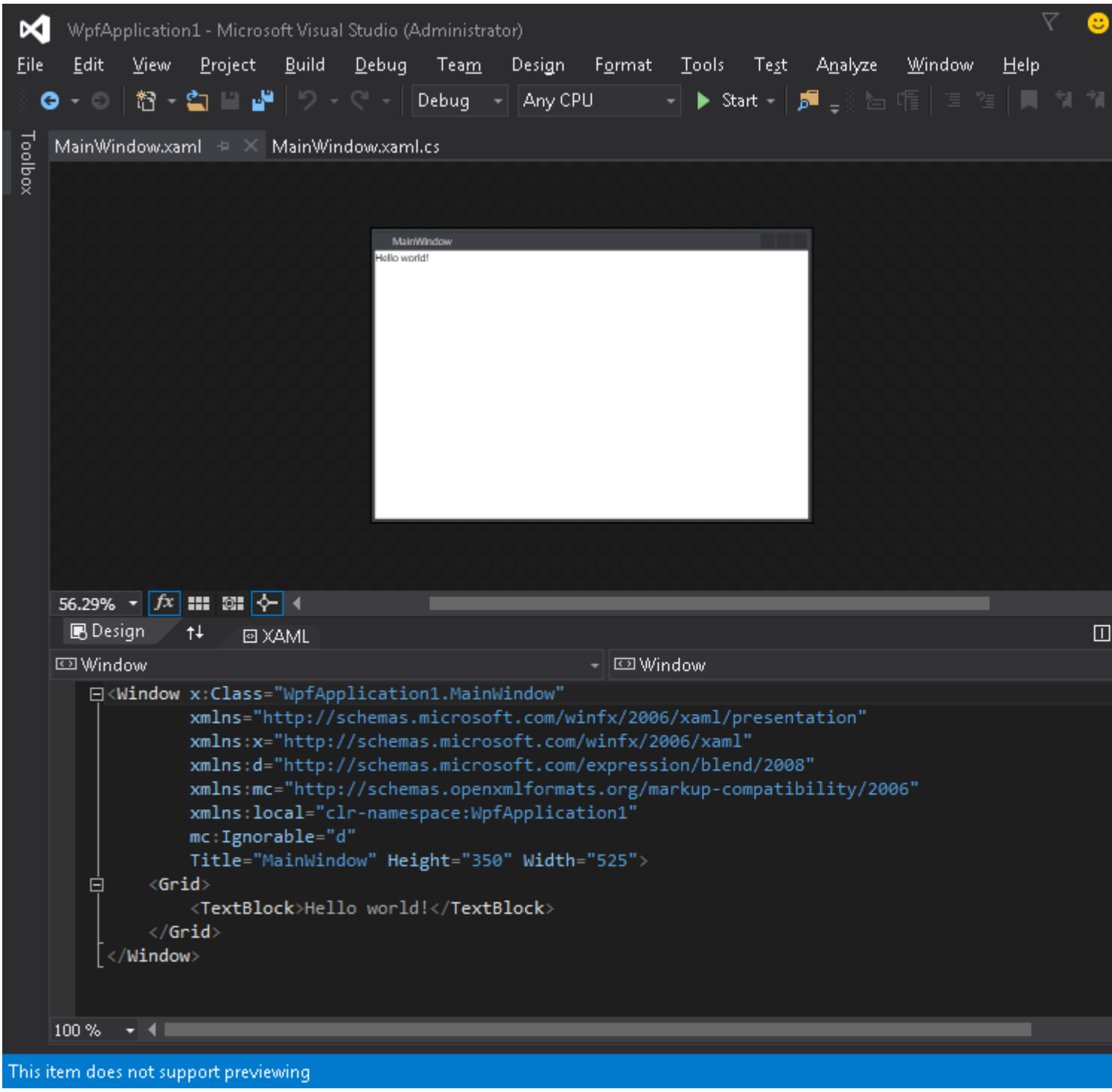
Lorsque l'application WPF est lancée, l'initialisation du langage .NET Framework peut prendre un certain temps. En conséquence, la première fenêtre de l'application peut apparaître quelque temps après le lancement de l'application, en fonction de la complexité de l'application. L'écran de démarrage dans WPF permet à l'application d'afficher une image statique ou un contenu dynamique personnalisé lors de l'initialisation avant que la première fenêtre ne s'affiche.

Examples

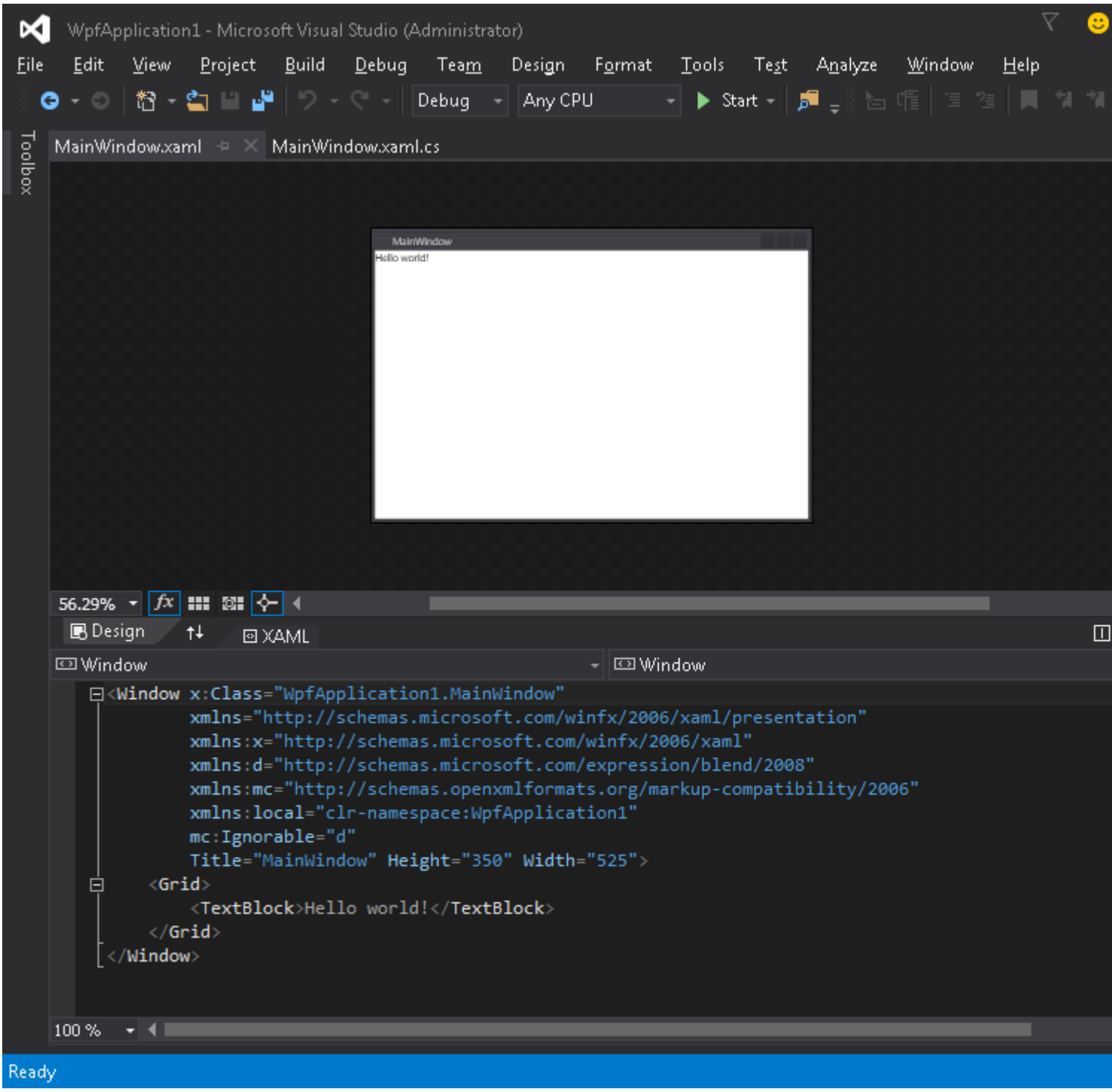
Ajout d'un écran de démarrage simple

Suivez ces étapes pour ajouter un écran de démarrage à l'application WPF dans Visual Studio:

1. Créez ou obtenez une image et ajoutez-la à votre projet (par exemple, dans le dossier *Images*):



2. Ouvrez la fenêtre des propriétés pour cette image (**Affichage → Fenêtre Propriétés**) et modifiez **le** paramètre **Action** de **génération en** valeur **SplashScreen** :



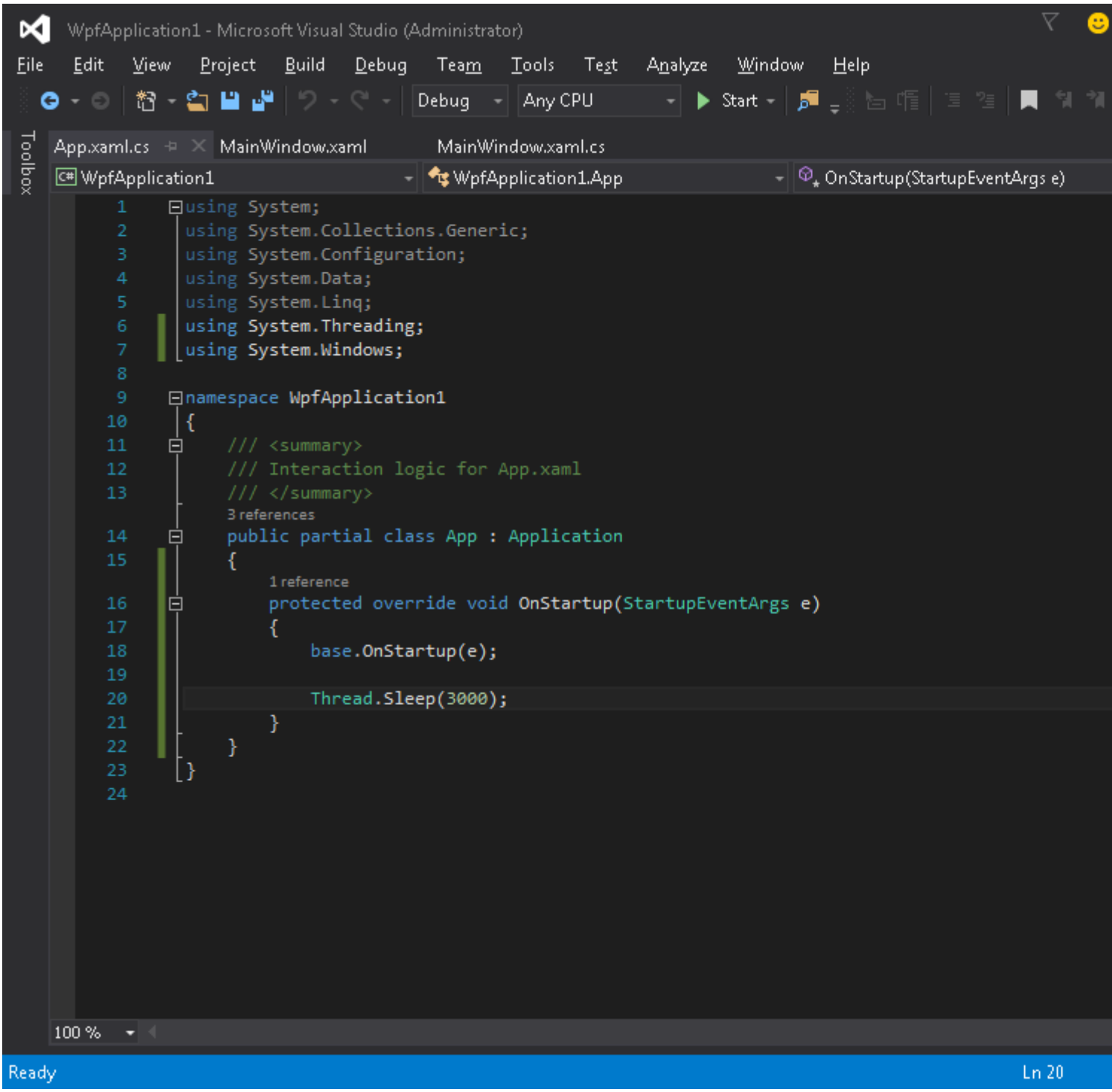
3. Exécutez l'application. Votre écran d'écran apparaît au centre de l'écran avant que la fenêtre d'application ne s'affiche (une fois la fenêtre affichée, l'image de l'écran de démarrage disparaîtra au bout de 300 millisecondes environ).

Test de l'écran de démarrage

Si votre application est légère et simple, elle se lancera très rapidement et, avec une vitesse similaire, apparaîtra et disparaîtra.

Dès que l'écran de démarrage disparaît après la méthode `Application.Startup` , vous pouvez simuler le délai de lancement de l'application en procédant comme suit:

1. Ouvrez le fichier **App.xaml.cs**
2. Ajouter en *utilisant un* espace de noms en `using System.Threading;`
3. Remplacer la méthode `OnStartup` et ajouter `Thread.Sleep(3000);` à l'intérieur:



Le code devrait ressembler à:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading;
using System.Windows;
```

```
namespace WpfApplication1
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);

            Thread.Sleep(3000);
        }
    }
}
```

4. Exécutez l'application. Maintenant, il sera lancé environ 3 secondes de plus, vous aurez donc plus de temps pour tester votre écran de démarrage.

Création d'une fenêtre d'écran de démarrage personnalisée

WPF ne prend en charge que l'affichage d'une image en tant qu'écran d'accueil, nous devons donc créer une `Window` qui servira d'écran de démarrage. Nous supposons que nous avons déjà créé un projet contenant la classe `MainWindow` , qui doit être la fenêtre principale de l'application.

Tout d'abord, nous ajoutons une fenêtre `SplashScreenWindow` à notre projet:

```
<Window x:Class="SplashScreenExample.SplashScreenWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        WindowStartupLocation="CenterScreen"
        WindowStyle="None"
        AllowsTransparency="True"
        Height="30"
        Width="200">
    <Grid>
        <ProgressBar IsIndeterminate="True" />
        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Center">Loading...</TextBlock>
    </Grid>
</Window>
```

Ensuite, nous substituons la méthode `Application.OnStartup` pour afficher l'écran de démarrage, travaillons et montrons la fenêtre principale (***App.xaml.cs***):

```
public partial class App
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);

        //initialize the splash screen and set it as the application main window
        var splashScreen = new SplashScreenWindow();
        this.MainWindow = splashScreen;
        splashScreen.Show();
```

```
        //in order to ensure the UI stays responsive, we need to
        //do the work on a different thread
        Task.Factory.StartNew(() =>
        {
            //simulate some work being done
            System.Threading.Thread.Sleep(3000);

            //since we're not on the UI thread
            //once we're done we need to use the Dispatcher
            //to create and show the main window
            this.Dispatcher.Invoke(() =>
            {
                //initialize the main window, set it as the application main window
                //and close the splash screen
                var mainWindow = new MainWindow();
                this.MainWindow = mainWindow;
                mainWindow.Show();
                splashScreen.Close();
            });
        });
    }
}
```

Enfin, nous devons nous occuper du mécanisme par défaut qui affiche `MainWindow` au démarrage de l'application. Il suffit de supprimer l' `StartupUri="MainWindow.xaml"` de la balise d' `Application` racine du fichier ***App.xaml*** .

Création d'une fenêtre d'écran de démarrage avec rapports de progression

WPF ne prend en charge que l'affichage d'une image en tant qu'écran d'accueil, nous devons donc créer une `Window` qui servira d'écran de démarrage. Nous supposons que nous avons déjà créé un projet contenant la classe `MainWindow` , qui doit être la fenêtre principale de l'application.

Tout d'abord, nous ajoutons une fenêtre `SplashScreenWindow` à notre projet:

```
<Window x:Class="SplashScreenExample.SplashScreenWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        WindowStartupLocation="CenterScreen"
        WindowStyle="None"
        AllowsTransparency="True"
        Height="30"
        Width="200">
    <Grid>
        <ProgressBar x:Name="progressBar" />
        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Center">Loading...</TextBlock>
    </Grid>
</Window>
```

Ensuite, nous exposons une propriété sur la classe `SplashScreenWindow` afin que nous puissions facilement mettre à jour la valeur de progression actuelle (***SplashScreenWindow.xaml.cs***):

```
public partial class SplashScreenWindow : Window
```

```
{
    public SplashScreenWindow()
    {
        InitializeComponent();
    }

    public double Progress
    {
        get { return progressBar.Value; }
        set { progressBar.Value = value; }
    }
}
```

Ensuite, nous substituons la méthode `Application.OnStartup` pour afficher l'écran de démarrage, travaillons et montrons la fenêtre principale (***App.xaml.cs***):

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);

        //initialize the splash screen and set it as the application main window
        var splashScreen = new SplashScreenWindow();
        this.MainWindow = splashScreen;
        splashScreen.Show();

        //in order to ensure the UI stays responsive, we need to
        //do the work on a different thread
        Task.Factory.StartNew(() =>
        {
            //we need to do the work in batches so that we can report progress
            for (int i = 1; i <= 100; i++)
            {
                //simulate a part of work being done
                System.Threading.Thread.Sleep(30);

                //because we're not on the UI thread, we need to use the Dispatcher
                //associated with the splash screen to update the progress bar
                splashScreen.Dispatcher.Invoke(() => splashScreen.Progress = i);
            }

            //once we're done we need to use the Dispatcher
            //to create and show the main window
            this.Dispatcher.Invoke(() =>
            {
                //initialize the main window, set it as the application main window
                //and close the splash screen
                var mainWindow = new MainWindow();
                this.MainWindow = mainWindow;
                mainWindow.Show();
                splashScreen.Close();
            });
        });
    }
}
```

Enfin, nous devons nous occuper du mécanisme par défaut qui affiche `MainWindow` au démarrage de l'application. Il suffit de supprimer l' `StartupUri="MainWindow.xaml"` de la balise d' `Application`

racine du fichier ***App.xaml*** .

Lire Création d'un écran de démarrage dans WPF en ligne:
https://riptutorial.com/fr/wpf/topic/3948/creation-d-un-ecran-de-demarrage-dans-wpf

Chapitre 8: Création de UserControls personnalisés avec liaison de données

Remarques

Notez qu'un UserControl est très différent d'un contrôle. L'une des principales différences est qu'un UserControl utilise un fichier de disposition XAML pour déterminer où placer plusieurs contrôles individuels. Un contrôle, par contre, est un code pur - il n'y a aucun fichier de mise en page. À certains égards, la création d'un contrôle personnalisé peut être plus efficace que la création d'un contrôle utilisateur personnalisé.

Examples

ComboBox avec un texte par défaut personnalisé

Cet UserControl personnalisé apparaîtra comme une liste déroulante standard, mais contrairement à l'objet ComboBox intégré, il peut afficher une chaîne de texte par défaut s'il n'a pas encore été sélectionné.

Pour ce faire, notre UserControl sera composé de deux contrôles. Nous avons évidemment besoin d'une ComboBox réelle, mais nous utiliserons également une étiquette régulière pour afficher le texte par défaut.

CustomComboBox.xaml

```
<UserControl x:Class="UserControlDemo.CustomComboBox"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:cnvrt="clr-namespace:UserControlDemo"
             x:Name="customComboBox">
    <UserControl.Resources>
        <cnvrt:InverseNullVisibilityConverter x:Key="invNullVisibleConverter" />
    </UserControl.Resources>
    <Grid>
        <ComboBox x:Name="comboBox"
                  ItemsSource="{Binding ElementName=customComboBox, Path=MyItemsSource}"
                  SelectedItem="{Binding ElementName=customComboBox, Path=MySelectedItem}"
                  HorizontalContentAlignment="Left" VerticalContentAlignment="Center"/>

        <Label HorizontalAlignment="Left" VerticalAlignment="Center"
               Margin="0,2,20,2" IsHitTestVisible="False"
               Content="{Binding ElementName=customComboBox, Path=DefaultText}"
               Visibility="{Binding ElementName=comboBox, Path=SelectedItem,
Converter={StaticResource invNullVisibleConverter}}"/>
    </Grid>
</UserControl>
```

Comme vous pouvez le voir, ce seul UserControl est en réalité un groupe de deux contrôles

individuels. Cela nous permet une certaine flexibilité qui n'est pas disponible dans une seule zone de liste déroulante.

Voici plusieurs points importants à noter:

- Le UserControl lui-même a un ensemble `x:Name` . Cela est dû au fait que nous souhaitons lier des propriétés situées dans le code-behind, ce qui signifie qu'il a besoin de se référencer.
- Chaque liaison de la zone de liste déroulante porte le nom de UserControl en tant que nom d' `ElementName` . Cela permet à UserControl de se regarder pour localiser les liaisons.
- L'étiquette n'est pas visible pour le test de réussite. Cela donne à l'utilisateur l'illusion que le Label fait partie de la ComboBox. En définissant `IsHitTestVisible=false` , nous interdisons à l'utilisateur de survoler ou de cliquer sur l'étiquette - toutes les entrées sont transmises à la zone de liste déroulante ci-dessous.
- Le Label utilise un convertisseur `InverseNullVisibility` pour déterminer s'il doit ou non s'afficher. Vous pouvez trouver le code pour cela au bas de cet exemple.

CustomComboBox.xaml.cs

```
public partial class CustomComboBox : UserControl
{
    public CustomComboBox()
    {
        InitializeComponent();
    }

    public static DependencyProperty DefaultTextProperty =
        DependencyProperty.Register("DefaultText", typeof(string), typeof(CustomComboBox));

    public static DependencyProperty MyItemsSourceProperty =
        DependencyProperty.Register("MyItemsSource", typeof(IEnumerable),
typeof(CustomComboBox));

    public static DependencyProperty SelectedItemProperty =
        DependencyProperty.Register("SelectedItem", typeof(object), typeof(CustomComboBox));

    public string DefaultText
    {
        get { return (string)GetValue(DefaultTextProperty); }
        set { SetValue(DefaultTextProperty, value); }
    }

    public IEnumerable MyItemsSource
    {
        get { return (IEnumerable)GetValue(MyItemsSourceProperty); }
        set { SetValue(MyItemsSourceProperty, value); }
    }

    public object MySelectedItem
    {
        get { return GetValue(MySelectedItemProperty); }
        set { SetValue(MySelectedItemProperty, value); }
    }
}
```

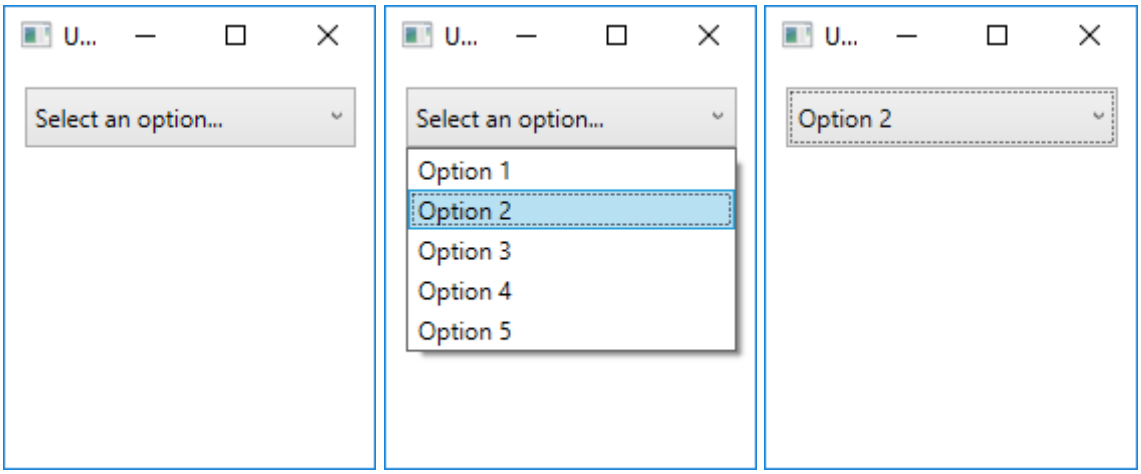
Dans le code-behind, nous exposons simplement les propriétés que nous voulons mettre à la

disposition du programmeur en utilisant ce UserControl. Malheureusement, comme nous n'avons pas d'accès direct au ComboBox en dehors de cette classe, nous devons exposer les propriétés en double (`MyItemsSource` pour `ItemsSource` de ComboBox, par exemple). Cependant, ceci est un compromis mineur étant donné que nous pouvons maintenant utiliser ceci comme un contrôle natif.

Voici comment utiliser le `CustomComboBox` UserControl:

```
<Window x:Class="UserControlDemo.UserControlDemo"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:cntrls="clr-namespace:UserControlDemo"
        Title="UserControlDemo" Height="240" Width=200>
    <Grid>
        <cntrls:CustomComboBox HorizontalAlignment="Left" Margin="10,10,0,0"
VerticalAlignment="Top" Width="165"
                               MyItemsSource="{Binding Options}"
                               MySelectedItem="{Binding SelectedOption, Mode=TwoWay}"
                               DefaultText="Select an option..."/>
    <Grid>
</Window>
```

Et le résultat final:



Voici le InverseNullVisibilityConverter nécessaire pour le Label sur UserControl, qui n'est qu'une légère variation de la version de lll :

```
public class InverseNullVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        return value == null ? Visibility.Visible : Visibility.Hidden;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        throw new NotImplementedException();
```

```
    }
}
```

Lire Création de UserControls personnalisés avec liaison de données en ligne: https://riptutorial.com/fr/wpf/topic/6508/creation-de-usercontrols-personnalises-avec-liaison-de-donnees

Chapitre 9: Déclencheurs

Introduction

Discussion sur les différents types de déclencheurs disponibles dans WPF, notamment `Trigger` , `DataTrigger` , `MultiTrigger` , `MultiDataTrigger` et `EventTrigger` .

Les déclencheurs permettent à toute classe dérivant de `FrameworkElement` ou `FrameworkContentElement` de définir ou de modifier leurs propriétés en fonction de certaines conditions définies dans le déclencheur. Fondamentalement, si un élément peut être stylé, il peut également être déclenché.

Remarques

- Tous les déclencheurs, à l'exception de `EventTrigger` doivent être définis dans un élément `<Style>` . Un `EventTrigger` peut être défini dans un élément `<Style>` ou dans la propriété `Triggers` un contrôle.
- `<Trigger>` éléments `<Trigger>` peuvent contenir un nombre quelconque d'éléments `<Setter>` . Ces éléments sont responsables de la définition des propriétés de l'élément contenant lorsque la condition de l'élément `<Trigger>` est remplie.
- Si une propriété est définie dans le balisage de l'élément racine, la modification de propriété définie dans l'élément `<Setter>` ne prendra pas effet, même si la condition de déclenchement a été remplie. Considérons le balisage `<TextBlock Text="Sample">` . La propriété `Text` du code de procédure ne changera jamais en fonction d'un déclencheur, car les définitions de la propriété racine prennent le pas sur les propriétés définies dans les styles.
- Comme les liaisons, une fois qu'un déclencheur a été utilisé, il ne peut plus être modifié.

Examples

Déclencheur

Le plus simple des cinq types de déclencheurs, le `Trigger` est responsable de la définition des propriétés en fonction d'autres propriétés **dans le même contrôle** .

```
<TextBlock>
    <TextBlock.Style>
        <Style TargetType="{x:Type TextBlock}">
            <Style.Triggers>
                <Trigger Property="Text" Value="Pass">
                    <Setter Property="Foreground" Value="Green"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </TextBlock.Style>
</TextBlock>
```

Dans cet exemple, la couleur de premier plan du `TextBlock` devient verte lorsque sa propriété `Text` est égale à la chaîne `"Pass"` .

MultiTrigger

Un `MultiTrigger` est similaire à un `Trigger` standard en ce sens qu'il s'applique uniquement aux propriétés **du même contrôle** . La différence est qu'un `MultiTrigger` a plusieurs conditions qui doivent être satisfaites avant que le déclencheur fonctionne. Les conditions sont définies à l'aide de la `<Condition>` .

```
<TextBlock x:Name="_txtBlock" IsEnabled="False">
    <TextBlock.Style>
        <Style TargetType="{x:Type TextBlock}">
            <Style.Triggers>
                <MultiTrigger>
                    <MultiTrigger.Conditions>
                        <Condition Property="Text" Value="Pass"/>
                        <Condition Property="IsEnabled" Value="True"/>
                    </MultiTrigger.Conditions>
                    <Setter Property="Foreground" Value="Green"/>
                </MultiTrigger>
            </Style.Triggers>
        </Style>
    </TextBlock.Style>
</TextBlock>
```

Notez que le `MultiTrigger` ne s'active pas tant que les deux conditions ne sont pas remplies.

DataTrigger

Un `DataTrigger` peut être attaché à n'importe quelle propriété, que ce soit sur son propre contrôle, un autre contrôle ou même une propriété dans une classe non-interface utilisateur. Considérons la classe simple suivante.

```
public class Cheese
{
    public string Name { get; set; }
    public double Age { get; set; }
    public int StinkLevel { get; set; }
}
```

Ce que nous allons attacher en tant que `DataContext` dans le `TextBlock` suivant.

```
<TextBlock Text="{Binding Name}">
    <TextBlock.DataContext>
        <local:Cheese Age="12" StinkLevel="100" Name="Limburger"/>
    </TextBlock.DataContext>
    <TextBlock.Style>
        <Style TargetType="{x:Type TextBlock}">
            <Style.Triggers>
                <DataTrigger Binding="{Binding StinkLevel}" Value="100">
                    <Setter Property="Foreground" Value="Green"/>
                </DataTrigger>
            </Style.Triggers>
```

```
        </Style>
    </TextBlock.Style>
</TextBlock>
```

Dans le code précédent, la propriété `TextBlock.Foreground` sera verte. Si nous modifions la propriété `StinkLevel` dans notre XAML à une valeur autre que 100, la propriété `Text.Foreground` reviendra à sa valeur par défaut.

Lire Déclencheurs en ligne: https://riptutorial.com/fr/wpf/topic/9624/declencheurs

Chapitre 10: Extensions de balisage

Paramètres

Méthode	La description
ProvideValue	La classe MarkupExtension ne possède qu'une méthode qui doit être remplacée, l'analyseur XAML utilise alors la valeur fournie par cette méthode pour évaluer le résultat de l'extension du balisage.

Remarques

Une extension de balisage peut être implémentée pour fournir des valeurs aux propriétés dans une utilisation d'attribut, des propriétés dans l'utilisation d'un élément de propriété ou les deux.

Lorsqu'elle est utilisée pour fournir une valeur d'attribut, la syntaxe qui distingue une séquence d'extension de balisage d'un processeur XAML est la présence des accolades ouvrantes et fermantes ({et}). Le type d'extension de balisage est ensuite identifié par le jeton de chaîne immédiatement après l'accolade d'ouverture.

Lorsqu'elle est utilisée dans une syntaxe d'élément de propriété, une extension de balisage est visuellement identique à tout autre élément utilisé pour fournir une valeur d'élément de propriété: une déclaration d'élément XAML qui référence la classe d'extension de balisage en tant qu'élément entre crochets (<>).

Pour plus d'informations, visitez https://msdn.microsoft.com/en-us/library/ms747254(v=vs.110).aspx

Examples

Extension de balisage utilisée avec IValueConverter

L'une des meilleures utilisations des extensions de balisage est une utilisation plus facile de IValueConverter. Dans l'exemple ci-dessous, BoolToVisibilityConverter est un convertisseur de valeur, mais comme il est indépendant de l'instance, il peut être utilisé sans les haches normales d'un convertisseur de valeur à l'aide de l'extension de balisage. En XAML, utilisez juste

```
Visibility="{Binding [BoolProperty], Converter={[xmlns]:BoolToVisibilityConverter}}"
```

et vous pouvez définir la visibilité de l'élément sur la valeur bool.

```
public class BoolToVisibilityConverter : MarkupExtension, IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
```

```
culture)
        {
            if (value is bool)
                return (bool)value ? Visibility.Visible : Visibility.Collapsed;
            else
                return Visibility.Collapsed;
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            if (value is Visibility)
            {
                if ((Visibility)value == Visibility.Visible)
                    return true;
                else
                    return false;
            }
            else
                return false;
        }

        public override object ProvideValue(IServiceProvider serviceProvider)
        {
            return this;
        }
    }
```

Extensions de balisage définies par XAML

Il existe quatre extensions de balisage prédéfinies dans XAML:

`x:Type` fournit l'objet Type pour le type nommé. Cette fonctionnalité est utilisée le plus fréquemment dans les styles et les modèles.

```
<object property="{x:Type prefix:typeNameValue}" .../>
```

`x:Static` produit des valeurs statiques. Les valeurs proviennent d'entités de code de type valeur qui ne sont pas directement du type de la valeur d'une propriété cible, mais peuvent être évaluées avec ce type.

```
<object property="{x:Static prefix:typeName.staticMemberName}" .../>
```

`x:Null` spécifie null comme valeur pour une propriété et peut être utilisé pour les attributs ou les valeurs d'élément de propriété.

```
<object property="{x:Null}" .../>
```

`x:Array` charge la création de tableaux généraux dans la syntaxe XAML, dans les cas où la prise en charge de la collection fournie par les éléments de base WPF et les modèles de contrôle n'est délibérément pas utilisée.

```
<x:Array Type="typeName">
```

```
  arrayContents
</x:Array>
```

Lire Extensions de balisage en ligne: https://riptutorial.com/fr/wpf/topic/6619/extensions-de-balisage

Chapitre 11: Introduction à la liaison de données WPF

Syntaxe

- {Binding PropertyName} *est équivalent à* {Binding Path = PropertyName}
- {Chemin de liaison = SomeProperty.SomeOtherProperty.YetAnotherProperty}
- {Chemin de liaison = SomeListProperty [1]}

Paramètres

Paramètre	Détails
Chemin	Spécifie le chemin d'accès à lier. Si non spécifié, se lie au DataContext lui-même.
UpdateSourceTrigger	Spécifie à quel moment la source de liaison a sa valeur mise à jour. Par défaut, `LostFocus` . La valeur la plus utilisée est `PropertyChanged` .
Mode	Généralement `OneWay` ou `TwoWay` . S'il n'est pas spécifié par la liaison, sa valeur par défaut est `OneWay` sauf si la cible de liaison le demande comme `TwoWay` . Une erreur se produit lorsque `TwoWay` est utilisé pour se lier à une propriété en lecture seule, par exemple, `OneWay` doit être défini explicitement lors de la liaison d'une propriété de chaîne en lecture seule à `TextBox.Text` .
La source	Permet d'utiliser un `StaticResource` comme source de liaison au lieu du DataContext actuel.
RelativeSource	Permet d'utiliser un autre élément XAML comme source de liaison au lieu du DataContext actuel.
ElementName	Permet d'utiliser un élément XAML nommé comme source de liaison au lieu du DataContext actuel.
Valeur de repli	Si la liaison échoue, cette valeur est fournie à la cible de liaison.
TargetNullValue	Si la valeur source de liaison est `null` , cette valeur est fournie à la cible de liaison.
Convertisseur	Spécifie le convertisseur `StaticResource` utilisé pour convertir la valeur de la liaison, par exemple, convertir un booléen en un élément enum de `Visibility` .

Paramètre	Détails
ConvertisseurParamètre	Spécifie un paramètre facultatif à fournir au convertisseur. Cette valeur doit être statique et ne peut pas être liée.
StringFormat	Spécifie une chaîne de format à utiliser lors de l'affichage de la valeur liée.
Retard	(WPF 4.5+) Spécifie un délai en `milliseconds` pendant `BindingSource` la liaison met à jour le `BindingSource` dans `ViewModel` . Cela doit être utilisé avec `Mode=TwoWay` et `UpdateSourceTrigger=PropertyChanged` pour prendre effet.

Remarques

UpdateSourceTrigger

Par défaut, WPF met à jour la source de liaison lorsque le contrôle perd le focus. Cependant, s'il n'y a qu'un seul contrôle capable d'obtenir le focus - quelque chose qui est courant dans les exemples - vous devrez spécifier `UpdateSourceTrigger=PropertyChanged` pour que les mises à jour fonctionnent.

Vous voudrez peut-être utiliser `PropertyChanged` comme déclencheur sur de nombreuses liaisons bidirectionnelles, à moins que la mise à jour de la source de liaison sur chaque frappe ne soit coûteuse ou que la validation des données en direct soit indésirable.

L'utilisation de `LostFocus` a un effet secondaire regrettable: appuyer sur Entrée pour soumettre un formulaire à l'aide d'un bouton marqué `IsDefault` ne met pas à jour la propriété de sauvegarde de votre liaison, annulant ainsi vos modifications. Heureusement, certaines solutions existent .

Notez également que, contrairement à UWP, WPF (4.5+) possède également la propriété `Delay` dans les liaisons, ce qui peut être suffisant pour certaines liaisons avec des paramètres d'intelligence mineure locale ou simple, comme certaines validations `TextBox` .

Examples

Convertir un booléen en valeur de visibilité

Cet exemple masque la zone rouge (bordure) si la case à cocher n'est pas cochée en utilisant un `IValueConverter` .

Remarque: Le `BooleanToVisibilityConverter` utilisé dans l'exemple ci-dessous est un convertisseur de valeur intégré situé dans l'espace de noms System.Windows.Controls.

XAML:

```
<Window x:Class="StackOverflowDataBindingExample.MainWindow"
```

```
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <BooleanToVisibilityConverter x:Key="VisibleIfTrueConverter" />
    </Window.Resources>
    <StackPanel>
        <CheckBox x:Name="MyCheckBox"
                  IsChecked="True" />
        <Border Background="Red" Width="20" Height="20"
                Visibility="{Binding Path=IsChecked,ElementName=MyCheckBox,
Converter={StaticResource VisibleIfTrueConverter}}" />
    </StackPanel>
</Window>
```

Définir le contexte de données

Pour pouvoir utiliser des liaisons dans WPF, vous devez définir un **DataContext** . Le DataContext indique aux liaisons où récupérer leurs données par défaut.

```
<Window x:Class="StackOverflowDataBindingExample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:StackOverflowDataBindingExample"
        xmlns:vm="clr-namespace:StackOverflowDataBindingExample.ViewModels"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.DataContext>
        <vm:HelloWorldViewModel />
    </Window.DataContext>
    ...
</Window>
```

Vous pouvez également définir le DataContext via code-behind, mais il convient de noter que XAML IntelliSense est un peu difficile: un DataContext fortement typé doit être défini dans XAML pour que IntelliSense suggère des propriétés disponibles pour la liaison.

```
/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        DataContext = new HelloWorldViewModel();
    }
}
```

Bien qu'il existe des frameworks pour vous aider à définir votre DataContext de manière plus flexible (par exemple, MVVM Light a un localisateur de vue qui utilise l' inversion du contrôle), nous utilisons la méthode rapide et sale pour les besoins de ce tutoriel.

Vous pouvez définir un DataContext pour pratiquement tous les éléments visuels de WPF. Le DataContext est généralement hérité des ancêtres de l'arborescence visuelle, sauf s'il a été explicitement remplacé, par exemple au sein d'un ContentPresenter.

Implémenter INotifyPropertyChanged

`INotifyPropertyChanged` est une interface utilisée par les sources de liaison (c.-à-d. Le DataContext) pour permettre à l'interface utilisateur ou à d'autres composants de savoir qu'une propriété a été modifiée. WPF met automatiquement à jour l'interface utilisateur lorsqu'elle détecte l'événement `PropertyChanged` déclenché. Il est souhaitable que cette interface soit implémentée sur une classe de base dont tous vos modèles de vues peuvent hériter.

En C # 6, c'est tout ce dont vous avez besoin:

```
public abstract class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void NotifyPropertyChanged([CallerMemberName] string name = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
}
```

Cela vous permet d'appeler `NotifyPropertyChanged` de deux manières différentes:

1. `NotifyPropertyChanged()` , qui `NotifyPropertyChanged()` l'événement pour le setter qui l'invoque, grâce à l'attribut CallerMemberName .
2. `NotifyPropertyChanged(nameof(SomeOtherProperty))` , qui déclenchera l'événement pour SomeOtherProperty.

Pour .NET 4.5 et versions ultérieures utilisant C # 5.0, ceci peut être utilisé à la place:

```
public abstract class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void NotifyPropertyChanged([CallerMemberName] string name = null)
    {
        var handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(name));
        }
    }
}
```

Dans les versions de .NET antérieures à la version 4.5, vous devez définir des noms de propriété sous forme de constantes de chaîne ou une solution utilisant des expressions .

Remarque: Il est possible de lier une propriété d'un "objet C # ancien" (POCO) qui

`INotifyPropertyChanged` pas `INotifyPropertyChanged` et observez que les liaisons fonctionnent mieux que prévu. Ceci est une fonctionnalité cachée dans .NET et devrait probablement être évitée. Surtout que cela entraînera des fuites de mémoire lorsque le `Mode` de liaison n'est pas `OneTime` (voir ici).

Pourquoi la liaison est-elle mise à jour sans implémenter INotifyPropertyChanged?

Lier à la propriété d'un autre élément nommé

Vous pouvez associer une propriété à un élément nommé, mais l'élément nommé doit avoir une portée.

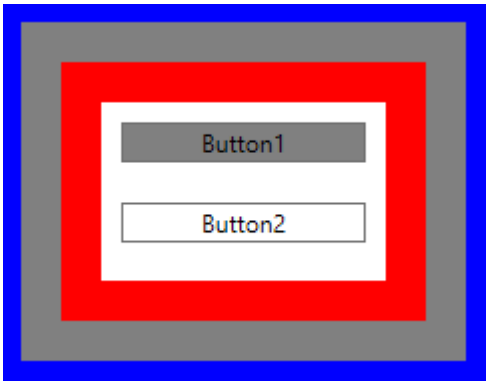
```
<StackPanel>
    <CheckBox x:Name="MyCheckBox" IsChecked="True" />
    <TextBlock Text="{Binding IsChecked, ElementName=MyCheckBox}" />
</StackPanel>
```

Lier à la propriété d'un ancêtre

Vous pouvez créer une liaison avec une propriété d'un ancêtre dans l'arborescence visuelle à l'aide d'une liaison `RelativeSource` . Le contrôle le plus proche dans l'arborescence visuelle qui a le même type ou est dérivé du type que vous spécifiez sera utilisé comme source de la liaison:

```
<Grid Background="Blue">
    <Grid Background="Gray" Margin="10">
        <Border Background="Red" Margin="20">
            <StackPanel Background="White" Margin="20">
                <Button Margin="10" Content="Button1" Background="{Binding Background,
RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type Grid}}}" />
                <Button Margin="10" Content="Button2" Background="{Binding Background,
RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type FrameworkElement}}}" />
            </StackPanel>
        </Border>
    </Grid>
</Grid>
```

Dans cet exemple, *Button1* a un arrière-plan gris car l'ancêtre de la `Grid` le plus proche a un arrière-plan gris. *Button2* a un arrière-plan blanc car l'ancêtre le plus proche dérivé de `FrameworkElement` est le `StackPanel` blanc.



Relier plusieurs valeurs avec un MultiBinding

MultiBinding permet de lier plusieurs valeurs à la même propriété. Dans l'exemple suivant, plusieurs valeurs sont liées à la propriété Text d'une zone de texte et mises en forme à l'aide de la propriété StringFormat.

```
<TextBlock>
    <TextBlock.Text>
        <MultiBinding StringFormat="{}{0} {1}">
            <Binding Path="User.Forename"/>
            <Binding Path="User.Surname"/>
        </MultiBinding>
    </TextBlock.Text>
</TextBlock>
```

En dehors de `StringFormat` , un `IMultiValueConverter` peut également être utilisé pour convertir les valeurs des liaisons en une seule valeur pour la cible du MultiBinding.

Toutefois, les MultiBindings ne peuvent pas être imbriqués.

Lire Introduction à la liaison de données WPF en ligne:
https://riptutorial.com/fr/wpf/topic/2236/introduction-a-la-liaison-de-donnees-wpf

Chapitre 12: Localisation WPF

Remarques

Le contenu des contrôles peut être localisé à l'aide de fichiers de ressources, comme cela est possible dans les classes. Pour XAML, il existe une syntaxe spécifique, différente entre une application C # et une application VB.

Les étapes sont les suivantes:

- Pour tout projet WPF: rendre le fichier de ressources public, la valeur par défaut est interne.
- Pour les projets WPF C #, utilisez le XAML fourni dans l'exemple
- Pour VB, les projets WPF utilisent le XAML fourni dans l'exemple et modifient la propriété Outil personnalisé en `PublicVbMyResourcesResXFileCodeGenerator` .
- Pour sélectionner le fichier Resources.resx dans un projet WPF VB:
    - Sélectionnez le projet dans l'explorateur de solutions
    - Sélectionnez "Afficher tous les fichiers"
    - Développez mon projet

Examples

XAML pour VB

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfApplication1"
    xmlns:my="clr-namespace:WpfApplication1.My.Resources"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
<Grid>
    <StackPanel>
        <Label Content="{Binding Source={x:Static my:Resources.MainWindow_Label_Country}}" />
    </StackPanel>
</Grid>
```

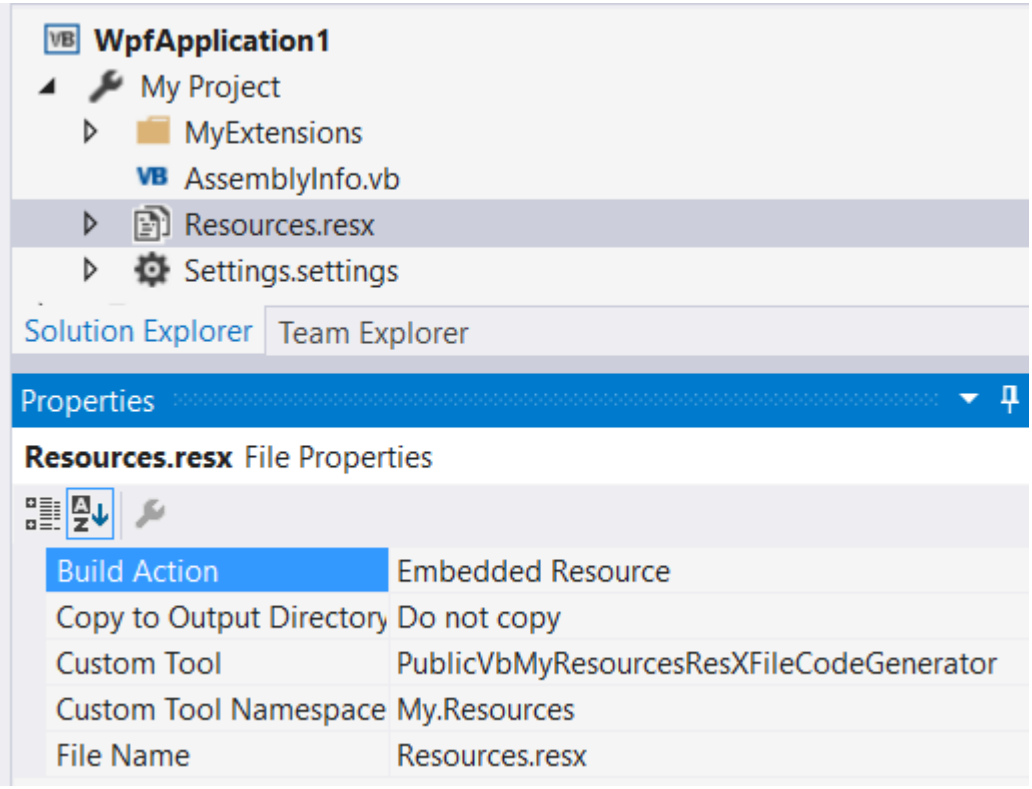
Propriétés du fichier de ressources dans VB

Par défaut, la propriété Outil personnalisé pour un fichier de ressources VB est `VbMyResourcesResXFileCodeGenerator` . Cependant, avec ce générateur de code, la vue (XAML) ne pourra pas accéder aux ressources. Pour résoudre ce problème, ajoutez `Public` avant la valeur de la propriété Outil personnalisé.

Pour sélectionner le fichier Resources.resx dans un projet WPF VB:

- Sélectionnez le projet dans l'explorateur de solutions

- Sélectionnez "Afficher tous les fichiers"
- Développez "Mon projet"

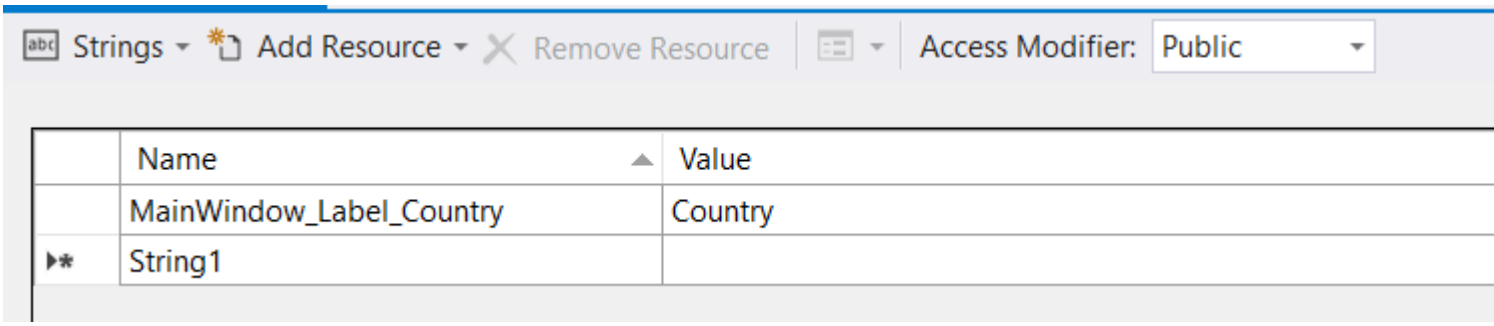


XAML pour C

```
<Window x:Class="WpfApplication2.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfApplication2"
    xmlns:resx="clr-namespace:WpfApplication2.Properties"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
<Grid>
    <StackPanel>
        <Label Content="{Binding Source={x:Static resx:Resources.MainWindow_Label_Country}}"/>
    </StackPanel>
</Grid>
```

Rendre les ressources publiques

Ouvrez le fichier de ressources en double-cliquant dessus. Modifiez le modificateur d'accès à "Public".



Lire Localisation WPF en ligne: https://riptutorial.com/fr/wpf/topic/3905/localisation-wpf

Chapitre 13: MVVM dans WPF

Remarques

Modèles et modèles de vue

La définition d'un modèle est souvent controversée et la ligne entre un modèle et un modèle de vue peut être floue. Certains préfèrent ne pas "polluer" leurs modèles avec l'interface `INotifyPropertyChanged` et dupliquer les propriétés du modèle dans le modèle de vue, ce *qui* implémente cette interface. Comme beaucoup de choses dans le développement de logiciels, il n'y a pas de bonne ou de mauvaise réponse. Soyez pragmatique et faites tout ce qui vous convient.

Voir la séparation

L'intention de MVVM est de séparer ces trois zones distinctes: modèle, modèle de vue et vue. Bien qu'il soit acceptable que la vue accède au modèle de vue (VM) et (indirectement) au modèle, la règle la plus importante avec MVVM est que la VM ne doit pas avoir accès à la vue ou à ses contrôles. La VM doit exposer tout ce dont la vue a besoin, via des propriétés publiques. La VM ne doit pas exposer ou manipuler directement les contrôles de l'interface utilisateur tels que `TextBox` , `Button` , etc.

Dans certains cas, cette séparation stricte peut être difficile à utiliser, en particulier si vous avez besoin de fonctionnalités d'interface utilisateur complexes. Ici, il est parfaitement acceptable d'utiliser des événements et des gestionnaires d'événements dans le fichier "code-behind" de la vue. Si c'est uniquement une fonctionnalité d'interface utilisateur, utilisez tous les événements de la vue. Il est également acceptable que ces gestionnaires d'événements appellent des méthodes publiques sur l'instance de VM - ne passez pas simplement les références aux contrôles de l'interface utilisateur ou quelque chose du genre.

RelayCommand

Malheureusement, la classe `RelayCommand` utilisée dans cet exemple ne fait pas partie du framework WPF (cela aurait dû être le cas!), Mais vous la trouverez dans presque toutes les boîtes à outils du développeur WPF. Une recherche rapide en ligne révélera de nombreux extraits de code que vous pouvez soulever pour créer les vôtres.

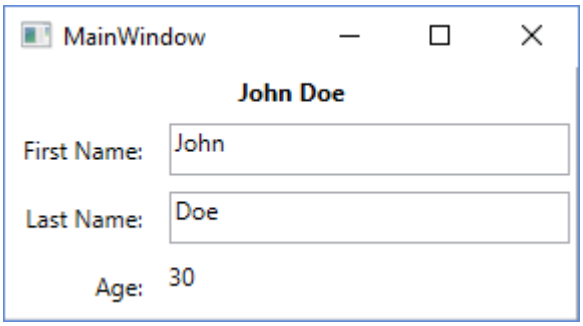
`ActionCommand` est une alternative utile à `RelayCommand` est fournie avec `Microsoft.Expression.Interactivity.Core` et fournit des fonctionnalités comparables.

Examples

Exemple MVVM de base utilisant WPF et C

Ceci est un exemple de base pour l'utilisation du modèle MVVM dans une application de bureau Windows, en utilisant WPF et C #. L'exemple de code implémente une simple boîte de dialogue

"info utilisateur".



La vue

Le XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
         <RowDefinition Height="Auto"/>
         <RowDefinition Height="Auto"/>
     </Grid.RowDefinitions>

    <TextBlock Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="2" Margin="4" Text="{Binding
FullName}" HorizontalAlignment="Center" FontWeight="Bold"/>

    <Label Grid.Column="0" Grid.Row="1" Margin="4" Content="First Name:"
HorizontalAlignment="Right"/>
    <!-- UpdateSourceTrigger=PropertyChanged makes sure that changes in the TextBoxes are
immediately applied to the model. -->
    <TextBox Grid.Column="1" Grid.Row="1" Margin="4" Text="{Binding FirstName,
UpdateSourceTrigger=PropertyChanged}" HorizontalAlignment="Left" Width="200"/>

    <Label Grid.Column="0" Grid.Row="2" Margin="4" Content="Last Name:"
HorizontalAlignment="Right"/>
    <TextBox Grid.Column="1" Grid.Row="2" Margin="4" Text="{Binding LastName,
UpdateSourceTrigger=PropertyChanged}" HorizontalAlignment="Left"  Width="200"/>

    <Label Grid.Column="0" Grid.Row="3" Margin="4" Content="Age:"
HorizontalAlignment="Right"/>
    <TextBlock Grid.Column="1" Grid.Row="3" Margin="4" Text="{Binding Age}"
HorizontalAlignment="Left"/>

</Grid>
```

et le code derrière

```
public partial class MainWindow : Window
{
    private readonly MyViewModel _viewModel;

    public MainWindow() {
        InitializeComponent();
```

```
        _viewModel = new MyViewModel();
        // The DataContext serves as the starting point of Binding Paths
        DataContext = _viewModel;
    }
}
```

Le modèle de vue

```
// INotifyPropertyChanged notifies the View of property changes, so that Bindings are updated.
sealed class MyViewModel : INotifyPropertyChanged
{
    private User user;

    public string FirstName {
        get {return user.FirstName;}
        set {
            if(user.FirstName != value) {
                user.FirstName = value;
                OnPropertyChange("FirstName");
                // If the first name has changed, the FullName property needs to be udpated as
well.
                OnPropertyChange("FullName");
            }
        }
    }

    public string LastName {
        get { return user.LastName; }
        set {
            if (user.LastName != value) {
                user.LastName = value;
                OnPropertyChange("LastName");
                // If the first name has changed, the FullName property needs to be udpated as
well.
                OnPropertyChange("FullName");
            }
        }
    }

    // This property is an example of how model properties can be presented differently to the
View.
    // In this case, we transform the birth date to the user's age, which is read only.
    public int Age {
        get {
            DateTime today = DateTime.Today;
            int age = today.Year - user.BirthDate.Year;
            if (user.BirthDate > today.AddYears(-age)) age--;
            return age;
        }
    }

    // This property is just for display purposes and is a composition of existing data.
    public string FullName {
        get { return FirstName + " " + LastName; }
    }

    public MyViewModel() {
        user = new User {
            FirstName = "John",
            LastName = "Doe",
```

```
            BirthDate = DateTime.Now.AddYears(-30)
        };
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChange(string propertyName) {
        if(PropertyChanged != null) {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Le modèle

```
sealed class User
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public DateTime BirthDate { get; set; }
}
```

Le modèle de vue

Le modèle de vue est la "VM" dans MV **VM** . Cette classe sert d'intermédiaire, expose le (s) modèle (s) à l'interface utilisateur (vue) et gère les requêtes de la vue, telles que les commandes déclenchées par des clics de bouton. Voici un modèle de vue de base:

```
public class CustomerEditViewModel
{
    /// <summary>
    /// The customer to edit.
    /// </summary>
    public Customer CustomerToEdit { get; set; }

    /// <summary>
    /// The "apply changes" command
    /// </summary>
    public ICommand ApplyChangesCommand { get; private set; }

    /// <summary>
    /// Constructor
    /// </summary>
    public CustomerEditViewModel()
    {
        CustomerToEdit = new Customer
                         {
                             Forename = "John",
                             Surname = "Smith"
                         };

        ApplyChangesCommand = new RelayCommand(
            o => ExecuteApplyChangesCommand(),
            o => CustomerToEdit.IsValid);
    }
```

```
    /// <summary>
    /// Executes the "apply changes" command.
    /// </summary>
    private void ExecuteApplyChangesCommand()
    {
        // E.g. save your customer to database
    }
}
```

Le constructeur crée un objet de modèle `Customer` et l'affecte à la propriété `CustomerToEdit` , de sorte qu'il soit visible pour la vue.

Le constructeur crée également un objet `RelayCommand` et l'affecte à la propriété `ApplyChangesCommand` , la rendant à nouveau visible à la vue. Les commandes WPF sont utilisées pour gérer les requêtes de la vue, telles que les clics sur les boutons ou les éléments de menu.

`RelayCommand` prend deux paramètres - le premier est le délégué qui est appelé lorsque la commande est exécutée (par exemple en réponse à un clic sur un bouton). Le second paramètre est un délégué qui renvoie une valeur booléenne indiquant si la commande peut être exécutée; Dans cet exemple, il est connecté à la propriété `IsValid` l'objet client. Lorsque ceci renvoie false, le bouton ou l'élément de menu lié à cette commande est désactivé (les autres contrôles peuvent se comporter différemment). Ceci est une fonctionnalité simple mais efficace, évitant d'avoir à écrire du code pour activer ou désactiver les contrôles en fonction de différentes conditions.

Si vous obtenez cet exemple opérationnel, essayez de vider un des `TextBox` es (pour placer le modèle du `Customer` dans un état non valide). Lorsque vous vous éloignez de la zone de `TextBox` vous devriez constater que le bouton "Appliquer" est désactivé.

Remarque sur la création de client

Le modèle de vue `INotifyPropertyChanged` pas `INotifyPropertyChanged` (INPC). Cela signifie que si un autre objet `Customer` devait être affecté à la propriété `CustomerToEdit` , les contrôles de la vue ne changeraient pas pour refléter le nouvel objet - les `TextBox` es contiendraient toujours le prénom et le nom de famille du client précédent.

L'exemple de code fonctionne car le `Customer` est créé dans le constructeur du modèle de vue, avant d'être affecté au `DataContext` la vue (à quel point les liaisons sont câblées). Dans une application du monde réel, vous pouvez récupérer des clients à partir d'une base de données dans des méthodes autres que le constructeur. Pour prendre cela en charge, la machine virtuelle doit implémenter INPC et la propriété `CustomerToEdit` doit être modifiée pour utiliser le modèle getter et setter "étendu" que vous voyez dans l'exemple de code Model, en levant l'événement `PropertyChanged` dans le setter.

La `ApplyChangesCommand` du modèle de `ApplyChangesCommand` n'a pas besoin d'implémenter INPC car il est très peu probable que la commande change. Vous *auriez* besoin de mettre en œuvre ce modèle si vous créez la commande quelque part autre que le constructeur, par exemple une sorte de `Initialize()` méthode.

La règle générale est la suivante: implémenter INPC si la propriété est liée à des contrôles de vue

et que la valeur de la propriété peut changer ailleurs que dans le constructeur. Vous n'avez pas besoin d'implémenter INPC si la valeur de la propriété n'est affectée que dans le constructeur (et vous vous épargnerez de la saisie dans le processus).

Le modèle

Le modèle est le premier "M" de **M** VVM. Le modèle est généralement une classe contenant les données que vous souhaitez exposer via une interface utilisateur.

Voici une classe de modèle très simple exposant quelques propriétés: -

```
public class Customer : INotifyPropertyChanged
{
    private string _forename;
    private string _surname;
    private bool _isValid;

    public event PropertyChangedEventHandler PropertyChanged;

    /// <summary>
    /// Customer forename.
    /// </summary>
    public string Forename
    {
        get
        {
            return _forename;
        }
        set
        {
            if (_forename != value)
            {
                _forename = value;
                OnPropertyChanged();
                SetIsValid();
            }
        }
    }

    /// <summary>
    /// Customer surname.
    /// </summary>
    public string Surname
    {
        get
        {
            return _surname;
        }
        set
        {
            if (_surname != value)
            {
                _surname = value;
                OnPropertyChanged();
                SetIsValid();
            }
        }
    }
```

```
    /// <summary>
    /// Indicates whether the model is in a valid state or not.
    /// </summary>
    public bool IsValid
    {
        get
        {
            return _isValid;
        }
        set
        {
            if (_isValid != value)
            {
                _isValid = value;
                OnPropertyChanged();
            }
        }
    }

    /// <summary>
    /// Sets the value of the IsValid property.
    /// </summary>
    private void SetIsValid()
    {
        IsValid = !string.IsNullOrEmpty(Forename) && !string.IsNullOrEmpty(Surname);
    }

    /// <summary>
    /// Raises the PropertyChanged event.
    /// </summary>
    /// <param name="propertyName">Name of the property.</param>
    private void OnPropertyChanged([CallerMemberName] string propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Cette classe implémente l'interface `INotifyPropertyChanged` qui expose un événement `PropertyChanged` . Cet événement doit être déclenché chaque fois qu'une des valeurs de propriété change - vous pouvez le voir dans le code ci-dessus. L'événement `PropertyChanged` est un élément clé des mécanismes de liaison de données WPF, sans lequel l'interface utilisateur ne serait pas en mesure de refléter les modifications apportées à la valeur d'une propriété.

Le modèle contient également une routine de validation très simple qui est appelée par les configurateurs de propriétés. Il définit une propriété publique indiquant si le modèle est ou non dans un état valide. J'ai inclus cette fonctionnalité pour démontrer une fonctionnalité "spéciale" des *commandes* WPF, que vous verrez bientôt. *L'infrastructure WPF fournit un certain nombre d'approches plus sophistiquées à la validation, mais celles-ci ne relèvent pas de cet article* .

La vue

La vue est le "V" dans M **V** VM. Ceci est votre interface utilisateur. Vous pouvez utiliser le concepteur de glisser-déposer de Visual Studio, mais la plupart des développeurs finissent par coder le fichier XAML brut - une expérience similaire à celle du langage HTML.

Voici le XAML d'une vue simple pour permettre l'édition d'un modèle de `Customer` . Plutôt que de créer une nouvelle vue, vous pouvez simplement la coller dans le fichier `MainWindow.xaml` un projet WPF, entre les balises `<Window ...>` et `</Window>` : -

```
<StackPanel Orientation="Vertical"
            VerticalAlignment="Top"
            Margin="20">
    <Label Content="Forename"/>
    <TextBox Text="{Binding CustomerToEdit.Forename}"/>

    <Label Content="Surname"/>
    <TextBox Text="{Binding CustomerToEdit.Surname}"/>

    <Button Content="Apply Changes"
            Command="{Binding ApplyChangesCommand}" />
</StackPanel>
```

Ce code crée un formulaire de saisie de données simple composé de deux `TextBox` : une pour le nom du client et une pour le nom de famille. Il y a une `Label` au-dessus de chaque zone de `TextBox` et un `Button` "Appliquer" au bas du formulaire.

Localisez le premier `TextBox` et regardez sa propriété `Text` :

```
Text="{Binding CustomerToEdit.Forename}"
```

Plutôt que de définir le texte du `TextBox` sur une valeur fixe, cette syntaxe d'accolade spéciale lie à la place le texte au "chemin" `CustomerToEdit.Forename` . Quel est ce chemin par rapport? C'est le "contexte de données" de la vue - dans ce cas, notre modèle de vue. Le chemin de liaison, comme vous pourrez peut-être le comprendre, est la propriété `CustomerToEdit` du modèle de vue, de type `Customer` qui à son tour expose une propriété appelée `Forename` - d'où la notation de chemin "en pointillés".

De même, si vous examinez le XAML du `Button` , il comporte une `Command` liée à la propriété `ApplyChangesCommand` du `ApplyChangesCommand` de vue. C'est tout ce qui est nécessaire pour connecter un bouton à la commande de la VM.

Le DataContext

Alors, comment définir le modèle de vue comme étant le contexte de données de la vue? Une façon consiste à le définir dans le code-behind de la vue. Appuyez sur F7 pour afficher ce fichier de code et ajoutez une ligne au constructeur existant pour créer une instance du modèle de vue et l'attribuer à la propriété `DataContext` la fenêtre. Il devrait finir par ressembler à ceci:

```
    public MainWindow()
    {
        InitializeComponent();

        // Our new line:-
        DataContext = new CustomerEditViewModel();
    }
```

Dans les systèmes du monde réel, d'autres approches sont souvent utilisées pour créer le modèle de vue, tel que l'injection de dépendances ou les frameworks MVVM.

Commandes dans MVVM

Les commandes sont utilisées pour gérer les `Events` dans WPF tout en respectant le modèle MVVM.

Un `EventHandler` normal ressemblerait à ceci (situé dans `Code-Behind`):

```
public MainWindow()
{
    _dataGrid.CollectionChanged += DataGrid_CollectionChanged;
}

private void DataGrid_CollectionChanged(object sender,
System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
        //Do what ever
}
```

Non, pour faire la même chose dans MVVM, nous utilisons des `Commands` :

```
 <Button Command="{Binding Path=CmdStartExecution}" Content="Start" />
```

> Je recommande d'utiliser une sorte de préfixe (`Cmd`) pour vos propriétés de commande, car vous en aurez principalement besoin dans xaml - de cette façon, elles seront plus faciles à reconnaître.

Comme il s'agit de MVVM, vous voulez gérer cette commande (pour le `Button` "eq" `Button_Click`) dans votre `ViewModel` .

Pour cela, nous avons essentiellement besoin de deux choses:

1. System.Windows.Input.ICommand
2. RelayCommand (par exemple pris ici .

Un **exemple** simple pourrait ressembler à ceci:

```
private RelayCommand _commandStart;
public ICommand CmdStartExecution
{
   get
   {
      if(_commandStart == null)
      {
          _commandStart = new RelayCommand(param => Start(), param => CanStart());
      }
      return _commandStart;
   }
}

public void Start()
```

```
{
   //Do what ever
}

public bool CanStart()
{
    return (DateTime.Now.DayOfWeek == DayOfWeek.Monday); //Can only click that button on
mondays.
}
```

Alors qu'est-ce que cela fait en détail:

`ICommand` est ce à quoi le `Control` dans xaml est lié. `RelayCommand` achemine votre commande vers une `Action` (c'est-à-dire, appelle une `Method`). Le Null-Check garantit simplement que chaque `Command` ne sera initialisée qu'une fois (en raison de problèmes de performance). Si vous avez lu le lien ci-dessus pour `RelayCommand` vous avez peut-être remarqué que `RelayCommand` a deux surcharges pour son constructeur. `(Action<object> execute)` et `(Action<object> execute, Predicate<object> canExecute)` .

Cela signifie que vous pouvez (en plus) ajouter une seconde `Method` renvoyant un `bool` pour indiquer que le `Control` peut être déclenché ou non.

Une bonne chose à cela est que `Button` s par exemple sera `Enabled="false"` si la `Method` retournera `false`

Paramètres de commande

```
<DataGrid x:Name="TicketsDataGrid">
    <DataGrid.InputBindings>
        <MouseBinding Gesture="LeftDoubleClick"
                      Command="{Binding CmdTicketClick}"
                      CommandParameter="{Binding ElementName=TicketsDataGrid,
                                                 Path=SelectedItem}" />
    </DataGrid.InputBindings>
<DataGrid />
```

Dans cet exemple, je souhaite transmettre `DataGrid.SelectedItem` à Click_Command dans mon ViewModel.

Votre méthode devrait ressembler à ceci alors que l'implémentation ICommand reste elle-même comme ci-dessus.

```
private RelayCommand _commandTicketClick;

public ICommand CmdTicketClick
{
   get
   {
       if(_commandTicketClick == null)
       {
           _commandTicketClick = new RelayCommand(param => HandleUserClick(param));
       }
       return _commandTicketClick;
   }
```

```
}

private void HandleUserClick(object item)
{
    MyModelClass selectedItem = item as MyModelClass;
    if (selectedItem != null)
    {
        //Do sth. with that item
    }
}
```

Lire MVVM dans WPF en ligne: https://riptutorial.com/fr/wpf/topic/2134/mvvm-dans-wpf

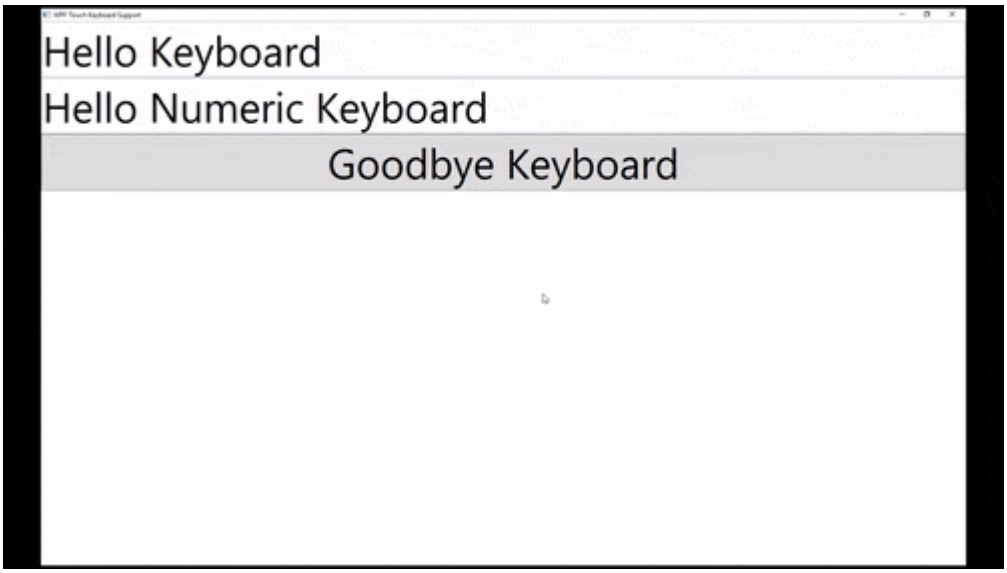
Chapitre 14: Optimisation pour l'interaction tactile

Examples

Affichage du clavier tactile sous Windows 8 et Windows 10

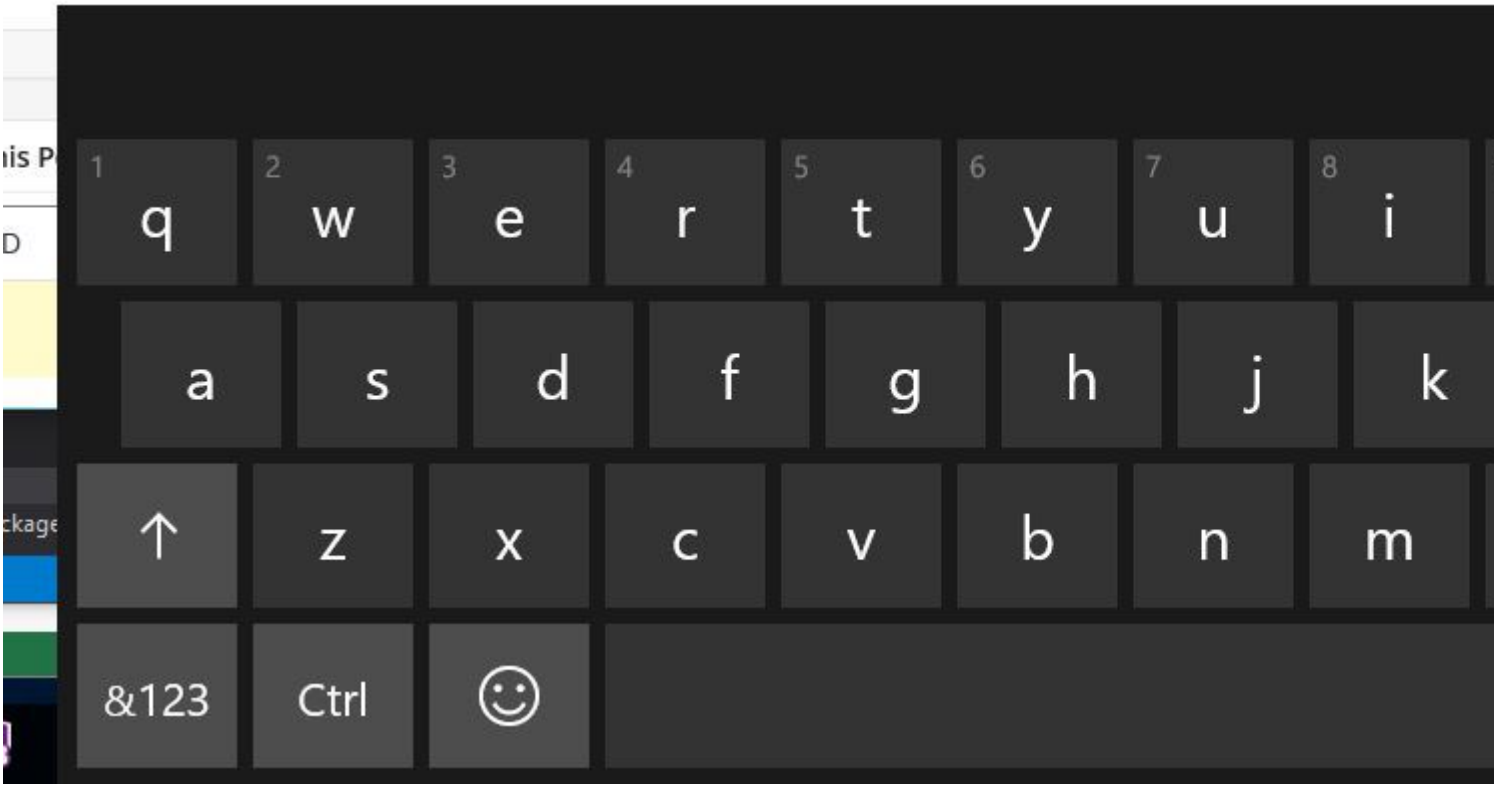
Applications WPF ciblant .NET Framework 4.6.2 et versions ultérieures

Avec les applications WPF ciblant .NET Framework 4.6.2 (et versions ultérieures), le clavier logiciel est automatiquement appelé et supprimé sans aucune étape supplémentaire.



Applications WPF ciblant .NET Framework 4.6.1 et versions antérieures

WPF n'est pas principalement activé par le toucher, ce qui signifie que lorsque l'utilisateur interagit avec une application WPF sur le bureau, l'application **n'affichera pas automatiquement le clavier tactile** lorsque les contrôles `TextBox` recevront le focus. Il s’agit d’un comportement peu pratique pour les utilisateurs de tablettes, les obligeant à ouvrir manuellement le clavier tactile via la barre des tâches du système.



solution de contournement

Le clavier tactile est en fait une application `exe` classique qui peut être trouvée sur chaque ordinateur Windows 8 et Windows 10 sur le chemin suivant: `C:\Program Files\Common Files\Microsoft Shared\Ink\TabTip.exe` .

Sur la base de ces connaissances, vous pouvez créer un contrôle personnalisé dérivé de `TextBox` , qui écoute l'événement `GotTouchCapture` (cet événement est appelé lorsque le contrôle obtient le focus à l'aide du toucher) et **démarre le processus du clavier tactile** .

```
public class TouchEnabledTextBox : TextBox
{
    public TouchEnabledTextBox()
    {
        this.GotTouchCapture += TouchEnabledTextBox_GotTouchCapture;
    }

    private void TouchEnabledTextBox_GotTouchCapture(
       object sender,
       System.Windows.Input.TouchEventArgs e )
    {
        string touchKeyboardPath =
           @"C:\Program Files\Common Files\Microsoft Shared\Ink\TabTip.exe";
        Process.Start( touchKeyboardPath );
    }
}
```

Vous pouvez encore améliorer cela en mettant en cache le processus créé et en le tuant après que le contrôle ait perdu le focus:

```
//added field
private Process _touchKeyboardProcess = null;

//replace Process.Start line from the previous listing with
_touchKeyboardProcess = Process.Start( touchKeyboardPath );
```

Vous pouvez maintenant `LostFocus` événement `LostFocus` :

```
//add this at the end of TouchEnabledTextBox's constructor
this.LostFocus += TouchEnabledTextBox_LostFocus;

//add this method as a member method of the class
private void TouchEnabledTextBox_LostFocus( object sender, RoutedEventArgs eventArgs ){
   if ( _touchKeyboardProcess != null )
   {
      _touchKeyboardProcess.Kill();
      //nullify the instance pointing to the now-invalid process
      _touchKeyboardProcess = null;
   }
}
```

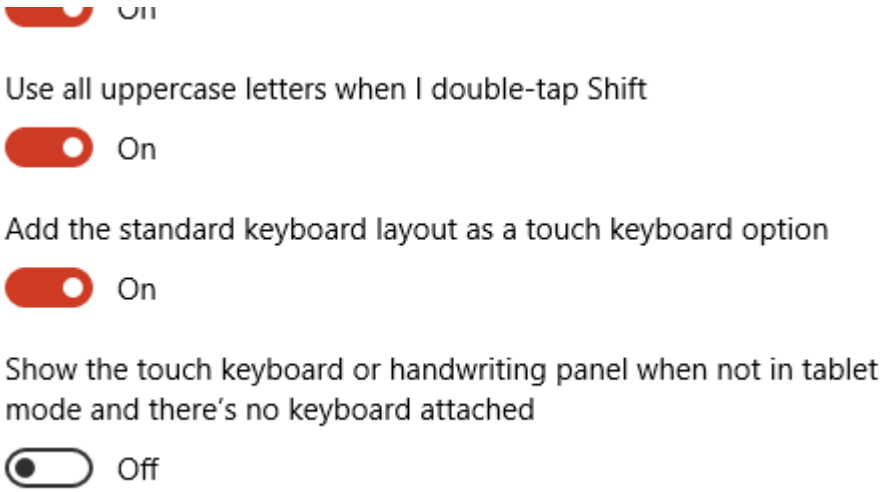
Remarque sur le mode tablette dans Windows 10

Windows 10 a introduit un **mode tablette** , ce qui simplifie l'interaction avec le système lors de l'utilisation du PC en mode tactile. Outre les autres améliorations, ce mode garantit que le **clavier tactile est automatiquement affiché,** même pour les applications de bureau classiques, y compris les applications WPF.

Approche des paramètres Windows 10

Outre le mode tablette, Windows 10 peut afficher automatiquement le clavier tactile pour les applications classiques, même en dehors du mode tablette. Ce comportement, qui est désactivé par défaut, peut être activé dans l'application Paramètres.

Dans l'application **Paramètres** , accédez à la catégorie **Périphériques** et sélectionnez **Saisie** . Si vous faites défiler complètement, vous pouvez sélectionner le paramètre "Afficher le clavier tactile ou le panneau d'écriture lorsque le mode n'est pas en mode tablette et qu'il n'y a pas de clavier connecté", que vous pouvez activer.



Il est à noter que ce paramètre est uniquement visible sur les appareils dotés de fonctions tactiles.

Lire Optimisation pour l'interaction tactile en ligne:
https://riptutorial.com/fr/wpf/topic/6799/optimisation-pour-l-interaction-tactile

Chapitre 15: Principe de conception "Half the Whitespace"

Introduction

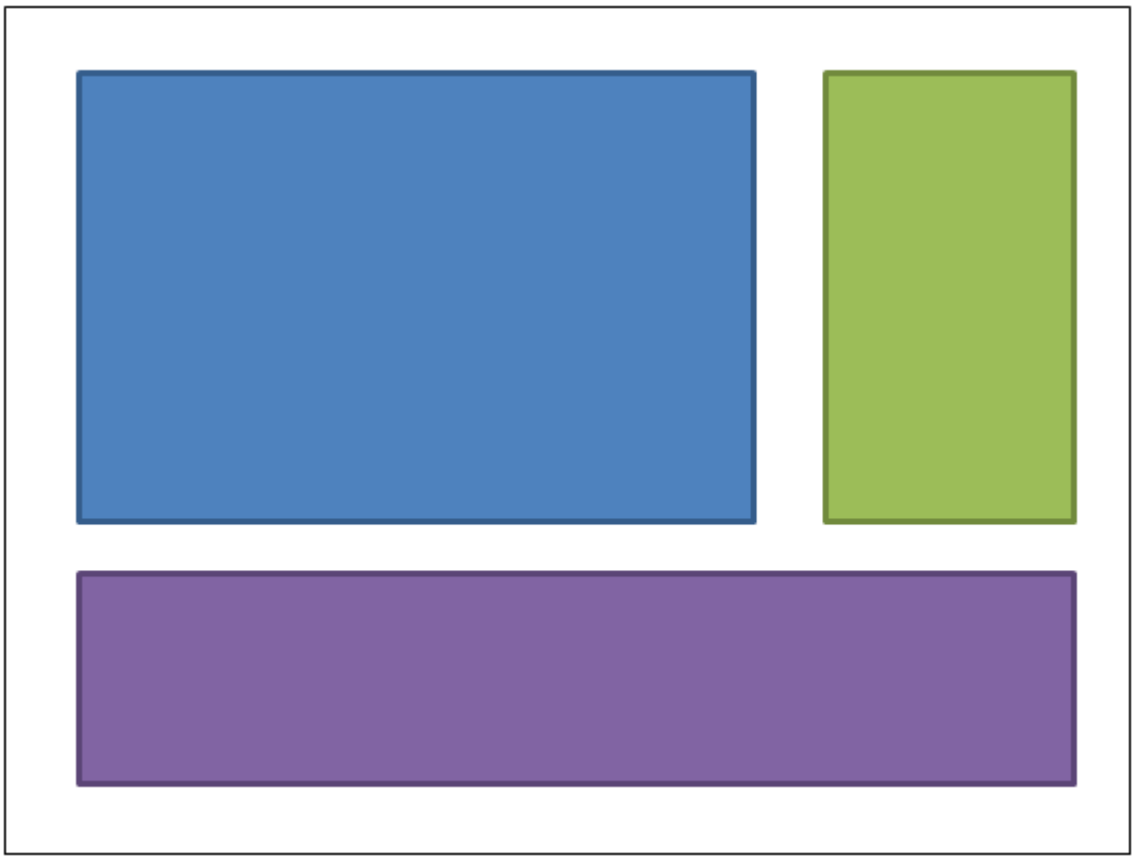
Lors de la mise en place des contrôles, il est facile de coder en dur des valeurs spécifiques dans les marges et les rembourrages afin de les adapter à la mise en page souhaitée. Cependant, en codant en dur ces valeurs, la maintenance devient beaucoup plus coûteuse. Si la disposition change, dans ce qui peut être considéré comme une manière triviale, alors beaucoup de travail doit être fait pour corriger ces valeurs.

Ce principe de conception réduit les coûts de maintenance de la mise en page en pensant à la disposition différemment.

Examples

Démonstration du problème et de la solution

Par exemple, imaginez un écran avec 3 sections, disposées comme ceci:



La boîte bleue pourrait recevoir une marge de 4,4,0,0. La boîte verte pourrait recevoir une marge de 4,4,4,0. La marge de la boîte mauve serait 4,4,4,4. Voici le XAML: (j'utilise une grille pour

réaliser la mise en page, mais ce principe s'applique quelle que soit la méthode choisie pour réaliser la mise en page):

```
<UserControl x:Class="WpfApplication5.UserControl1HardCoded"
             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">
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="3*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Border Grid.Column="0" Grid.Row="0" Margin="4,4,0,0" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
    <Border Grid.Column="1" Grid.Row="0" Margin="4,4,4,0" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
    <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Margin="4,4,4,4"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
</Grid>
</UserControl>
```

Maintenant, imaginons que nous voulons modifier la disposition, pour mettre la boîte verte à gauche de la boîte bleue. Devrait être simple, n'est-ce pas? Sauf que quand on déplace cette boîte, il faut maintenant bricoler les marges. Soit nous pouvons changer les marges de la boîte bleue à 0,4,4,0; ou nous pourrions changer en bleu à 4,4,4,0 et vert à 4,4,0,0. Voici le XAML:

```
<UserControl x:Class="WpfApplication5.UserControl2HardCoded"
             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">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="1" Grid.Row="0" Margin="4,4,4,0" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.Row="0" Margin="4,4,0,0" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Margin="4,4,4,4"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
```

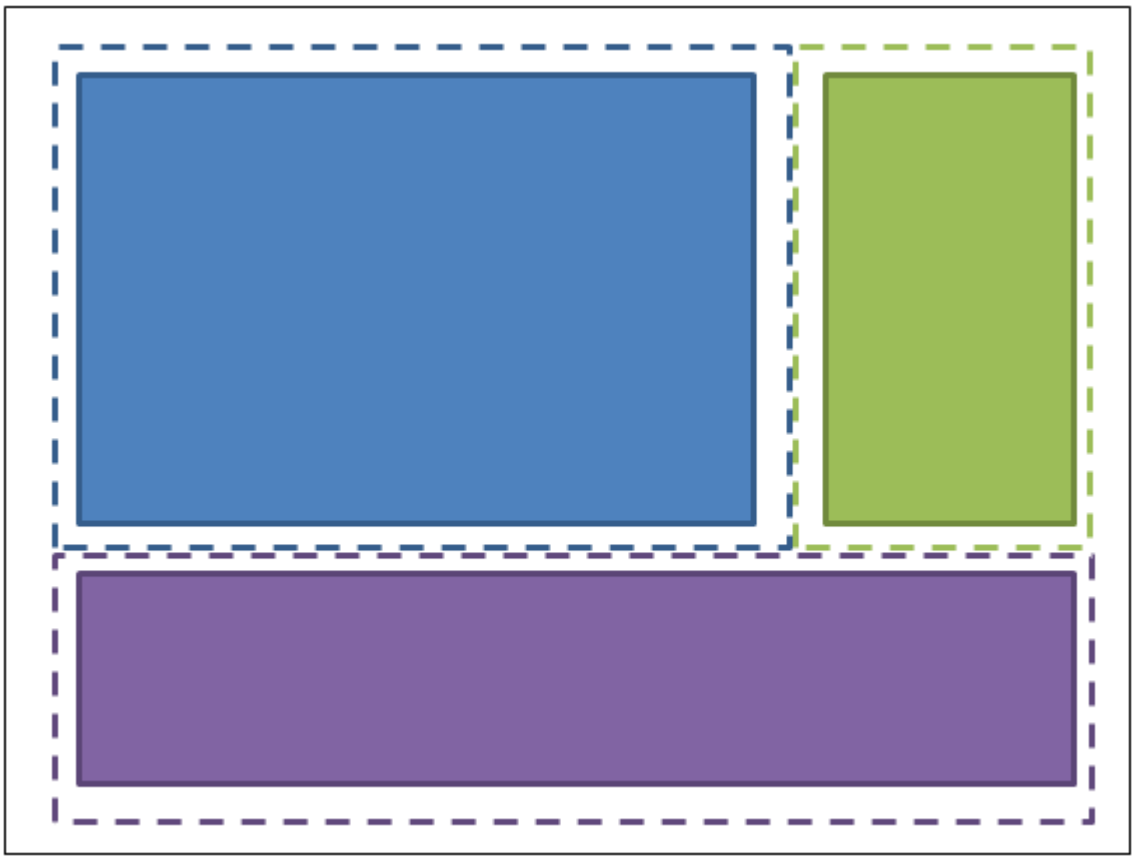
```
</UserControl>
```

Maintenant, mettons la boîte violette en haut. Les marges bleues deviennent alors 4,0,4,4; le vert devient 4,0,0,4.

```
<UserControl x:Class="WpfApplication5.UserControl3HardCoded"
             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">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="2*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="1" Grid.Row="1" Margin="4,0,4,4" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.Row="1" Margin="4,0,0,4" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" Margin="4,4,4,4"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
</UserControl>
```

Ne serait-ce pas bien de pouvoir déplacer les choses pour ne pas avoir à ajuster ces valeurs? Cela peut être réalisé simplement en pensant aux espaces de manière différente. Plutôt que d'allouer tout l'espace à un contrôle ou à un autre, imaginez que la moitié des espaces soient attribués à chaque case: (mon dessin n'est pas tout à fait à l'échelle - les lignes pointillées doivent se trouver à mi-chemin .



La boîte bleue a donc des marges de 2,2,2,2; la boîte verte a des marges de 2,2,2,2; la boîte pourpre a des marges de 2,2,2,2. Et le contenant dans lequel ils sont logés reçoit un rembourrage (pas de marge) de 2,2,2,2. Voici le XAML:

```
<UserControl x:Class="WpfApplication5.UserControl1HalfTheWhitespace"
             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"
             Padding="2,2,2,2">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="3*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="0" Grid.Row="0" Margin="2,2,2,2" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="1" Grid.Row="0" Margin="2,2,2,2" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Margin="2,2,2,2"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
</UserControl>
```

Maintenant, essayons de déplacer les boîtes, comme avant ... Mettons la boîte verte à gauche de la boîte bleue. OK fait. Et il n'y avait pas besoin de changer les rembourrages ou les marges. Voici le XAML:

```
<UserControl x:Class="WpfApplication5.UserControl2HalfTheWhitespace"
             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"
             Padding="2,2,2,2">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="1" Grid.Row="0" Margin="2,2,2,2" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.Row="0" Margin="2,2,2,2" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Margin="2,2,2,2"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
</UserControl>
```

Maintenant, mettons la boîte violette en haut. OK fait. Et il n'y avait pas besoin de changer les rembourrages ou les marges. Voici le XAML:

```
<UserControl x:Class="WpfApplication5.UserControl3HalfTheWhitespace"
             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"
             Padding="2,2,2,2">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="2*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="1" Grid.Row="1" Margin="2,2,2,2" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.Row="1" Margin="2,2,2,2" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" Margin="2,2,2,2"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
```

```
    </Grid>
</UserControl>
```

Comment l'utiliser dans le code réel

Pour généraliser ce que nous avons démontré ci - dessus: les choses individuelles contiennent une *marge* fixe de « demi-the-espaces », et le conteneur , ils sont maintenus en devrait avoir un *rembourrage* de « demi-the-espaces ». Vous pouvez appliquer ces styles dans votre dictionnaire de ressources d'application et vous n'aurez même pas besoin de les mentionner sur les éléments individuels. Voici comment définir "HalfTheWhiteSpace":

```
<system:Double x:Key="DefaultMarginSize">2</system:Double>
<Thickness x:Key="HalfTheWhiteSpace" Left="{StaticResource DefaultMarginSize}"
Top="{StaticResource DefaultMarginSize}" Right="{StaticResource DefaultMarginSize}"
Bottom="{StaticResource DefaultMarginSize}"/>
```

Ensuite, je peux définir un style de base sur lequel baser mes autres styles de contrôles: (cela pourrait aussi contenir vos polices FontFamily, FontSize, etc. par défaut)

```
<Style x:Key="BaseStyle" TargetType="{x:Type Control}">
    <Setter Property="Margin" Value="{StaticResource HalfTheWhiteSpace}"/>
</Style>
```

Ensuite, je peux définir mon style par défaut pour TextBox afin d'utiliser cette marge:

```
<Style TargetType="TextBox" BasedOn="{StaticResource BaseStyle}"/>
```

Je peux faire ce genre de chose pour DatePickers, Labels, etc., etc. (tout ce qui peut être contenu dans un conteneur). Méfiez-vous de styling TextBlock comme ça ... ce contrôle est utilisé en interne par un grand nombre de contrôles. Je vous suggère de créer votre propre contrôle qui dérive simplement de TextBlock. Vous pouvez styler *votre* TextBlock pour utiliser la marge par défaut; et vous devriez utiliser *votre* TextBlock chaque fois que vous utilisez explicitement un TextBlock dans votre XAML.

Vous pouvez utiliser une approche similaire pour appliquer le remplissage aux conteneurs communs (par exemple, ScrollViewer, Border, etc.).

Une fois cela fait, la *plupart* de vos contrôles n'auront plus besoin de marges et de remplissage - et vous devrez uniquement spécifier des valeurs dans des endroits où vous souhaitez intentionnellement vous écarter de ce principe de conception.

Lire Principe de conception "Half the Whitespace" en ligne:
https://riptutorial.com/fr/wpf/topic/9407/principe-de-conception--half-the-whitespace-

Chapitre 16: Prise en charge de la diffusion vidéo en continu et de l'affectation des tableaux de pixels à un contrôle d'image

Paramètres

Paramètres	Détails
PixelHeight (System.Int32)	La hauteur de l'image en unités de pixels d'image
PixelWidth (System.Int32)	La largeur de l'image en unités de pixels d'image
PixelFormat (System.Windows.Media.PixelFormat)	La largeur de l'image en unités de pixels d'image
Des pixels	Tout ce qui implémente IList <T> - y compris le tableau d'octets C #
DpiX	Spécifie le Dpi horizontal - Facultatif
DpiY	Spécifie le Dpi vertical - Facultatif

Remarques

- Assurez-vous de référencer l' *assembly System.Windows.Interactivity* afin que l'analyseur XAML reconnaisse la déclaration *xmlns: i* .
- Notez que l'instruction *xmlns: b* correspond à l'espace de noms où réside l'implémentation du comportement
- L'exemple suppose une connaissance pratique des expressions de liaison et de XAML.
- Ce comportement prend en charge l'affectation de pixels à une image sous la forme d'un tableau d'octets, même si le type de propriété de dépendance est spécifié en tant *qu'IList* . Cela fonctionne car le tableau d'octets C # implémente le *IList* interface.
- Le comportement atteint de très hautes performances et peut être utilisé pour le streaming vidéo
- N'affectez pas la propriété de dépendance à la *source de* l'image - liez plutôt la propriété de dépendance *Pixels*
- Les propriétés Pixels, PixelWidth, PixelHeight et PixelFormat doivent être affectées aux pixels à afficher
- L'attribution de la propriété de l'ordre de dépendance n'a pas d'importance

Examples

Mise en œuvre du comportement

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Interactivity;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace MyBehaviorAssembly
{
    public class PixelSupportBehavior : Behavior<Image>
    {

    public static readonly DependencyProperty PixelsProperty = DependencyProperty.Register(
        "Pixels", typeof (IList<byte>), typeof (PixelSupportBehavior), new
PropertyMetadata(default(IList<byte>),OnPixelsChanged));

    private static void OnPixelsChanged(DependencyObject d, DependencyPropertyChangedEventArgs
e)
    {
        var b = (PixelSupportBehavior) d;
        var pixels = (IList<byte>) e.NewValue;

        b.RenderPixels(pixels);
    }

    public IList<byte> Pixels
    {
        get { return (IList<byte>) GetValue(PixelsProperty); }
        set { SetValue(PixelsProperty, value); }
    }

    public static readonly DependencyProperty PixelFormatProperty =
DependencyProperty.Register(
        "PixelFormat", typeof (PixelFormat), typeof (PixelSupportBehavior), new
PropertyMetadata(PixelFormats.Default,OnPixelFormatChanged));


    private static void OnPixelFormatChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
    {
        var b = (PixelSupportBehavior) d;
        var pixelFormat = (PixelFormat) e.NewValue;

        if(pixelFormat==PixelFormats.Default)
            return;

        b._pixelFormat = pixelFormat;

        b.InitializeBufferIfAttached();
    }

    public PixelFormat PixelFormat
```

```
    {
        get { return (PixelFormat) GetValue(PixelFormatProperty); }
        set { SetValue(PixelFormatProperty, value); }
    }

    public static readonly DependencyProperty PixelWidthProperty =
DependencyProperty.Register(
        "PixelWidth", typeof (int), typeof (PixelSupportBehavior), new
PropertyMetadata(default(int),OnPixelWidthChanged));

    private static void OnPixelWidthChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
    {
        var b = (PixelSupportBehavior)d;
        var value = (int)e.NewValue;

        if(value<=0)
            return;

        b._pixelWidth = value;

        b.InitializeBufferIfAttached();
    }

    public int PixelWidth
    {
        get { return (int) GetValue(PixelWidthProperty); }
        set { SetValue(PixelWidthProperty, value); }
    }


    public static readonly DependencyProperty PixelHeightProperty =
DependencyProperty.Register(
        "PixelHeight", typeof (int), typeof (PixelSupportBehavior), new
PropertyMetadata(default(int),OnPixelHeightChanged));

    private static void OnPixelHeightChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
    {
        var b = (PixelSupportBehavior)d;
        var value = (int)e.NewValue;

        if (value <= 0)
            return;

        b._pixelHeight = value;

        b.InitializeBufferIfAttached();
    }

    public int PixelHeight
    {
        get { return (int) GetValue(PixelHeightProperty); }
        set { SetValue(PixelHeightProperty, value); }
    }

    public static readonly DependencyProperty DpiXProperty = DependencyProperty.Register(
        "DpiX", typeof (int), typeof (PixelSupportBehavior), new
PropertyMetadata(96,OnDpiXChanged));

    private static void OnDpiXChanged(DependencyObject d, DependencyPropertyChangedEventArgs
```

```
e)
    {
        var b = (PixelSupportBehavior)d;
        var value = (int)e.NewValue;

        if (value <= 0)
            return;

        b._dpiX = value;

        b.InitializeBufferIfAttached();
    }

    public int DpiX
    {
        get { return (int) GetValue(DpiXProperty); }
        set { SetValue(DpiXProperty, value); }
    }

    public static readonly DependencyProperty DpiYProperty = DependencyProperty.Register(
        "DpiY", typeof (int), typeof (PixelSupportBehavior), new
PropertyMetadata(96,OnDpiYChanged));

    private static void OnDpiYChanged(DependencyObject d, DependencyPropertyChangedEventArgs
e)
    {
        var b = (PixelSupportBehavior)d;
        var value = (int)e.NewValue;

        if (value <= 0)
            return;

        b._dpiY = value;

        b.InitializeBufferIfAttached();
    }

    public int DpiY
    {
        get { return (int) GetValue(DpiYProperty); }
        set { SetValue(DpiYProperty, value); }
    }

    private IntPtr _backBuffer = IntPtr.Zero;
    private int _bytesPerPixel;
    private const int BitsPerByte = 8;
    private int _pixelWidth;
    private int _pixelHeight;
    private int _dpiX;
    private int _dpiY;
    private Int32Rect _imageRectangle;
    private readonly GCHandle _defaultGcHandle = new GCHandle();
    private PixelFormat _pixelFormat;

    private int _byteArraySize;
    private WriteableBitmap _bitMap;

    private bool _attached;

    protected override void OnAttached()
    {
```

```
        _attached = true;
        InitializeBufferIfAttached();
    }

    private void InitializeBufferIfAttached()
    {
        if(_attached==false)
            return;

        ReevaluateBitsPerPixel();

        RecomputeByteArraySize();

        ReinitializeImageSource();
    }

    private void ReevaluateBitsPerPixel()
    {
        var f = _pixelFormat;

        if (f == PixelFormats.Default)
        {
            _bytesPerPixel = 0;
            return;
        };

        _bytesPerPixel = f.BitsPerPixel/BitsPerByte;
    }

    private void ReinitializeImageSource()
    {
        var f = _pixelFormat;
        var h = _pixelHeight;
        var w = _pixelWidth;

        if (w<=0|| h<=0 || f== PixelFormats.Default)
            return;

        _imageRectangle = new Int32Rect(0, 0, w, h);
        _bitMap = new WriteableBitmap(w, h, _dpiX, _dpiY, f, null);
        _backBuffer = _bitMap.BackBuffer;
        AssociatedObject.Source = _bitMap;
    }

    private void RenderPixels(IList<byte> pixels)
    {
        if (pixels == null)
        {
            return;
        }

        var buffer = _backBuffer;
        if (buffer == IntPtr.Zero)
            return;

        var size = _byteArraySize;

        var gcHandle = _defaultGcHandle;
        var allocated = false;
        var bitMap = _bitMap;
        var rect = _imageRectangle;
```

```
        var w = _pixelWidth;
        var locked = false;
        try
        {
            gcHandle = GCHandle.Alloc(pixels, GCHandleType.Pinned);
            allocated = true;

            bitMap.Lock();
            locked = true;
            var ptr = gcHandle.AddrOfPinnedObject();
            _bitMap.WritePixels(rect, ptr, size,w);
        }
        finally
        {
            if(locked)
                bitMap.Unlock();

            if (allocated)
                gcHandle.Free();
        }
    }

    private void RecomputeByteArraySize()
    {
        var h = _pixelHeight;
        var w = _pixelWidth;
        var bpp = _bytesPerPixel;

        if (w<=0|| h<=0 || bpp<=0)
            return;

        _byteArraySize = (w * h * bpp);
    }

    public PixelSupportBehavior()
    {
        _pixelFormat = PixelFormats.Default;
    }
}
}
```

Utilisation XAML

```
<UserControl x:Class="Example.View"
         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"
         xmlns:b="clr-namespace:MyBehaviorAssembly;assembly=MyBehaviorAssembly"
         xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
         mc:Ignorable="d"
         d:DesignHeight="200" d:DesignWidth="200"
         >

                <Image Stretch="Uniform">

                <i:Interaction.Behaviors>

                    <b:PixelSupportBehavior
```

```
                            PixelHeight="{Binding PixelHeight}"
                            PixelWidth="{Binding PixelWidth}"
                            PixelFormat="{Binding PixelFormat}"
                            Pixels="{Binding Pixels}"
                            />

                </i:Interaction.Behaviors>

            </Image>

</UserControl>
```

Lire Prise en charge de la diffusion vidéo en continu et de l'affectation des tableaux de pixels à un contrôle d'image en ligne: https://riptutorial.com/fr/wpf/topic/6435/prise-en-charge-de-la-diffusion-video-en-continu-et-de-l-affectation-des-tableaux-de-pixels-a-un-controle-d-image

Chapitre 17: Propriétés de dépendance

Introduction

Les propriétés de dépendance sont un type de propriété qui étend une propriété CLR. Alors qu'une propriété CLR est lue directement à partir d'un membre de votre classe, une propriété de dépendance sera résolue dynamiquement lors de l'appel de la méthode GetValue () que votre objet gagne via l'héritage de la classe DependencyObject de base.

Cette section va décomposer les propriétés de dépendance et expliquer leur utilisation à la fois conceptuellement et à travers des exemples de code.

Syntaxe

- DependencyProperty.Register (nom de la chaîne, type propertyType, type ownerType)
- DependencyProperty.Register (nom de la chaîne, type propertyType, type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.Register (nom de la chaîne, type propertyType, type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterAttached (nom de la chaîne, type propertyType, type ownerType)
- DependencyProperty.RegisterAttached (nom de la chaîne, type propertyType, type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterAttached (nom de la chaîne, type propertyType, type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterReadOnly (nom de la chaîne, type propertyType, type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterReadOnly (nom de la chaîne, type propertyType, type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterAttachedReadOnly (nom de la chaîne, type propertyType, type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterAttachedReadOnly (nom de la chaîne, type propertyType, type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)

Paramètres

Paramètre	Détails
prénom	La représentation sous forme de `String` du nom de la propriété
Type de propriété	Le `Type` de la propriété, par exemple `typeof(int)`
propriétaireType	`Type` de la classe dans laquelle la propriété est définie, par exemple `typeof(MyControl)` ou `typeof(MyAttachedProperties)` .

Paramètre	Détails
typeMetadata	Instance de `System.Windows.PropertyMetadata` (ou de l'une de ses sous-classes) qui définit les valeurs par défaut, les callbacks de propriété modifiés, `FrameworkPropertyMetadata` permet de définir des options de liaison telles que `System.Windows.Data.BindingMode.TwoWay` .
ValiderValueCallback	Rappel personnalisé qui renvoie true si la nouvelle valeur de la propriété est valide, sinon fausse.

Examples

Propriétés de dépendance standard

Quand utiliser

Pratiquement tous les contrôles WPF font un usage intensif des propriétés de dépendance. Une propriété de dépendance permet d'utiliser de nombreuses fonctionnalités WPF qui ne sont pas possibles avec les propriétés CLR standard uniquement, notamment la prise en charge des styles, des animations, de la liaison de données, de l'héritage des valeurs et des notifications de modification.

La propriété `TextBox.Text` est un exemple simple de la nécessité d'une propriété de dépendance standard. Ici, la liaison de données ne serait pas possible si `Text` était une propriété CLR standard.

```
<TextBox Text="{Binding FirstName}" />
```

Comment définir

Les propriétés de dépendance ne peuvent être définies que dans des classes dérivées de `DependencyObject` , telles que `FrameworkElement` , `Control` , etc.

L'un des moyens les plus rapides de créer une propriété de dépendance standard sans avoir à retenir la syntaxe consiste à utiliser l'extrait de code "propdp" en tapant `propdp` , puis en appuyant sur `Tab` . Un extrait de code sera inséré et pourra être modifié pour répondre à vos besoins:

```
public class MyControl : Control
{
    public int MyProperty
    {
        get { return (int)GetValue(MyPropertyProperty); }
        set { SetValue(MyPropertyProperty, value); }
    }

    // Using a DependencyProperty as the backing store for MyProperty.
    // This enables animation, styling, binding, etc...
    public static readonly DependencyProperty MyPropertyProperty =
```

```
        DependencyProperty.Register("MyProperty", typeof(int), typeof(MyControl),
            new PropertyMetadata(0));
}
```

Vous devez `naviguer` à travers les différentes parties de l'extrait de code pour effectuer les modifications nécessaires, y compris la mise à jour du nom de la propriété, type de propriété, contenant un type de classe, et la valeur par défaut.

Conventions importantes

Il existe quelques conventions / règles importantes à suivre ici:

1. **Créez une propriété CLR pour la propriété de dépendance.** Cette propriété est utilisée dans le code-derrière de votre objet ou par d'autres consommateurs. Il devrait invoquer `GetValue` et `SetValue` pour que les consommateurs n'aient pas à le faire.

2. **Nommez correctement la propriété de dépendance.** Le champ `DependencyProperty` doit être `public static readonly` . Il doit avoir un nom qui correspond au nom de la propriété CLR et se terminer par "Property", par exemple `Text` et `TextProperty` .

3. **N'ajoutez pas de logique supplémentaire au setter de la propriété CLR.** Le système de propriété de dépendance (et spécifiquement XAML) n'utilise pas la propriété CLR. Si vous souhaitez effectuer une action lorsque la valeur de la propriété change, vous devez fournir un rappel via `PropertyMetadata` :

   ```
   public static readonly DependencyProperty MyPropertyProperty =
       DependencyProperty.Register("MyProperty", typeof(int), typeof(MyControl),
           new PropertyMetadata(0, MyPropertyChangedHandler));

   private static void MyPropertyChangedHandler(DependencyObject sender,
   DependencyPropertyChangedEventArgs args)
   {
       // Use args.OldValue and args.NewValue here as needed.
       // sender is the object whose property changed.
       // Some unboxing required.
   }
   ```

Mode de liaison

Pour éliminer la nécessité de spécifier `Mode=TwoWay` dans les liaisons (similaire au comportement de `TextBox.Text`), mettez à jour le code pour utiliser `FrameworkPropertyMetadata` au lieu de `PropertyMetadata` et spécifiez l'indicateur approprié:

```
public static readonly DependencyProperty MyPropertyProperty =
    DependencyProperty.Register("MyProperty", typeof(int), typeof(MyControl),
        new FrameworkPropertyMetadata(0,
FrameworkPropertyMetadataOptions.BindsTwoWayByDefault));
```

Quand utiliser

Une propriété attachée est une propriété de dépendance qui peut être appliquée à n'importe quel `DependencyObject` pour améliorer le comportement de divers contrôles ou services qui connaissent l'existence de la propriété.

Certains cas d'utilisation pour les propriétés jointes incluent:

1. Avoir un élément parent itérer à travers ses enfants et agir sur les enfants d'une certaine manière. Par exemple, le contrôle `Grid` utilise les `Grid.Row` `Grid.Column` `Grid.Row` , `Grid.Column` , `Grid.RowSpan` et `Grid.ColumnSpan` pour organiser les éléments en lignes et en colonnes.
2. Ajouter des éléments visuels aux contrôles existants à l'aide de modèles personnalisés, par exemple en ajoutant des filigranes à des zones de texte vides dans toute l'application sans avoir à sous- `TextBox` .
3. Fournir un service ou une fonctionnalité générique à certains ou à tous les contrôles existants, par exemple `ToolTipService` ou `FocusManager` . Ceux-ci sont communément appelés *comportements attachés* .
4. Lorsque l'héritage de l'arborescence visuelle est requis, par exemple, similaire au comportement de `DataContext` .

Cela démontre davantage ce qui se passe dans le cas d'utilisation de la `Grid` :

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

    <Label Grid.Column="0" Content="Your Name:" />
    <TextBox Grid.Column="1" Text="{Binding FirstName}" />
</Grid>
```

`Grid.Column` n'est pas une propriété existant sur `Label` ou `TextBox` . Le contrôle `Grid` examine plutôt ses éléments enfants et les organise en fonction des valeurs des propriétés attachées.

Comment définir

Nous continuerons à utiliser la `Grid` pour cet exemple. La définition de `Grid.Column` est indiquée ci-dessous, mais `DependencyPropertyChangedEventHandler` est exclu pour des raisons de concision.

```
public static readonly DependencyProperty RowProperty =
    DependencyProperty.RegisterAttached("Row", typeof(int), typeof(Grid),
        new FrameworkPropertyMetadata(0, ...));

public static void SetRow(UIElement element, int value)
{
```

```
    if (element == null)
        throw new ArgumentNullException("element");

    element.SetValue(RowProperty, value);
}

public static int GetRow(UIElement element)
{
    if (element == null)
        throw new ArgumentNullException("element");

    return ((int)element.GetValue(RowProperty));
}
```

Les propriétés attachées pouvant être associées à une grande variété d'éléments, elles ne peuvent pas être implémentées en tant que propriétés CLR. Une paire de méthodes statiques est introduite à la place.

Par conséquent, contrairement aux propriétés de dépendance standard, les propriétés attachées peuvent également être définies dans des classes non dérivées de `DependencyObject` .

Les mêmes conventions de dénomination qui s'appliquent aux propriétés de dépendance normales s'appliquent également ici: la propriété de dépendance `RowProperty` a les méthodes correspondantes `GetRow` et `SetRow` .

Mises en garde

Comme documenté sur MSDN :

> Bien que l'héritage des valeurs de propriété puisse sembler fonctionner pour les propriétés de dépendance non attachées, le comportement d'héritage pour une propriété non attachée via certaines limites d'éléments dans l'arborescence d'exécution n'est pas défini. Utilisez toujours RegisterAttached pour enregistrer les propriétés dans lesquelles vous spécifiez Inherits dans les métadonnées.

Propriétés de dépendance en lecture seule

Quand utiliser

Une propriété de dépendance en lecture seule est similaire à une propriété de dépendance normale, mais elle est structurée pour ne pas permettre que sa valeur soit définie en dehors du contrôle. Cela fonctionne bien si vous avez une propriété purement informative pour les consommateurs, par exemple `IsMouseOver` ou `IsKeyboardFocusWithin` .

Comment définir

Tout comme les propriétés de dépendance standard, une propriété de dépendance en lecture

seule doit être définie sur une classe dérivée de `DependencyObject` .

```
public class MyControl : Control
{
    private static readonly DependencyPropertyKey MyPropertyPropertyKey =
        DependencyProperty.RegisterReadOnly("MyProperty", typeof(int), typeof(MyControl),
            new FrameworkPropertyMetadata(0));

    public static readonly DependencyProperty MyPropertyProperty =
MyPropertyPropertyKey.DependencyProperty;

    public int MyProperty
    {
        get { return (int)GetValue(MyPropertyProperty); }
        private set { SetValue(MyPropertyPropertyKey, value); }
    }
}
```

Les mêmes conventions qui s'appliquent aux propriétés de dépendance régulières s'appliquent également ici, mais avec deux différences principales:

1. Le `DependencyProperty` provient d'un `DependencyPropertyKey` `private` .
2. Le setter de propriétés CLR est `protected` ou `private` au lieu de `public` .

Notez que le poseur passe `MyPropertyPropertyKey` et non `MyPropertyProperty` à la `SetValue` méthode. Étant donné que la propriété a été définie en lecture seule, toute tentative d'utilisation de `SetValue` sur la propriété doit être utilisée avec une surcharge qui reçoit `DependencyPropertyKey` ; sinon, une `InvalidOperationException` sera lancée.

Lire Propriétés de dépendance en ligne: https://riptutorial.com/fr/wpf/topic/2914/proprietes-de-dependance

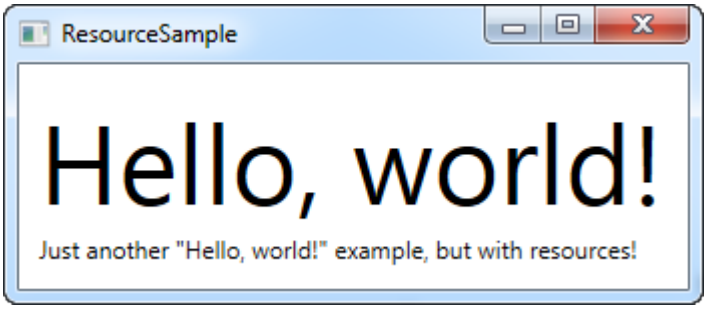
Chapitre 18: Ressources WPF

Examples

Bonjour ressources

WPF introduit un concept très pratique: possibilité de stocker des données en tant que ressource, localement pour un contrôle, localement pour la fenêtre entière ou globalement pour l'application entière. Les données peuvent être à peu près ce que vous voulez, des informations réelles à une hiérarchie de contrôles WPF. Cela vous permet de placer des données à un endroit et de les utiliser à partir de plusieurs autres endroits, ce qui est très utile. Le concept est beaucoup utilisé pour les styles et les modèles.

```
<Window x:Class="WPFApplication.ResourceSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="ResourceSample" Height="150" Width="350">
    <Window.Resources>
        <sys:String x:Key="strHelloWorld">Hello, world!</sys:String>
    </Window.Resources>
    <StackPanel Margin="10">
        <TextBlock Text="{StaticResource strHelloWorld}" FontSize="56" />
        <TextBlock>Just another "<TextBlock Text="{StaticResource strHelloWorld}" />" example,
but with resources!</TextBlock>
    </StackPanel>
</Window>
```



Les ressources reçoivent une clé, en utilisant l'attribut x: Key, qui vous permet de le référencer à partir d'autres parties de l'application en utilisant cette clé, en combinaison avec l'extension de balisage StaticResource. Dans cet exemple, je stocke simplement une chaîne simple, que j'utilise ensuite à partir de deux contrôles TextBlock différents.

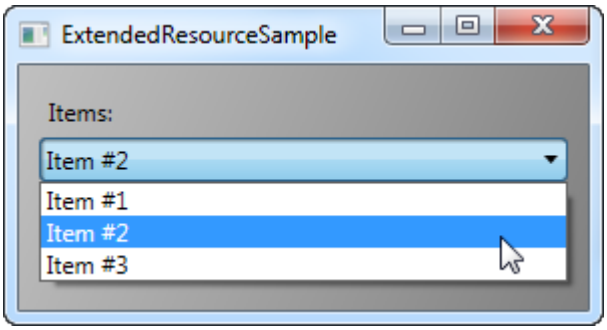
Types de ressources

Partager une chaîne simple était facile, mais vous pouvez faire beaucoup plus. Dans cet exemple, je vais également stocker un tableau complet de chaînes, ainsi qu'un pinceau de dégradé à utiliser pour l'arrière-plan. Cela devrait vous donner une bonne idée de ce que vous pouvez faire avec les ressources:

```
<Window x:Class="WPFApplication.ExtendedResourceSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="ExtendedResourceSample" Height="160" Width="300"
        Background="{DynamicResource WindowBackgroundBrush}">
    <Window.Resources>
        <sys:String x:Key="ComboBoxTitle">Items:</sys:String>

        <x:Array x:Key="ComboBoxItems" Type="sys:String">
            <sys:String>Item #1</sys:String>
            <sys:String>Item #2</sys:String>
            <sys:String>Item #3</sys:String>
        </x:Array>

        <LinearGradientBrush x:Key="WindowBackgroundBrush">
            <GradientStop Offset="0" Color="Silver"/>
            <GradientStop Offset="1" Color="Gray"/>
        </LinearGradientBrush>
    </Window.Resources>
    <StackPanel Margin="10">
        <Label Content="{StaticResource ComboBoxTitle}" />
        <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
    </StackPanel>
</Window>
```



Cette fois, nous avons ajouté quelques ressources supplémentaires, de sorte que notre fenêtre contient maintenant une chaîne simple, un tableau de chaînes et un LinearGradientBrush. La chaîne est utilisée pour l'étiquette, le tableau de chaînes est utilisé comme éléments pour le contrôle ComboBox et le pinceau dégradé est utilisé comme arrière-plan pour la fenêtre entière. Ainsi, comme vous pouvez le voir, pratiquement tout peut être stocké en tant que ressource.

Ressources locales et applicatives

Si vous avez uniquement besoin d'une ressource donnée pour un contrôle spécifique, vous pouvez la rendre plus locale en l'ajoutant à ce contrôle spécifique, au lieu de la fenêtre. Cela fonctionne exactement de la même manière, la seule différence étant que vous ne pouvez désormais accéder qu'à partir de la portée du contrôle où vous l'avez placé:

```
<StackPanel Margin="10">
    <StackPanel.Resources>
        <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
    </StackPanel.Resources>
    <Label Content="{StaticResource ComboBoxTitle}" />
</StackPanel>
```

Dans ce cas, nous ajoutons la ressource au StackPanel, puis l'utilisons depuis son contrôle enfant, le Label. D'autres contrôles à l'intérieur du StackPanel auraient pu l'utiliser aussi, tout comme les enfants de ces contrôles enfants auraient pu y accéder. Les contrôles en dehors de ce StackPanel particulier n'y auraient pas accès.

Si vous avez besoin de la possibilité d'accéder à la ressource à partir de plusieurs fenêtres, cela est également possible. Le fichier App.xaml peut contenir des ressources comme la fenêtre et tout type de contrôle WPF. Lorsque vous les stockez dans App.xaml, ils sont globalement accessibles dans toutes les fenêtres et les contrôles utilisateur du projet. Cela fonctionne exactement comme lors du stockage et de l'utilisation d'une fenêtre:

```
<Application x:Class="WpfSamples.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             StartupUri="WPFApplication/ExtendedResourceSample.xaml">
    <Application.Resources>
        <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
    </Application.Resources>
</Application>
```

En l'utilisant également, WPF monte automatiquement la portée, du contrôle local à la fenêtre puis à App.xaml, pour trouver une ressource donnée:

```
<Label Content="{StaticResource ComboBoxTitle}" />
```

Ressources de Code-behind

Dans cet exemple, nous accédons à trois ressources différentes de Code-behind, chacune stockée dans une portée différente.

App.xaml:

```
<Application x:Class="WpfSamples.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             StartupUri="WPFApplication/ResourcesFromCodeBehindSample.xaml">
    <Application.Resources>
        <sys:String x:Key="strApp">Hello, Application world!</sys:String>
    </Application.Resources>
</Application>
```

Fenêtre:

```
<Window x:Class="WpfSamples.WPFApplication.ResourcesFromCodeBehindSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="ResourcesFromCodeBehindSample" Height="175" Width="250">
    <Window.Resources>
        <sys:String x:Key="strWindow">Hello, Window world!</sys:String>
```

```
    </Window.Resources>
    <DockPanel Margin="10" Name="pnlMain">
        <DockPanel.Resources>
            <sys:String x:Key="strPanel">Hello, Panel world!</sys:String>
        </DockPanel.Resources>

        <WrapPanel DockPanel.Dock="Top" HorizontalAlignment="Center" Margin="10">
            <Button Name="btnClickMe" Click="btnClickMe_Click">Click me!</Button>
        </WrapPanel>

        <ListBox Name="lbResult" />
    </DockPanel>
</Window>
```

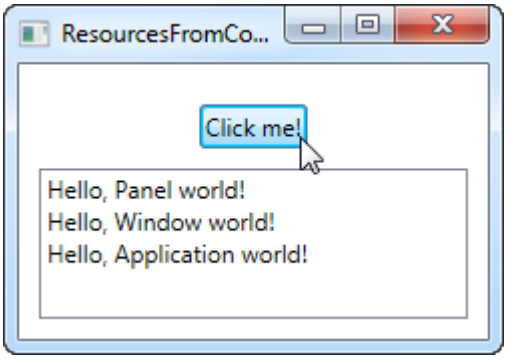
Code-behind:

```
using System;
using System.Windows;

namespace WpfSamples.WPFApplication
{
        public partial class ResourcesFromCodeBehindSample : Window
        {
                public ResourcesFromCodeBehindSample()
                {
                        InitializeComponent();
                }

                private void btnClickMe_Click(object sender, RoutedEventArgs e)
                {
                        lbResult.Items.Add(pnlMain.FindResource("strPanel").ToString());
                        lbResult.Items.Add(this.FindResource("strWindow").ToString());

lbResult.Items.Add(Application.Current.FindResource("strApp").ToString());
                }
        }
}
```



Donc, comme vous pouvez le voir, nous stockons trois différents "Hello, world!" messages: un dans App.xaml, un dans la fenêtre et un autre pour le panneau principal. L'interface est composée d'un bouton et d'une ListBox.

Dans Code-behind, nous gérons l'événement click du bouton, dans lequel nous ajoutons chacune des chaînes de texte au contrôle ListBox, comme le montre la capture d'écran. Nous utilisons la méthode FindResource (), qui retournera la ressource en tant qu'objet (si elle est trouvée), puis

nous la transformons en chaîne que nous connaissons en utilisant la méthode ToString ().

Remarquez comment nous utilisons la méthode FindResource () sur différentes étendues - d'abord sur le panneau, puis sur la fenêtre, puis sur l'objet Application actuel. Il est logique de rechercher la ressource à sa connaissance, mais, comme nous l'avons déjà mentionné, si une ressource n'est pas trouvée, la recherche progresse dans la hiérarchie. En principe, nous aurions pu utiliser la méthode FindResource () sur le panneau tous les trois cas, car il aurait continué jusqu'à la fenêtre et plus tard au niveau de l'application, si non trouvé.

La même chose n'est pas vraie dans l'autre sens: la recherche ne navigue pas dans l'arborescence. Vous ne pouvez donc pas rechercher une ressource au niveau de l'application, si elle a été définie localement pour le contrôle ou pour la fenêtre.

Lire Ressources WPF en ligne: https://riptutorial.com/fr/wpf/topic/4371/ressources-wpf

Chapitre 19: Slider Binding: mise à jour uniquement sur Drag Ended

Paramètres

Paramètre	Détail
Valeur (flottant)	La propriété liée à cette propriété de dépendance sera mise à jour chaque fois que l'utilisateur cessera de faire glisser le curseur

Remarques

- Assurez-vous de référencer l'assembly *System.Windows.Interactivity* afin que l'analyseur XAML reconnaisse la déclaration *xmlns: i* .
- Notez que l'instruction *xmlns: b* correspond à l'espace de noms où réside l'implémentation du comportement
- L'exemple suppose une connaissance pratique des expressions de liaison et de XAML.

Examples

Mise en œuvre du comportement

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Interactivity;

namespace MyBehaviorAssembly
{

public class SliderDragEndValueBehavior : Behavior<Slider>
{

    public static readonly DependencyProperty ValueProperty = DependencyProperty.Register(
        "Value", typeof (float), typeof (SliderDragEndValueBehavior), new
PropertyMetadata(default(float)));

    public float Value
    {
        get { return (float) GetValue(ValueProperty); }
        set { SetValue(ValueProperty, value); }
    }

    protected override void OnAttached()
    {
        RoutedEventHandler handler = AssociatedObject_DragCompleted;
        AssociatedObject.AddHandler(Thumb.DragCompletedEvent, handler);
    }
```

```
    private void AssociatedObject_DragCompleted(object sender, RoutedEventArgs e)
    {
        Value = (float) AssociatedObject.Value;
    }

    protected override void OnDetaching()
    {
        RoutedEventHandler handler = AssociatedObject_DragCompleted;
        AssociatedObject.RemoveHandler(Thumb.DragCompletedEvent, handler);
    }
}
}
```

Utilisation XAML

```
<UserControl x:Class="Example.View"
         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"
         xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"

         xmlns:b="MyBehaviorAssembly;assembly=MyBehaviorAssembly"

         mc:Ignorable="d"
         d:DesignHeight="200" d:DesignWidth="200"

         >
           <Slider>
            <i:Interaction.Behaviors>
                <b:SliderDragEndValueBehavior

                        Value="{Binding Value, Mode=OneWayToSource,
UpdateSourceTrigger=PropertyChanged}"

                        />
            </i:Interaction.Behaviors>
          </Slider>

</UserControl>
```

Lire Slider Binding: mise à jour uniquement sur Drag Ended en ligne: https://riptutorial.com/fr/wpf/topic/6339/slider-binding--mise-a-jour-uniquement-sur-drag-ended

Chapitre 20: Styles dans WPF

Remarques

Remarques introductives

Dans WPF, un **style** définit les valeurs d'une ou plusieurs propriétés de dépendance pour un élément visuel donné. Les styles sont utilisés tout au long de l'application pour rendre l'interface utilisateur plus cohérente (par exemple, donner à tous les boutons de dialogue une taille cohérente) et pour faciliter les modifications en bloc (par exemple, modifier la largeur de tous les boutons).

Les styles sont généralement définis dans un `ResourceDictionary` à un niveau élevé dans l'application (par exemple, dans *App.xaml* ou dans un thème). Ils sont donc disponibles dans toute l'application, mais ils peuvent également être définis pour un seul élément et ses enfants. style à tous les éléments `TextBlock` intérieur d'un `StackPanel` .

```
<StackPanel>
    <StackPanel.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Margin" Value="5,5,5,0"/>
            <Setter Property="Background" Value="#FFF0F0F0"/>
            <Setter Property="Padding" Value="5"/>
        </Style>
    </StackPanel.Resources>

    <TextBlock Text="First Child"/>
    <TextBlock Text="Second Child"/>
    <TextBlock Text="Third Child"/>
</StackPanel>
```

Notes IMPORTANTES

- L'emplacement où le style est défini affecte l'endroit où il est disponible.
- Les références en `StaticResource` ne peuvent pas être résolues par `StaticResource` . En d'autres termes, si vous définissez un style qui dépend d'un autre style ou d'une autre ressource dans un dictionnaire de ressources, il doit être défini après / sous la ressource dont il dépend.
- `StaticResource` est la méthode recommandée pour référencer les styles et autres ressources (pour des raisons de performances et de comportement), sauf si vous avez spécifiquement besoin de `DynamicResource` , par exemple pour des thèmes pouvant être modifiés à l'exécution.

Ressources

MSDN propose des articles détaillés sur les styles et les ressources qui ont plus de profondeur que ce qu'il est possible de fournir ici.

- Aperçu des ressources
- Styling et Templating
- Présentation de la création de contrôles

Examples

Définir un style nommé

Un style nommé requiert que la propriété `x:Key` soit définie et s'applique uniquement aux éléments qui le référencent explicitement par nom:

```
<StackPanel>
    <StackPanel.Resources>
        <Style x:Key="MyTextBlockStyle" TargetType="TextBlock">
            <Setter Property="Background" Value="Yellow"/>
            <Setter Property="FontWeight" Value="Bold"/>
        </Style>
    </StackPanel.Resources>

    <TextBlock Text="Yellow and bold!" Style="{StaticResource MyTextBlockStyle}" />
    <TextBlock Text="Also yellow and bold!" Style="{DynamicResource MyTextBlockStyle}" />
    <TextBlock Text="Plain text." />
</StackPanel>
```

Définir un style implicite

Un style implicite s'applique à tous les éléments d'un type donné dans la portée. Un style implicite peut omettre `x:Key` car il est implicitement identique à la propriété `TargetType` du style.

```
<StackPanel>
    <StackPanel.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Background" Value="Yellow"/>
            <Setter Property="FontWeight" Value="Bold"/>
        </Style>
    </StackPanel.Resources>

    <TextBlock Text="Yellow and bold!"  />
    <TextBlock Text="Also yellow and bold!" />
    <TextBlock Style="{x:Null}" Text="I'm not yellow or bold; I'm the control's default
style!" />
</StackPanel>
```

Hériter d'un style

Il est courant d'avoir besoin d'un style de base qui définisse des propriétés / valeurs partagées entre plusieurs styles appartenant au même contrôle, en particulier pour quelque chose comme `TextBlock` . Ceci est accompli en utilisant la propriété `BasedOn` . Les valeurs sont héritées et peuvent

ensuite être remplacées.

```
<Style x:Key="BaseTextBlockStyle" TargetType="TextBlock">
    <Setter Property="FontSize" Value="12"/>
    <Setter Property="Foreground" Value="#FFBBBBBB" />
    <Setter Property="FontFamily" Value="Arial" />
</Style>

<Style x:Key="WarningTextBlockStyle"
       TargetType="TextBlock"
       BasedOn="{StaticResource BaseTextBlockStyle">
    <Setter Property="Foreground" Value="Red"/>
    <Setter Property="FontWeight" Value="Bold" />
</Style>
```

Dans l'exemple ci-dessus, tout `TextBlock` utilisant le style `WarningTextBlockStyle` serait présenté sous la forme de 12px Arial en rouge et en gras.

Comme les styles implicites ont un `x:Key` implicite qui correspond à leur `TargetType` , vous pouvez également en hériter:

```
<!-- Implicit -->
<Style TargetType="TextBlock">
    <Setter Property="FontSize" Value="12"/>
    <Setter Property="Foreground" Value="#FFBBBBBB" />
    <Setter Property="FontFamily" Value="Arial" />
</Style>

<Style x:Key="WarningTextBlockStyle"
       TargetType="TextBlock"
       BasedOn="{StaticResource {x:Type TextBlock}}">
    <Setter Property="Foreground" Value="Red"/>
    <Setter Property="FontWeight" Value="Bold" />
</Style>
```

Lire Styles dans WPF en ligne: https://riptutorial.com/fr/wpf/topic/4090/styles-dans-wpf

Chapitre 21: Synthèse de discours

Introduction

Dans l'assembly `System.Speech` , Microsoft a ajouté la **synthèse vocale** , la possibilité de transformer du texte en mots parlés.

Syntaxe

1. SpeechSynthesizer speechSynthesizerObject = new SpeechSynthesizer ();

   speechSynthesizerObject.Speak ("Texte à parler");

Examples

Exemple de synthèse vocale - Bonjour tout le monde

```
using System;
using System.Speech.Synthesis;
using System.Windows;

namespace Stackoverflow.SpeechSynthesisExample
{
        public partial class SpeechSynthesisSample : Window
        {
                public SpeechSynthesisSample()
                {
                        InitializeComponent();
                        SpeechSynthesizer speechSynthesizer = new SpeechSynthesizer();
                        speechSynthesizer.Speak("Hello, world!");
                }
        }
}
```

Lire Synthèse de discours en ligne: https://riptutorial.com/fr/wpf/topic/8368/synthese-de-discours

Chapitre 22: System.Windows.Controls.WebBrowser

Introduction

Cela vous permet de placer un navigateur Web dans votre application WPF.

Remarques

Un point important à noter, qui n'est pas évident dans la documentation, et que vous pouvez utiliser pendant des années sans savoir, c'est qu'il se comporte par défaut comme InternetExplorer7 plutôt que comme votre installation InternetExplorer la plus récente (voir https: // weblog). west-wind.com/posts/2011/may/21/web-browser-control-specifying-the-ie-version).

Cela ne peut pas être résolu en définissant une propriété sur le contrôle; Vous devez soit modifier les pages affichées en ajoutant une balise Meta HTML, soit en appliquant un paramètre de registre (!). (Les détails des deux approches sont sur le lien ci-dessus.)

Par exemple, ce comportement bizarre peut vous amener à afficher un message "Erreur de script" / "Une erreur s'est produite dans le script sur cette page". Googler cette erreur peut vous faire penser que la solution consiste à essayer de supprimer l'erreur, plutôt que de comprendre le problème réel et d'appliquer la solution appropriée.

Examples

Exemple de WebBrowser dans un BusyIndicator

Sachez que le contrôle WebBrowser n'est pas compatible avec votre définition XAML et s'affiche par-dessus tout. Par exemple, si vous le mettez dans un BusyIndicator qui a été marqué comme étant occupé, il restera toujours au-dessus de ce contrôle. La solution consiste à lier la visibilité de WebBrowser à la valeur utilisée par BusyIndicator et à utiliser un convertisseur pour inverser le booléen et le convertir en Visibility. Par exemple:

```
    <telerik:RadBusyIndicator IsBusy="{Binding IsBusy}">
        <WebBrowser Visibility="{Binding IsBusy, Converter={StaticResource
InvertBooleanToVisibilityConverter}}"/>
    </telerik:RadBusyIndicator>
```

Lire System.Windows.Controls.WebBrowser en ligne: https://riptutorial.com/fr/wpf/topic/9115/system-windows-controls-webbrowser

Chapitre 23: Une introduction aux styles WPF

Introduction

Un style permet la modification complète de l'apparence visuelle d'un contrôle WPF. Voici quelques exemples de styles de base et une introduction aux dictionnaires de ressources et à l'animation.

Examples

Styling d'un bouton

Le moyen le plus simple de créer un style consiste à copier un style existant et à le modifier.

Créez une fenêtre simple avec deux boutons:

```
<Window x:Class="WPF_Style_Example.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" ResizeMode="NoResize"
    Title="MainWindow"
    Height="150" Width="200">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Margin="5" Content="Button 1"/>
    <Button Margin="5" Grid.Row="1" Content="Button 2"/>
</Grid>
```

Dans Visual Studio, la copie peut être effectuée en cliquant avec le bouton droit sur le premier bouton de l'éditeur et en choisissant "Modifier une copie ..." dans le menu "Modifier le modèle".

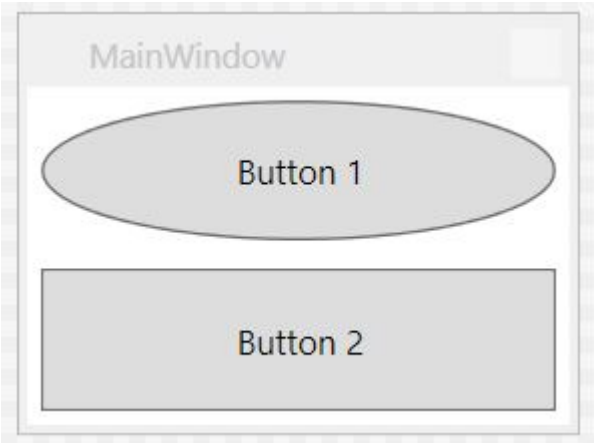
Définir dans "Application".

L'exemple suivant montre un modèle modifié pour créer un bouton ellipse:

```
<Style x:Key="ButtonStyle1" TargetType="{x:Type Button}">
        <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual}"/>
        <Setter Property="Background" Value="{StaticResource Button.Static.Background}"/>
        <Setter Property="BorderBrush" Value="{StaticResource Button.Static.Border}"/>
        <Setter Property="Foreground" Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}"/>
        <Setter Property="BorderThickness" Value="1"/>
        <Setter Property="HorizontalContentAlignment" Value="Center"/>
        <Setter Property="VerticalContentAlignment" Value="Center"/>
        <Setter Property="Padding" Value="1"/>
        <Setter Property="Template">
```

```
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type Button}">
                    <Grid>
                        <Ellipse x:Name="ellipse" StrokeThickness="{TemplateBinding
BorderThickness}" Stroke="{TemplateBinding BorderBrush}" Fill="{TemplateBinding Background}"
SnapsToDevicePixels="true"/>
                        <ContentPresenter x:Name="contentPresenter" Focusable="False"
HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}" Margin="{TemplateBinding
Padding}" RecognizesAccessKey="True" SnapsToDevicePixels="{TemplateBinding
SnapsToDevicePixels}" VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
                    </Grid>
                    <ControlTemplate.Triggers>
                        <Trigger Property="IsDefaulted" Value="true">
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
                        </Trigger>
                        <Trigger Property="IsMouseOver" Value="true">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.MouseOver.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.MouseOver.Border}"/>
                        </Trigger>
                        <Trigger Property="IsPressed" Value="true">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.Pressed.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.Pressed.Border}"/>
                        </Trigger>
                        <Trigger Property="IsEnabled" Value="false">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.Disabled.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.Disabled.Border}"/>
                            <Setter Property="TextElement.Foreground"
TargetName="contentPresenter" Value="{StaticResource Button.Disabled.Foreground}"/>
                        </Trigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

Le résultat:



Style appliqué à tous les boutons

En prenant l'exemple précédent, la suppression de l'élément x: Key du style applique le style à tous les boutons de la portée de l'application.

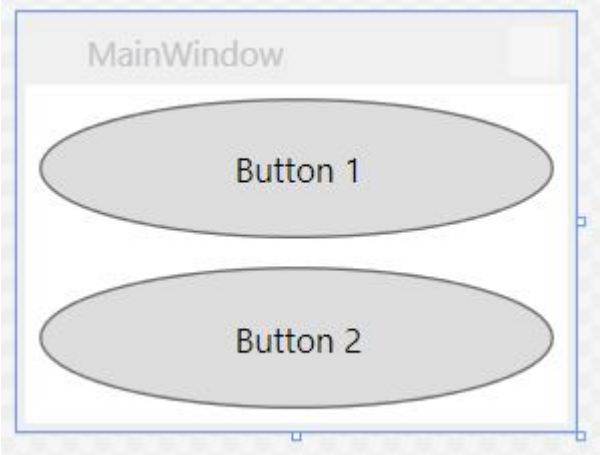
```
<Style TargetType="{x:Type Button}">
        <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual}"/>
        <Setter Property="Background" Value="{StaticResource Button.Static.Background}"/>
        <Setter Property="BorderBrush" Value="{StaticResource Button.Static.Border}"/>
        <Setter Property="Foreground" Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}"/>
        <Setter Property="BorderThickness" Value="1"/>
        <Setter Property="HorizontalContentAlignment" Value="Center"/>
        <Setter Property="VerticalContentAlignment" Value="Center"/>
        <Setter Property="Padding" Value="1"/>
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type Button}">
                    <Grid>
                        <Ellipse x:Name="ellipse" StrokeThickness="{TemplateBinding
BorderThickness}" Stroke="{TemplateBinding BorderBrush}" Fill="{TemplateBinding Background}"
SnapsToDevicePixels="true"/>
                        <ContentPresenter x:Name="contentPresenter" Focusable="False"
HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}" Margin="{TemplateBinding
Padding}" RecognizesAccessKey="True" SnapsToDevicePixels="{TemplateBinding
SnapsToDevicePixels}" VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
                    </Grid>
                    <ControlTemplate.Triggers>
                        <Trigger Property="IsDefaulted" Value="true">
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
                        </Trigger>
                        <Trigger Property="IsMouseOver" Value="true">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.MouseOver.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.MouseOver.Border}"/>
                        </Trigger>
                        <Trigger Property="IsPressed" Value="true">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.Pressed.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.Pressed.Border}"/>
                        </Trigger>
                        <Trigger Property="IsEnabled" Value="false">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.Disabled.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.Disabled.Border}"/>
                            <Setter Property="TextElement.Foreground"
TargetName="contentPresenter" Value="{StaticResource Button.Disabled.Foreground}"/>
                        </Trigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

Notez que le style ne doit plus être spécifié pour les boutons individuels:

```
<Window x:Class="WPF_Style_Example.MainWindow"
```

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" ResizeMode="NoResize"
    Title="MainWindow"
    Height="150" Width="200">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Margin="5" Content="Button 1"/>
    <Button Margin="5" Grid.Row="1" Content="Button 2"/>
</Grid>
```

Les deux boutons sont maintenant stylés.



Créer un ComboBox

En commençant par les `ComboBox` suivants:

```
<Window x:Class="WPF_Style_Example.ComboBoxWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" ResizeMode="NoResize"
    Title="ComboBoxWindow"
    Height="100" Width="150">
<StackPanel>
    <ComboBox Margin="5" SelectedIndex="0">
        <ComboBoxItem Content="Item A"/>
        <ComboBoxItem Content="Item B"/>
        <ComboBoxItem Content="Item C"/>
    </ComboBox>
    <ComboBox IsEditable="True" Margin="5" SelectedIndex="0">
        <ComboBoxItem Content="Item 1"/>
        <ComboBoxItem Content="Item 2"/>
        <ComboBoxItem Content="Item 3"/>
    </ComboBox>
</StackPanel>
```

Cliquez avec le bouton droit sur le premier `ComboBox` dans le concepteur, choisissez "Modifier le modèle -> Modifier une copie". Définissez le style dans la portée de l'application.

Il y a 3 styles créés:

```
ComboBoxToggleButton
ComboBoxEditableTextBox
ComboBoxStyle1
```

Et 2 modèles:

```
ComboBoxTemplate
ComboBoxEditableTemplate
```

Exemple de modification du style `ComboBoxToggleButton` :

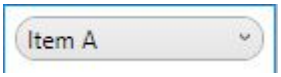
```
<SolidColorBrush x:Key="ComboBox.Static.Border" Color="#FFACACAC"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Border" Color="#FFABADB3"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Button.Background" Color="Transparent"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Button.Border" Color="Transparent"/>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Glyph" Color="#FF000000"/>
    <LinearGradientBrush x:Key="ComboBox.MouseOver.Background" EndPoint="0,1"
StartPoint="0,0">
        <GradientStop Color="Orange" Offset="0.0"/>
        <GradientStop Color="OrangeRed" Offset="1.0"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Border" Color="Red"/>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Editable.Border" Color="#FF7EB4EA"/>
    <LinearGradientBrush x:Key="ComboBox.MouseOver.Editable.Button.Background" EndPoint="0,1"
StartPoint="0,0">
        <GradientStop Color="#FFEBF4FC" Offset="0.0"/>
        <GradientStop Color="#FFDCECFC" Offset="1.0"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Editable.Button.Border" Color="#FF7EB4EA"/>
    <SolidColorBrush x:Key="ComboBox.Pressed.Glyph" Color="#FF000000"/>
    <LinearGradientBrush x:Key="ComboBox.Pressed.Background" EndPoint="0,1" StartPoint="0,0">
        <GradientStop Color="OrangeRed" Offset="0.0"/>
        <GradientStop Color="Red" Offset="1.0"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ComboBox.Pressed.Border" Color="DarkRed"/>
    <SolidColorBrush x:Key="ComboBox.Pressed.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.Pressed.Editable.Border" Color="#FF569DE5"/>
    <LinearGradientBrush x:Key="ComboBox.Pressed.Editable.Button.Background" EndPoint="0,1"
StartPoint="0,0">
        <GradientStop Color="#FFDAEBFC" Offset="0.0"/>
        <GradientStop Color="#FFC4E0FC" Offset="1.0"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ComboBox.Pressed.Editable.Button.Border" Color="#FF569DE5"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Glyph" Color="#FFBFBFBF"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Background" Color="#FFF0F0F0"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Border" Color="#FFD9D9D9"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Editable.Border" Color="#FFBFBFBF"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Editable.Button.Background"
Color="Transparent"/>
```

```
    <SolidColorBrush x:Key="ComboBox.Disabled.Editable.Button.Border" Color="Transparent"/>
    <SolidColorBrush x:Key="ComboBox.Static.Glyph" Color="#FF606060"/>
    <Style x:Key="ComboBoxToggleButton" TargetType="{x:Type ToggleButton}">
        <Setter Property="OverridesDefaultStyle" Value="true"/>
        <Setter Property="IsTabStop" Value="false"/>
        <Setter Property="Focusable" Value="false"/>
        <Setter Property="ClickMode" Value="Press"/>
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type ToggleButton}">
                    <Border x:Name="templateRoot" CornerRadius="10"
BorderBrush="{StaticResource ComboBox.Static.Border}" BorderThickness="{TemplateBinding
BorderThickness}" Background="{StaticResource ComboBox.Static.Background}"
SnapsToDevicePixels="true">
                        <Border x:Name="splitBorder" BorderBrush="Transparent"
BorderThickness="1" HorizontalAlignment="Right" Margin="0" SnapsToDevicePixels="true"
Width="{DynamicResource {x:Static SystemParameters.VerticalScrollBarWidthKey}}">
                            <Path x:Name="arrow" Data="F1 M 0,0 L 2.667,2.66665 L 5.3334,0 L
5.3334,-1.78168 L 2.6667,0.88501 L0,-1.78168 L0,0 Z" Fill="{StaticResource
ComboBox.Static.Glyph}" HorizontalAlignment="Center" Margin="0" VerticalAlignment="Center"/>
                        </Border>
                    </Border>
                    <ControlTemplate.Triggers>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="true"/>
                                <Condition Binding="{Binding IsMouseOver,
RelativeSource={RelativeSource Self}}" Value="false"/>
                                <Condition Binding="{Binding IsPressed,
RelativeSource={RelativeSource Self}}" Value="false"/>
                                <Condition Binding="{Binding IsEnabled,
RelativeSource={RelativeSource Self}}" Value="true"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Static.Editable.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Static.Editable.Border}"/>
                            <Setter Property="Background" TargetName="splitBorder"
Value="{StaticResource ComboBox.Static.Editable.Button.Background}"/>
                            <Setter Property="BorderBrush" TargetName="splitBorder"
Value="{StaticResource ComboBox.Static.Editable.Button.Border}"/>
                        </MultiDataTrigger>
                        <Trigger Property="IsMouseOver" Value="true">
                            <Setter Property="BorderThickness" TargetName="templateRoot"
Value="2"/>
                        </Trigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsMouseOver,
RelativeSource={RelativeSource Self}}" Value="true"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="false"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.MouseOver.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.MouseOver.Border}"/>
                        </MultiDataTrigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
```

```
                                <Condition Binding="{Binding IsMouseOver,
RelativeSource={RelativeSource Self}}" Value="true"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="true"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.MouseOver.Editable.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.MouseOver.Editable.Border}"/>
                            <Setter Property="Background" TargetName="splitBorder"
Value="{StaticResource ComboBox.MouseOver.Editable.Button.Background}"/>
                            <Setter Property="BorderBrush" TargetName="splitBorder"
Value="{StaticResource ComboBox.MouseOver.Editable.Button.Border}"/>
                        </MultiDataTrigger>
                        <Trigger Property="IsPressed" Value="true">
                            <Setter Property="Fill" TargetName="arrow" Value="{StaticResource
ComboBox.Pressed.Glyph}"/>
                        </Trigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsPressed,
RelativeSource={RelativeSource Self}}" Value="true"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="false"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Pressed.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Pressed.Border}"/>
                        </MultiDataTrigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsPressed,
RelativeSource={RelativeSource Self}}" Value="true"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="true"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Pressed.Editable.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Pressed.Editable.Border}"/>
                            <Setter Property="Background" TargetName="splitBorder"
Value="{StaticResource ComboBox.Pressed.Editable.Button.Background}"/>
                            <Setter Property="BorderBrush" TargetName="splitBorder"
Value="{StaticResource ComboBox.Pressed.Editable.Button.Border}"/>
                        </MultiDataTrigger>
                        <Trigger Property="IsEnabled" Value="false">
                            <Setter Property="Fill" TargetName="arrow" Value="{StaticResource
ComboBox.Disabled.Glyph}"/>
                        </Trigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsEnabled,
RelativeSource={RelativeSource Self}}" Value="false"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="false"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Disabled.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Disabled.Border}"/>
```

```
                        </MultiDataTrigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsEnabled,
RelativeSource={RelativeSource Self}}" Value="false"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="true"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Disabled.Editable.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Disabled.Editable.Border}"/>
                            <Setter Property="Background" TargetName="splitBorder"
Value="{StaticResource ComboBox.Disabled.Editable.Button.Background}"/>
                            <Setter Property="BorderBrush" TargetName="splitBorder"
Value="{StaticResource ComboBox.Disabled.Editable.Button.Border}"/>
                        </MultiDataTrigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

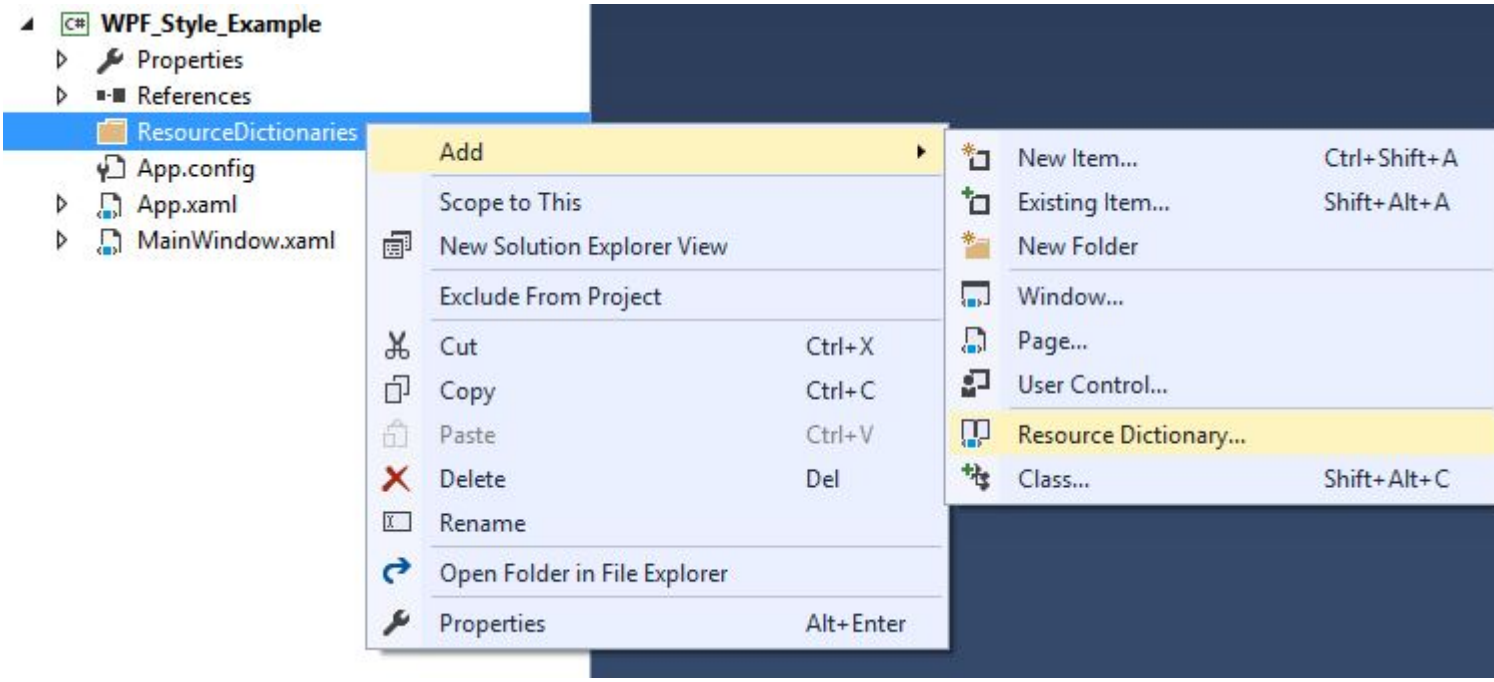
Cela crée un `ComboBox` arrondi qui met en surbrillance orange au passage de la souris et devient rouge lorsque vous appuyez dessus.



Notez que cela ne changera pas la liste déroulante modifiable en dessous; modification qui nécessite de modifier le style `ComboBoxEditableTextBox` ou le `ComboBoxEditableTemplate` .

Création d'un dictionnaire de ressources

Avoir beaucoup de styles dans App.xaml deviendra rapidement complexe, de sorte qu'ils peuvent être placés dans des dictionnaires de ressources séparés.



Pour utiliser le dictionnaire, il doit être fusionné avec App.xaml. Ainsi, dans App.xaml, après la création du dictionnaire de ressources:

```
<Application
         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
         x:Class="WPF_Style_Example.App"
         StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="ResourceDictionaries/Dictionary1.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

De nouveaux styles peuvent maintenant être créés dans Dictionary1.xaml et ils peuvent être référencés comme s'ils étaient dans App.xaml. Après avoir créé le projet, l'option apparaît également dans Visual Studio lors de la copie d'un style pour le localiser dans le nouveau dictionnaire de ressources.

Bouton Style DoubleAnimation

La `Window` suivante a été créée:

```
<Window x:Class="WPF_Style_Example.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" ResizeMode="NoResize"
    Title="MainWindow"
    Height="150" Width="250">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Margin="5" Content="Button 1" Width="200"/>
    <Button Margin="5" Grid.Row="1" Content="Button 2" Width="200"/>
</Grid>
```

Un style (créé dans App.xaml) a été appliqué aux boutons, qui anime la largeur de 200 à 100 lorsque la souris entre dans le contrôle et de 100 à 200 quand elle quitte:

```
<Style TargetType="{x:Type Button}">
    <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual}"/>
    <Setter Property="Background" Value="{StaticResource Button.Static.Background}"/>
    <Setter Property="BorderBrush" Value="{StaticResource Button.Static.Border}"/>
    <Setter Property="Foreground" Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}"/>
    <Setter Property="BorderThickness" Value="1"/>
    <Setter Property="HorizontalContentAlignment" Value="Center"/>
    <Setter Property="VerticalContentAlignment" Value="Center"/>
```

```
    <Setter Property="Padding" Value="1"/>
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type Button}">
                <Grid Background="White">
                    <Border x:Name="border" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" Background="{TemplateBinding Background}"
SnapsToDevicePixels="true">
                        <ContentPresenter x:Name="contentPresenter" Focusable="False"
HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}" Margin="{TemplateBinding
Padding}" RecognizesAccessKey="True" SnapsToDevicePixels="{TemplateBinding
SnapsToDevicePixels}" VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
                    </Border>
                </Grid>
                <ControlTemplate.Triggers>
                    <EventTrigger RoutedEvent="MouseEnter">
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation To="100" From="200"
Storyboard.TargetProperty="Width" Storyboard.TargetName="border" Duration="0:0:0.25"/>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger>
                    <EventTrigger RoutedEvent="MouseLeave">
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation To="200" From="100"
Storyboard.TargetProperty="Width" Storyboard.TargetName="border" Duration="0:0:0.25"/>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger>
                    <Trigger Property="IsDefaulted" Value="true">
                        <Setter Property="BorderBrush" TargetName="border"
Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
                    </Trigger>
                    <Trigger Property="IsMouseOver" Value="true">
                        <Setter Property="Background" TargetName="border"
Value="{StaticResource Button.MouseOver.Background}"/>
                        <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.MouseOver.Border}"/>
                    </Trigger>
                    <Trigger Property="IsPressed" Value="true">
                        <Setter Property="Background" TargetName="border"
Value="{StaticResource Button.Pressed.Background}"/>
                        <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.Pressed.Border}"/>
                    </Trigger>
                    <Trigger Property="IsEnabled" Value="false">
                        <Setter Property="Background" TargetName="border"
Value="{StaticResource Button.Disabled.Background}"/>
                        <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.Disabled.Border}"/>
                        <Setter Property="TextElement.Foreground"
TargetName="contentPresenter" Value="{StaticResource Button.Disabled.Foreground}"/>
                    </Trigger>
                </ControlTemplate.Triggers>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Lire Une introduction aux styles WPF en ligne: https://riptutorial.com/fr/wpf/topic/9670/une-introduction-aux-styles-wpf

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec wpf	Community, Derpcode, Gusdor, Matthew Cargille, Nasreddine, Sam, Stephen Wilson
2	Affinité des threads Accès aux éléments de l'interface utilisateur	Mert Gülsoy
3	Architecture WPF	Adi Lester
4	Comportements WPF	Bradley Uffner
5	Contrôle de la grille	Alexander Mandt, vkluge
6	Convertisseurs de valeur et de valeurs multiples	Adi Lester, Arie, Dalstroem, galakt, ltiveron
7	Création d'un écran de démarrage dans WPF	Grx70, Sam
8	Création de UserControls personnalisés avec liaison de données	ltiveron, Mage Xy
9	Déclencheurs	John Strit, Maxim
10	Extensions de balisage	Alexander Pacha, Emad
11	Introduction à la liaison de données WPF	Adi Lester, Arie, Gabor Barat, Guttsy, Ian Wold, Jirajha, vkluge, wkl
12	Localisation WPF	Dabblernl
13	MVVM dans WPF	Andrew Stephens, Athafoud, Dutts, Felix D., Felix Too, H.B., James LaPenn, Kcvin, kowsky, Matt Klein, RamenChef, STiLeTT, TrBBoI, vkluge
14	Optimisation pour l'interaction tactile	Martin Zikmund
15	Principe de conception "Half the Whitespace"	Richardissimo
16	Prise en charge de la diffusion vidéo en continu et de l'affectation des tableaux de pixels à un contrôle d'image	Eyal Perry

17	Propriétés de dépendance	Adi Lester, auticus, Clemens, Guttsy
18	Ressources WPF	Elangovan, John Strit, SUB-HDR
19	Slider Binding: mise à jour uniquement sur Drag Ended	Eyal Perry
20	Styles dans WPF	Guttsy, Jakub Lokša
21	Synthèse de discours	BKO
22	System.Windows.Controls.WebBrowser	Richardissimo
23	Une introduction aux styles WPF	J R