EBook Gratis

APRENDIZAJE wpf

Free unaffiliated eBook created from
Stack Overflow contributors.

#wpf

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con wpf ... 2

Observaciones ... 2

Versiones ... 2

Examples ... 2

Hola aplicación mundial ... 2

Capítulo 2: Afinidad de hilos que accede a los elementos de la interfaz de usuario ... 7

Examples ... 7

Acceder a un elemento UI desde dentro de una tarea ... 7

Capítulo 3: Arquitectura WPF ... 9

Examples ... 9

DispatcherObject ... 9

Deriva de ... 9

Miembros clave ... 9

Resumen ... 9

DependencyObject ... 9

Deriva de ... 9

Miembros clave ... 9

Resumen ... 9

Capítulo 4: Comportamientos de WPF ... 11

Introducción ... 11

Examples ... 11

Comportamiento simple para interceptar eventos de la rueda del ratón ... 11

Capítulo 5: Control de cuadrícula ... 13

Examples ... 13

Una cuadrícula simple ... 13

Niños de cuadrícula que abarcan varias filas / columnas ... 13

Sincronizando filas o columnas de múltiples grillas ... 13

Capítulo 6: Convertidores de Valor y Multivalor ... 15

Parámetros ... 15

Observaciones 15
Qué son IValueConverter e IMultiValueConverter 15
Para que sirven 15
Examples 15
Construido en BooleanToVisibilityConverter [IValueConverter] 16
Usando el convertidor 16
Convertidor con propiedad [IValueConverter] 17
Usando el convertidor 18
Convertidor simple de agregar [IMultiValueConverter] 18
Usando el convertidor 19
Convertidores de uso con ConverterParameter 19
Usando el convertidor 20
Grupo de convertidores multiples [IValueConverter] 20
Uso de MarkupExtension con convertidores para omitir la declaración de recurso 21
Use IMultiValueConverter para pasar múltiples parámetros a un comando 22
Capítulo 7: Creación de UserControls personalizados con enlace de datos 24
Observaciones 24
Examples 24
ComboBox con texto predeterminado personalizado 24
Capítulo 8: Creando la pantalla de bienvenida en WPF 28
Introducción 28
Examples 28
Añadiendo pantalla de bienvenida simple 28
Prueba de pantalla de bienvenida 30
Creando ventana de pantalla de bienvenida personalizada 32
Creación de la ventana de la pantalla de bienvenida con informes de progreso 33
Capítulo 9: Enlace de barra deslizante: Actualizar solo en arrastre finalizado 36
Parámetros 36
Observaciones 36
Examples 36
Implementación del comportamiento 36
Uso de XAML 37

Capítulo 10: Estilos en WPF 38
Observaciones 38
Notas introductorias 38
Notas importantes 38
Recursos 38
Examples 39
Definiendo un estilo nombrado 39
Definiendo un estilo implícito 39
Heredando de un estilo 39
Capítulo 11: Extensiones de marcado 41
Parámetros 41
Observaciones 41
Examples 41
Extensión de marcado utilizada con IValueConverter 41
Extensiones de marcado definidas por XAML 42
Capítulo 12: Gatillos 44
Introducción 44
Observaciones 44
Examples 44
Desencadenar 44
MultiTrigger 45
Data Trigger 45
Capítulo 13: Introducción al enlace de datos WPF 47
Sintaxis 47
Parámetros 47
Observaciones 48
UpdateSourceTrigger 48
Examples 48
Convertir un valor booleano a visibilidad 48
Definiendo el DataContext 49
Implementando INotifyPropertyChanged 50

Vincular a propiedad de otro elemento nombrado ... 51
Vincular a la propiedad de un antepasado ... 51
Vinculando múltiples valores con un MultiBinding ... 51
Capítulo 14: Localización WPF ... 53
Observaciones ... 53
Examples ... 53
XAML para VB ... 53
Propiedades para el archivo de recursos en VB ... 53
XAML para C # ... 54
Hacer públicos los recursos ... 54
Capítulo 15: MVVM en WPF ... 56
Observaciones ... 56
Examples ... 56
Ejemplo básico de MVVM usando WPF y C # ... 56
El modelo de vista ... 59
El modelo ... 61
La vista ... 62
Comandando en MVVM ... 64
Capítulo 16: Optimización para la interacción táctil ... 67
Examples ... 67
Mostrando teclado táctil en Windows 8 y Windows 10 ... 67
Aplicaciones WPF dirigidas a .NET Framework 4.6.2 y posteriores ... 67
Aplicaciones WPF dirigidas a .NET Framework 4.6.1 y anteriores ... 67
Solución ... 68
Nota sobre el modo tableta en Windows 10 ... 69
Enfoque de configuración de Windows 10 ... 69
Capítulo 17: Principio de diseño "La mitad del espacio en blanco" ... 71
Introducción ... 71
Examples ... 71
Demostración del problema y la solución ... 71
Cómo usar esto en código real ... 76
Capítulo 18: Propiedades de dependencia ... 77

Introducción ... 77
Sintaxis ... 77
Parámetros ... 77
Examples ... 78
Propiedades de dependencia estándar ... 78
Cuándo usar ... 78
Como definir ... 78
Convenciones importantes ... 79
Modo de encuadernación ... 79
Propiedades de dependencia adjuntas ... 80
Cuándo usar ... 80
Como definir ... 80
Advertencias ... 81
Propiedades de dependencia de solo lectura ... 81
Cuándo usar ... 81
Como definir ... 81
Capítulo 19: Recursos WPF ... 83
Examples ... 83
Hola Recursos ... 83
Tipos de recursos ... 83
Recursos amplios locales y de aplicación ... 84
Recursos de Code-behind ... 85
Capítulo 20: Síntesis del habla ... 88
Introducción ... 88
Sintaxis ... 88
Examples ... 88
Ejemplo de síntesis de voz - Hola mundo ... 88
Capítulo 21: Soporta transmisión de video y asignación de píxeles a un control de imagen ... 89
Parámetros ... 89
Observaciones ... 89
Examples ... 89

Implementación del comportamiento ... 90
Uso de XAML ... 94
Capítulo 22: System.Windows.Controls.WebBrowser ... 96
Introducción ... 96
Observaciones ... 96
Examples ... 96
Ejemplo de un WebBrowser dentro de un BusyIndicator ... 96
Capítulo 23: Una introducción a los estilos WPF ... 97
Introducción ... 97
Examples ... 97
Estilo de un botón ... 97
Estilo aplicado a todos los botones ... 99
Diseñando un ComboBox ... 100
Creando un Diccionario de Recursos ... 104
Estilo de botón DoubleAnimation ... 105
Creditos ... 108

Acerca de

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

Capítulo 1: Empezando con wpf

Observaciones

WPF (Windows Presentation Foundation) es la tecnología de presentación recomendada por Microsoft para las aplicaciones clásicas de escritorio de Windows. WPF no debe confundirse con UWP (plataforma universal de Windows) aunque existen similitudes entre los dos.

WPF fomenta las aplicaciones basadas en datos con un fuerte enfoque en multimedia, animación y enlace de datos. Las interfaces se crean utilizando un lenguaje llamado XAML (lenguaje de marcado de aplicaciones extensible), un derivado de XML. XAML ayuda a los programadores de WPF a mantener la separación entre el diseño visual y la lógica de la interfaz.

A diferencia de su Windows Forms predecesor, WPF usa un modelo de caja para diseñar todos los elementos de la interfaz. Cada elemento tiene una altura, anchura y márgenes y se organiza en la pantalla en relación con su padre.

WPF significa Windows Presentation Foundation y también es conocido bajo su nombre en clave Avalon. Es un marco gráfico y parte de Microsofts .NET Framework. WPF está preinstalado en Windows Vista, 7, 8 y 10 y puede instalarse en Windows XP y Server 2003.

Versiones

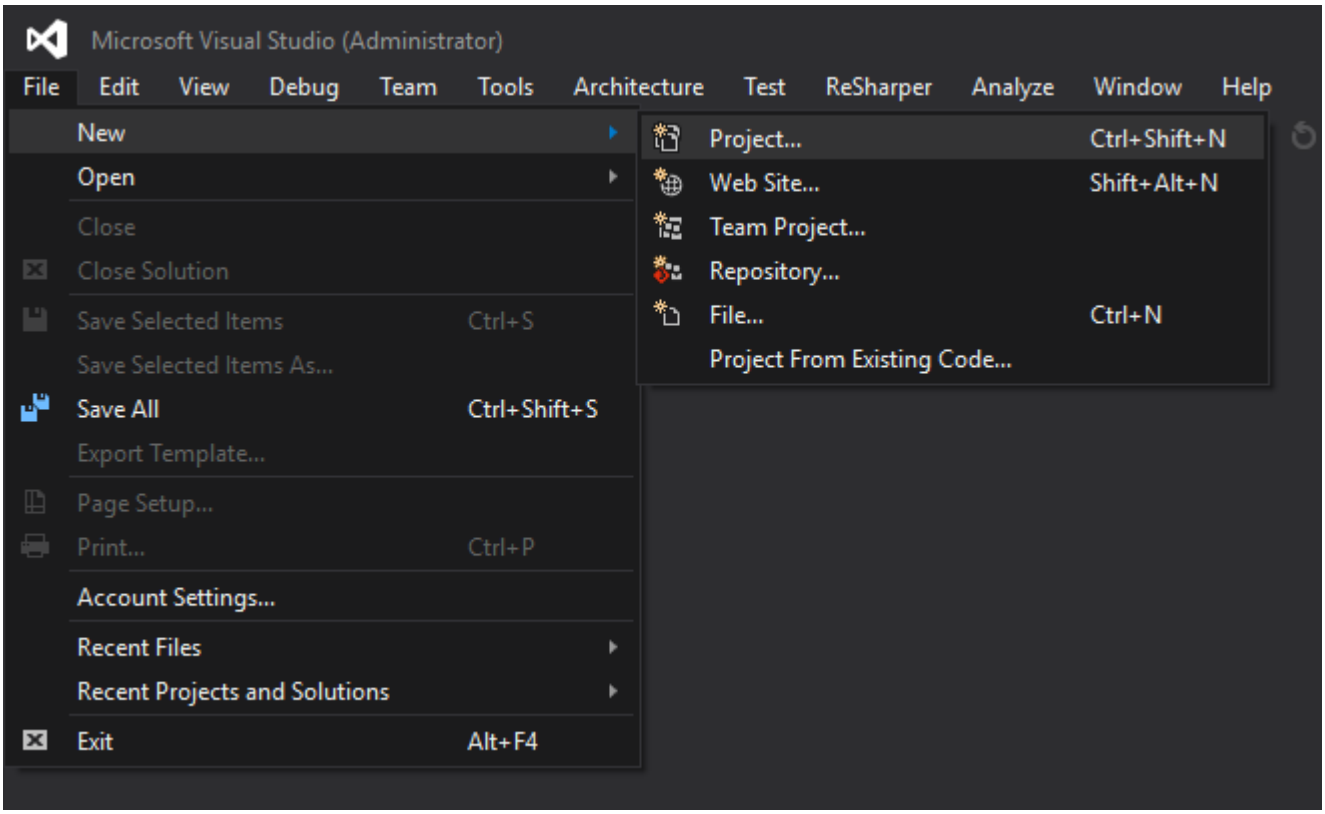
Versión 4.6.1 - Diciembre 2015

Examples

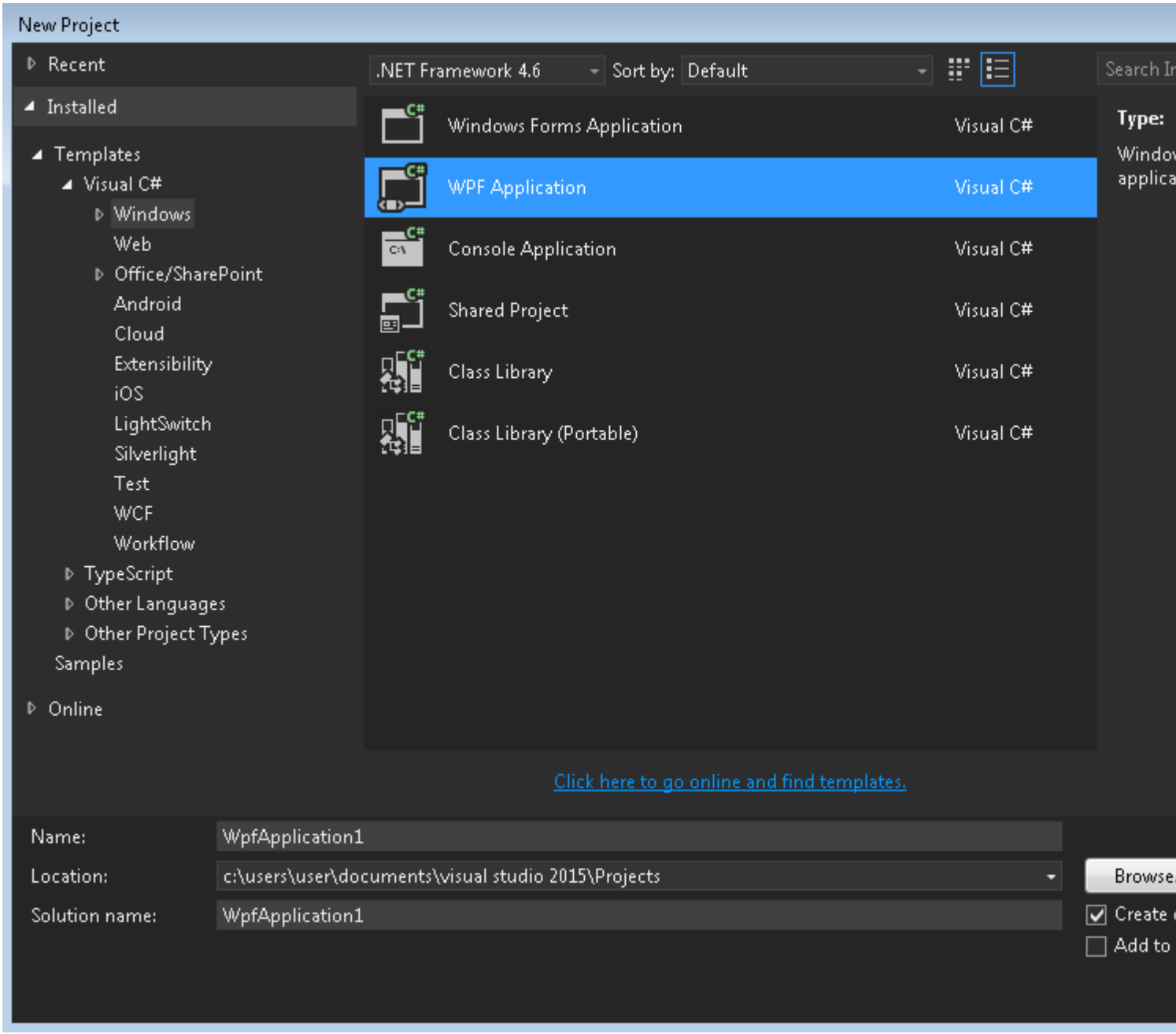
Hola aplicación mundial

Para crear y ejecutar un nuevo proyecto WPF en Visual Studio:

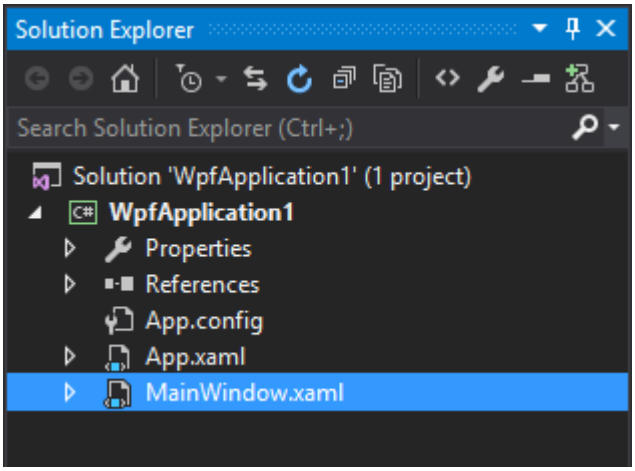
1. Haga clic en **Archivo → Nuevo → Proyecto**



2. Seleccione la plantilla haciendo clic en **Plantillas → Visual C # → Windows → Aplicación WPF** y presione `Aceptar` :



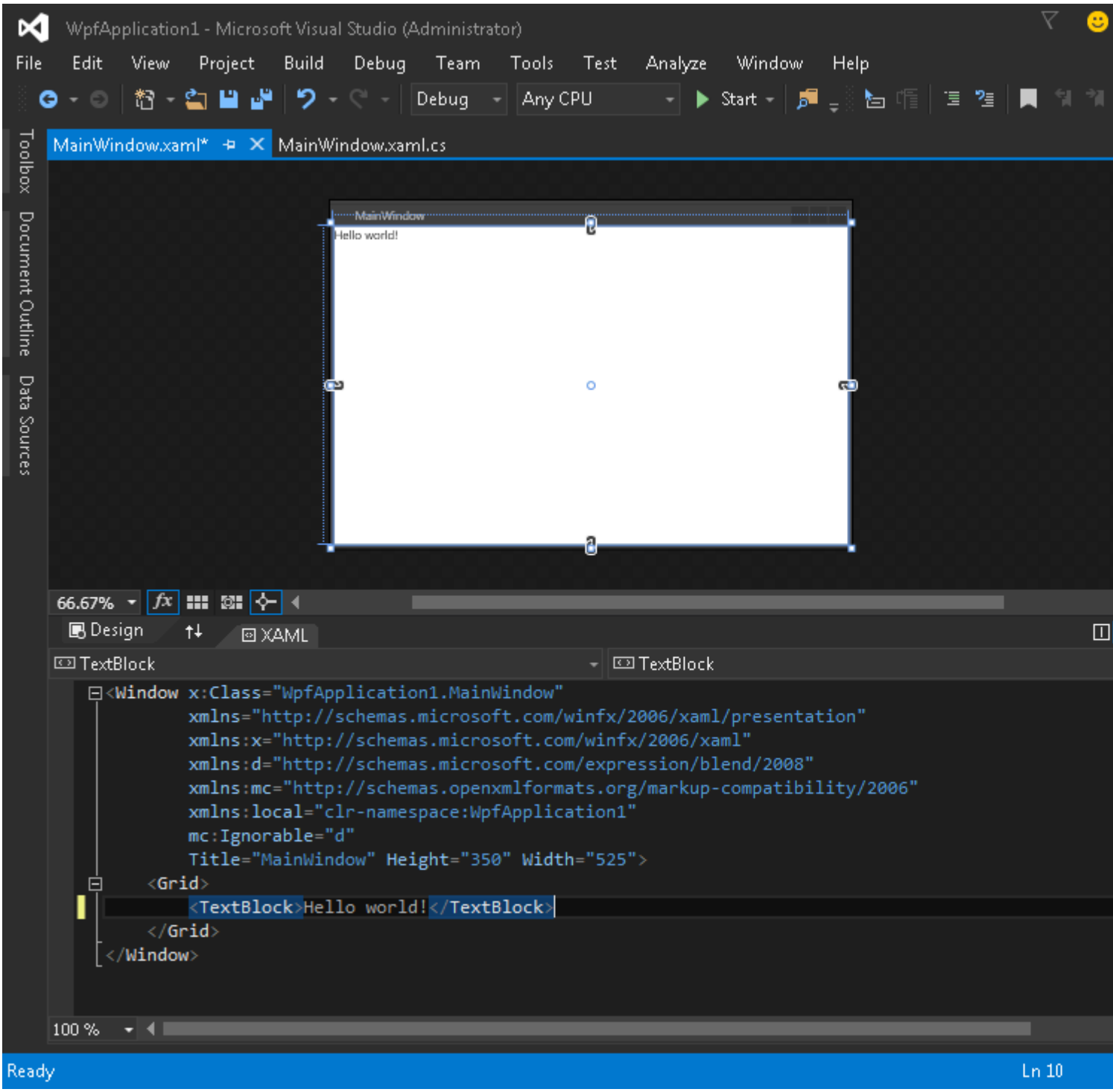
3. Abra el archivo **MainWindow.xaml** en el *Explorador de soluciones* (si no ve la ventana del *Explorador de soluciones* , ábralo haciendo clic en **Ver → Explorador de soluciones**):



4. En la sección *XAML* (por defecto debajo de la sección *Diseño*) agregue este código

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

dentro de la etiqueta de la `Grid` :

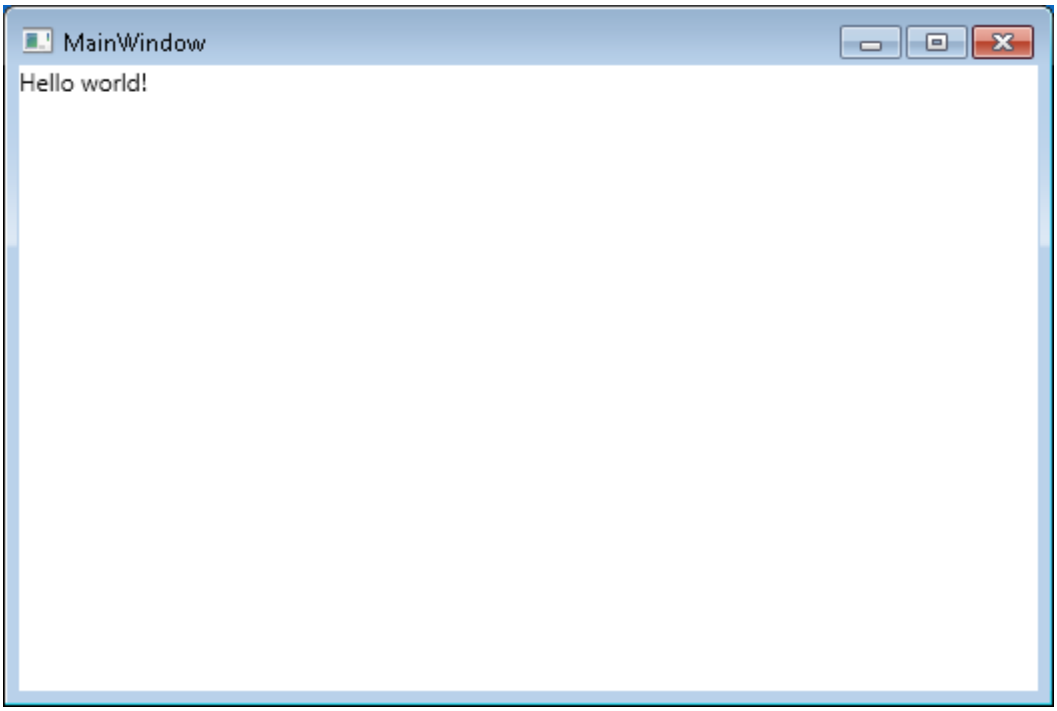


El código debería verse como:

```
<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. Ejecute la aplicación presionando `F5` o haciendo clic en el menú **Depurar → Iniciar depuración** . Debería verse como



Lea Empezando con wpf en línea: https://riptutorial.com/es/wpf/topic/820/empezando-con-wpf

Capítulo 2: Afinidad de hilos que accede a los elementos de la interfaz de usuario

Examples

Acceder a un elemento UI desde dentro de una tarea

Todos los elementos de la interfaz de usuario creados y residen en el hilo principal de un programa. Acceder a estos desde otro hilo está prohibido por el tiempo de ejecución de .net framework. Básicamente, se debe a que todos los elementos de la interfaz de usuario son **recursos sensibles a los subprocesos** y el acceso a un recurso en un entorno de subprocesos múltiples debe ser seguro para subprocesos. Si se permite este acceso a objetos de subprocesos cruzados, la consistencia se vería afectada en primer lugar.

Considere este escenario:

Tenemos un cálculo que sucede dentro de una tarea. Las tareas se ejecutan en otro hilo que el hilo principal. Mientras el cálculo continúa, necesitamos actualizar una barra de progreso. Para hacer esto:

```
//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);
```

Cada elemento de la interfaz de usuario tiene un objeto Dispatcher que proviene de su ancestro `DispatcherObject` (dentro del espacio de nombres `System.Windows.Threading`). Dispatcher ejecuta el delegado especificado de forma sincrónica en la prioridad especificada en el subproceso en el que está asociado el Dispatcher. Dado que la ejecución está sincronizada, la tarea del que llama debe esperar su resultado. Esto nos da la oportunidad de utilizar el `int progress` también dentro de un delegado de despacho.

Es posible que queramos actualizar un elemento de la interfaz de usuario de forma asíncrona y luego `invokeAction` cambios de definición de `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);
```

Esta vez empaquetamos el `int progress` y lo usamos como un parámetro para delegar.

Lea Afinidad de hilos que accede a los elementos de la interfaz de usuario en línea: https://riptutorial.com/es/wpf/topic/6128/afinidad-de-hilos-que-accede-a-los-elementos-de-la-interfaz-de-usuario

Capítulo 3: Arquitectura WPF

Examples

DispatcherObject

Deriva de

`Object`

Miembros clave

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

Resumen

La mayoría de los objetos en WPF se derivan de `DispatcherObject` , que proporciona las construcciones básicas para tratar la concurrencia y el subprocesamiento. Tales objetos están asociados con un Dispatcher.

Solo el hilo en el que se creó el Dispatcher puede acceder al DispatcherObject directamente. Para acceder a DispatcherObject desde un subproceso que no sea el subproceso en el que se creó DispatcherObject, se requiere una llamada a `Invoke` o `BeginInvoke` en el Dispatcher al que está asociado el objeto.

DependencyObject

Deriva de

`DispatcherObject`

Miembros clave

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

Resumen

Las clases derivadas de `DependencyObject` participan en el sistema de propiedades de dependencia , que incluye el registro de propiedades de dependencia y la identificación e información sobre dichas propiedades. Dado que las propiedades de dependencia son la piedra angular del desarrollo de WPF, todos los controles de WPF se derivan en última instancia de `DependencyObject`

.

Lea Arquitectura WPF en línea: https://riptutorial.com/es/wpf/topic/3571/arquitectura-wpf

Capítulo 4: Comportamientos de WPF

Introducción

Los comportamientos de WPF permiten que un desarrollador altere la forma en que WPF controla los actos en respuesta a los eventos del sistema y del usuario. Los comportamientos se heredan de la clase `Behavior` del espacio de nombres `System.Windows.Interactity` . Este espacio de nombres es una parte del SDK de Expression Blend general, pero una versión más liviana, adecuada para bibliotecas de comportamiento, está disponible como [paquete de nuget] [1]. [1]: https://www.nuget.org/packages/System.Windows.Interactivity.WPF/

Examples

Comportamiento simple para interceptar eventos de la rueda del ratón

Implementando el Comportamiento

Este comportamiento provocará que los eventos de la rueda del ratón desde un `ScrollViewer` interno `ScrollViewer` hasta el `ScrollViewer` principal cuando el interno esté en su límite superior o inferior. Sin este comportamiento, los eventos nunca saldrán del `ScrollViewer` interno.

```
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);
```

```
        }
    }
}
```

Comportamientos subclase la clase base de `Behavior<T>` , siendo `T` el tipo de control al que se puede adjuntar, en este caso `UIElement` . Cuando el `Behavior` se crea desde XAML, se `OnAttached` método `OnAttached` . Este método permite que el comportamiento se enganche a los eventos del control al que se adjunta (a través de `AssociatedControl`). Un método similar, se llama a `OnDetached` cuando el comportamiento debe ser desconectado del elemento asociado. Se debe tener cuidado de eliminar cualquier controlador de eventos, o de lo contrario limpiar objetos para evitar pérdidas de memoria.

Este comportamiento se engancha al evento `PreviewMouseWheel` , que le da un cambio para interceptar el evento antes de que `ScrollViewer` tenga la oportunidad de verlo. Verifica la posición para ver si necesita reenviar el evento hacia arriba del árbol visual a cualquier jerarquía superior de `ScrollViewer` . Si es así, establece `e.Handled` en `true` para evitar la acción predeterminada del evento. A continuación, genera un nuevo `MouseWheelEvent` enrutado a `AssociatedObject` . De lo contrario, el evento se enruta de forma normal.

Adjuntando el comportamiento a un elemento en XAML

Primero, el espacio de nombres xml de `interactivity` debe colocar en el ámbito antes de poder utilizarlo en XAML. Agregue la siguiente línea a los espacios de nombres de su XAML.

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

El comportamiento se puede adjuntar así:

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

Esto crea una colección de `Behaviors` como una propiedad adjunta en el `ScrollViewer` interno que contiene un comportamiento de `BubbleMouseWheelEvents` .

Este comportamiento en particular también podría adjuntarse a cualquier control existente que contenga un `ScrollViewer` incorporado, como un `GridView` , y aún funcionaría correctamente.

Lea Comportamientos de WPF en línea:
https://riptutorial.com/es/wpf/topic/8365/comportamientos-de-wpf

Capítulo 5: Control de cuadrícula

Examples

Una cuadrícula 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>
```

Las filas y columnas se definen agregando elementos `RowDefinition` y `ColumnDefinition` a las colecciones correspondientes.

Puede haber una gran cantidad de niños en la `Grid` . Para especificar en qué fila o columna se colocará un hijo en las propiedades `Grid.Column` se utilizan `Grid.Row` y `Grid.Column` . Los números de fila y columna se basan en cero. Si no se configura ninguna fila o columna, el valor predeterminado es `0` .

Los niños colocados en la misma fila y columna se dibujan en orden de definición. Por lo tanto, el último niño definido se dibujará sobre el niño definido anteriormente.

Niños de cuadrícula que abarcan varias filas / columnas

Al usar las propiedades adjuntas `Grid.RowSpan` y `Grid.ColumnSpan` , los hijos de una `Grid` pueden abarcar varias filas o columnas. En el siguiente ejemplo, el segundo `TextBlock` abarcará la segunda y la tercera columna 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>
```

Sincronizando filas o columnas de múltiples grillas

Los altos de fila o los anchos de columna de múltiples `Grid` s se pueden sincronizar configurando

un `SharedSizeGroup` común en las filas o columnas para sincronizar. Luego, un control primario en algún lugar del árbol sobre las `Grid` debe tener la propiedad adjunta `Grid.IsSharedSizeScope` establecida en `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>
```

En este ejemplo, la primera columna de ambas `Grid` siempre tendrá el mismo ancho, también cuando una de ellas cambie de tamaño por su contenido.

Lea Control de cuadrícula en línea: https://riptutorial.com/es/wpf/topic/6483/control-de-cuadricula

Capítulo 6: Convertidores de Valor y Multivalor

Parámetros

Parámetro	Detalles
valor	El valor producido por la fuente de enlace.
valores	La matriz de valores, producida por la fuente de enlace.
tipo de objetivo	El tipo de la propiedad objetivo de enlace.
parámetro	El parámetro convertidor a utilizar.
cultura	La cultura a utilizar en el convertidor.

Observaciones

Qué son IValueConverter e IMultiValueConverter

IValueConverter e IMultiValueConverter: interfaces que proporcionan una manera de aplicar una lógica personalizada a un enlace.

Para que sirven

1. Tiene algún valor de tipo pero desea mostrar valores cero de una manera y números positivos de otra manera
2. Tiene algún valor de tipo y desea mostrar el elemento en un caso y ocultarse en otro
3. Tienes un valor numérico de dinero pero quieres mostrarlo como palabras
4. Tiene un valor numérico pero desea mostrar imágenes diferentes para números diferentes

Estos son algunos de los casos simples, pero hay muchos más.

Para casos como este, puede usar un convertidor de valores. Estas clases pequeñas, que implementan la interfaz IValueConverter o IMultiValueConverter, actuarán como intermediarios y traducirán un valor entre la fuente y el destino. Por lo tanto, en cualquier situación en la que necesite transformar un valor antes de que llegue a su destino o vuelva a su origen, es probable que necesite un convertidor.

Examples

Construido en BooleanToVisibilityConverter [IValueConverter]

Convertidor entre booleanos y visibilidad. Obtener valor `bool` en la entrada y devuelve el valor de `Visibility` .

NOTA: Este convertidor ya existe en el espacio de nombres `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;
        }
    }
}
```

Usando el convertidor

1. Definir recurso

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

3. Utilízalo en encuadernación

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

Convertidor con propiedad [IValueConverter]

Muestre cómo crear un convertidor simple con parámetro a través de la propiedad y luego páselo en la declaración. Convertir el valor `bool` a la `Visibility` . Permite invertir el valor del resultado estableciendo la propiedad `Inverted` en `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;
    }
}
```

Usando el convertidor

1. Definir espacio de nombres

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

2. Definir recurso

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

3. Utilízalo en encuadernación

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

Convertidor simple de agregar [IMultiValueConverter]

Muestre cómo crear un simple convertidor de `IMultiValueConverter` y use `MultiBinding` en xaml. Obtener la suma de todos los valores pasados por la matriz 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();
```

```
    }
}
```

Usando el convertidor

1. Definir espacio de nombres

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

2. Definir recurso

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

3. Utilízalo en encuadernación

```
<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>
```

Convertidores de uso con ConverterParameter

Muestre cómo crear un convertidor simple y use `ConverterParameter` para pasar el parámetro al convertidor. Multiplique el valor por el coeficiente pasado en 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();
    }
}
```

Usando el convertidor

1. Definir espacio de nombres

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

2. Definir recurso

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

3. Utilízalo en encuadernación

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

Grupo de convertidores multiples [IValueConverter]

Este convertidor encadenará varios convertidores juntos.

```
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();
    }
}
```

En este ejemplo, el resultado booleano de `EnumToBooleanConverter` se usa como entrada en `BooleanToVisibilityConverter` .

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

El botón solo estará visible cuando la propiedad `CurrentMode` esté configurada en `Ready` .

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

Uso de MarkupExtension con convertidores para omitir la declaración de recurso

Normalmente para usar el convertidor, tenemos que definirlo como recurso de la siguiente manera:

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

Es posible omitir este paso definiendo un convertidor como `MarkupExtension` e implementando el método `ProvideValue` . El siguiente ejemplo convierte un valor a su negativo:

```
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;
    }
}
}
```

Usando el convertidor:

1. Definir espacio de nombres

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

2. Ejemplo de uso de este convertidor en enlace

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

Use IMultiValueConverter para pasar múltiples parámetros a un comando

Es posible pasar varios valores enlazados como un `CommandParameter` utilizando `MultiBinding` con un `IMultiValueConverter` muy 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()
        {
        }
    }
}
```

Usando el convertidor:

1. Ejemplo de implementación: método llamado cuando se ejecuta `SomeCommand` (*nota:* `DelegateCommand` *es una implementación de* `ICommand` *que no se proporciona en este ejemplo*):

```
 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. Definir espacio de nombres

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

3. Ejemplo de uso de este convertidor en enlace

```
<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>
```

Lea Convertidores de Valor y Multivalor en línea:
https://riptutorial.com/es/wpf/topic/3950/convertidores-de-valor-y-multivalor

Capítulo 7: Creación de UserControls personalizados con enlace de datos

Observaciones

Tenga en cuenta que un UserControl es muy diferente de un Control. Una de las principales diferencias es que un UserControl hace uso de un archivo de diseño XAML para determinar dónde colocar varios controles individuales. Un control, por otro lado, es solo código puro, no hay ningún archivo de diseño. De alguna manera, crear un Control personalizado puede ser más efectivo que crear un UserControl personalizado.

Examples

ComboBox con texto predeterminado personalizado

Este UserControl personalizado aparecerá como un cuadro combinado regular, pero a diferencia del objeto ComboBox incorporado, puede mostrar al usuario una cadena de texto predeterminada si aún no ha realizado una selección.

Para lograr esto, nuestro UserControl estará compuesto por dos controles. Obviamente necesitamos un ComboBox real, pero también usaremos una etiqueta regular para mostrar el texto predeterminado.

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>
```

Como puede ver, este único UserControl es en realidad un grupo de dos controles individuales.

Esto nos permite cierta flexibilidad que no está disponible en un solo ComboBox solo.

Aquí hay varias cosas importantes a tener en cuenta:

- El UserControl en sí tiene un `x:Name` establecido. Esto se debe a que queremos vincularnos a las propiedades que se encuentran en el código subyacente, lo que significa que necesita alguna forma de referenciarse a sí mismo.
- Cada uno de los enlaces en el ComboBox tiene el nombre de UserControl como `ElementName` . Esto es para que el UserControl sepa mirarse a sí mismo para localizar los enlaces.
- La etiqueta no es visible. Esto es para dar al usuario la ilusión de que la etiqueta es parte de ComboBox. Al establecer `IsHitTestVisible=false` , no permitimos que el usuario se `IsHitTestVisible=false` o haga clic en la etiqueta: todas las entradas se pasan a través del ComboBox a continuación.
- La etiqueta utiliza un convertidor de `InverseNullVisibility` para determinar si debe mostrarse o no. Puede encontrar el código para esto al final de este ejemplo.

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); }
    }
}
```

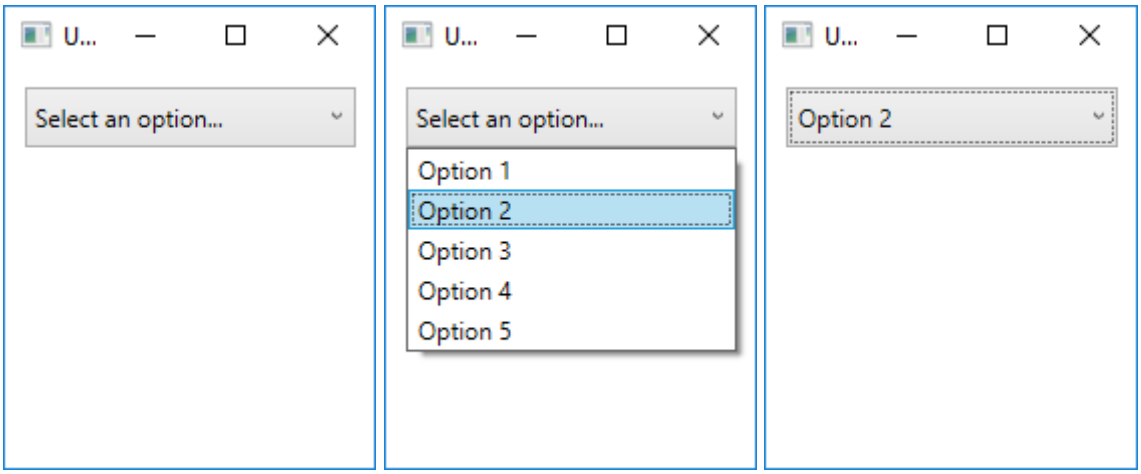
En el código subyacente, simplemente estamos exponiendo qué propiedades queremos que

estén disponibles para el programador utilizando este UserControl. Desafortunadamente, debido a que no tenemos acceso directo a la ComboBox desde fuera de esta clase, que necesitamos para mostrar las propiedades duplicadas (`MyItemsSource` para el cuadro combinado de `ItemsSource` , por ejemplo). Sin embargo, esta es una compensación menor considerando que ahora podemos usar esto de manera similar a un control nativo.

Aquí es cómo se puede utilizar el `CustomComboBox` :

```
<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>
```

Y el resultado final:



Aquí está el InverseNullVisibilityConverter necesario para la etiqueta en el UserControl, que es solo una pequeña variación en la versión 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();
```

```
    }
}
```

Lea Creación de UserControls personalizados con enlace de datos en línea: https://riptutorial.com/es/wpf/topic/6508/creacion-de-usercontrols-personalizados-con-enlace-de-datos

Capítulo 8: Creando la pantalla de bienvenida en WPF

Introducción

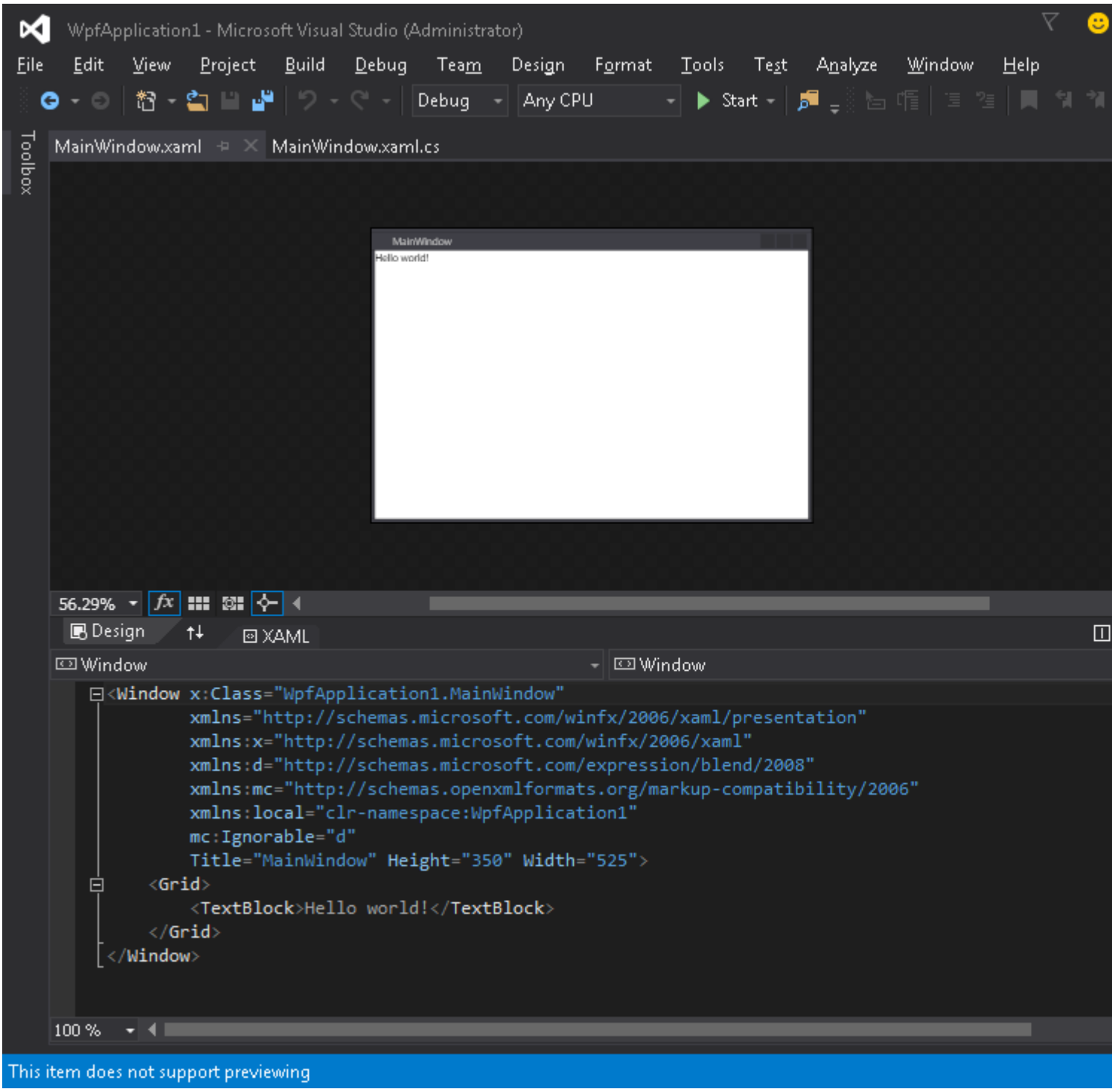
Cuando se inicia la aplicación WPF, el tiempo de ejecución de un idioma actual (CLR) puede tardar un tiempo en inicializar .NET Framework. Como resultado, la primera ventana de la aplicación puede aparecer algún tiempo después del inicio de la aplicación, dependiendo de la complejidad de la aplicación. La pantalla de bienvenida en WPF permite que la aplicación muestre imágenes estáticas o contenido dinámico personalizado durante la inicialización antes de que aparezca la primera ventana.

Examples

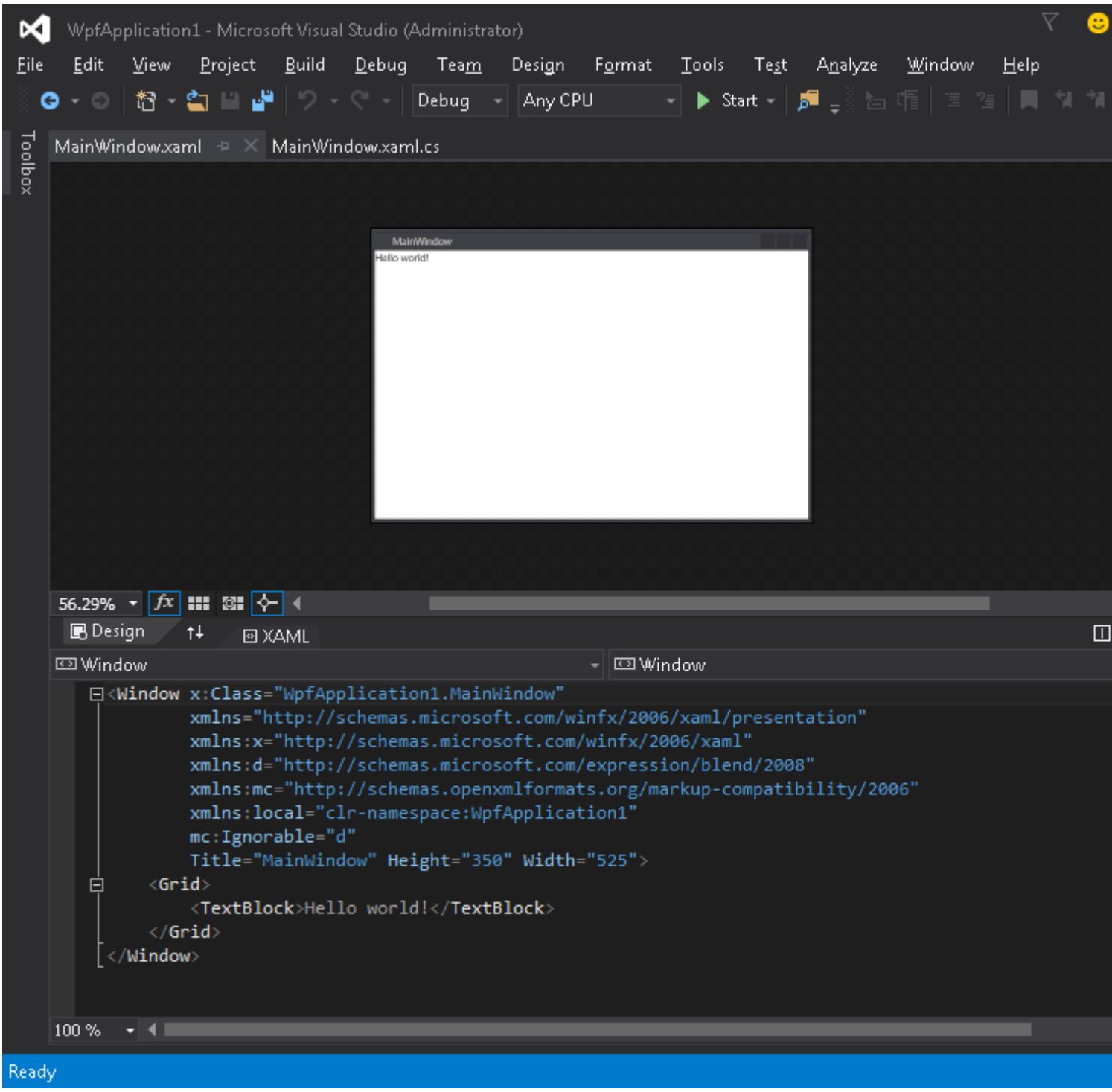
Añadiendo pantalla de bienvenida simple

Siga estos pasos para agregar la pantalla de bienvenida a la aplicación WPF en Visual Studio:

1. Cree u obtenga cualquier imagen y agréguela a su proyecto (por ejemplo, dentro de la carpeta *Imágenes*):



2. Abra la ventana de propiedades para esta imagen (**Ver → Ventana de propiedades**) y cambie la configuración de la **acción de compilación** al valor de **SplashScreen** :



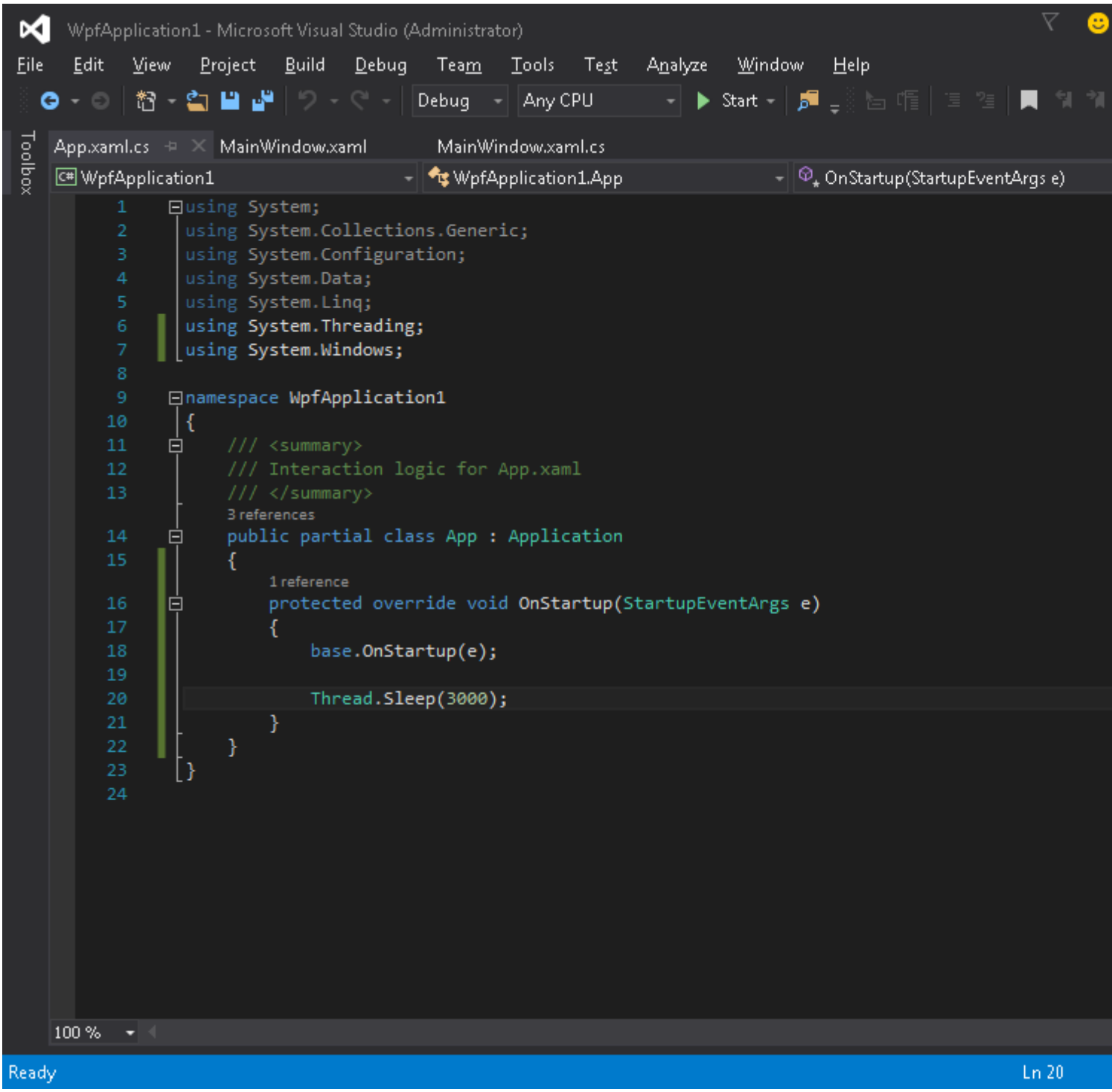
3. Ejecutar la aplicación. Verá la imagen de la pantalla de inicio en el centro de la pantalla antes de que aparezca la ventana de la aplicación (después de que aparezca la ventana, la imagen de la pantalla de inicio se desvanecerá en unos 300 milisegundos).

Prueba de pantalla de bienvenida

Si su aplicación es liviana y simple, se iniciará muy rápido y con una velocidad similar aparecerá y desaparecerá la pantalla de bienvenida.

Tan pronto como la pantalla de inicio desaparezca después de que se complete el método `Application.Startup` , puede simular el retraso de inicio de la aplicación siguiendo estos pasos:

1. Abrir el archivo **App.xaml.cs**
2. Agregar *usando el* espacio de nombres `using System.Threading;`
3. `OnStartup` método `OnStartup` y agregue `Thread.Sleep(3000);` dentro de eso:



El código debería verse como:

```
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. Ejecutar la aplicación. Ahora se iniciará durante unos 3 segundos más, por lo que tendrá más tiempo para probar su pantalla de inicio.

Creando ventana de pantalla de bienvenida personalizada

WPF no admite la visualización de nada más que una imagen como una pantalla de inicio, así que tendremos que crear una `Window` que servirá como pantalla de inicio. Suponemos que ya hemos creado un proyecto que contiene la clase `MainWindow` , que será la ventana principal de la aplicación.

En primer lugar, agregamos una ventana `SplashScreenWindow` a nuestro proyecto:

```
<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>
```

Luego, anulamos el método `Application.OnStartup` para mostrar la pantalla de ***inicio*** , realizar algún trabajo y finalmente mostrar la ventana principal (***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();
            });
        });
    }
}
```

Por último, debemos cuidar el mecanismo predeterminado que muestra `MainWindow` en el inicio de la aplicación. Todo lo que necesitamos hacer es eliminar el `StartupUri="MainWindow.xaml"` de la etiqueta de la `Application` raíz en el archivo ***App.xaml*** .

Creación de la ventana de la pantalla de bienvenida con informes de progreso

WPF no admite la visualización de nada más que una imagen como una pantalla de inicio, así que tendremos que crear una `Window` que servirá como pantalla de inicio. Suponemos que ya hemos creado un proyecto que contiene la clase `MainWindow` , que será la ventana principal de la aplicación.

En primer lugar, agregamos una ventana `SplashScreenWindow` a nuestro proyecto:

```
<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>
```

Luego exponemos una propiedad en la clase `SplashScreenWindow` para que podamos actualizar fácilmente el valor de progreso actual (***SplashScreenWindow.xaml.cs***):

```
public partial class SplashScreenWindow : Window
{
    public SplashScreenWindow()
    {
        InitializeComponent();
    }

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

A continuación, anulamos el método `Application.OnStartup` para mostrar la pantalla de ***inicio*** , realizar algún trabajo y finalmente mostrar la ventana principal (***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();
            });
        });
    }
}
```

Por último, debemos cuidar el mecanismo predeterminado que muestra `MainWindow` en el inicio de

la aplicación. Todo lo que necesitamos hacer es eliminar el `StartupUri="MainWindow.xaml"` de la etiqueta de la `Application` raíz en el archivo ***App.xaml*** .

Lea Creando la pantalla de bienvenida en WPF en línea:
https://riptutorial.com/es/wpf/topic/3948/creando-la-pantalla-de-bienvenida-en-wpf

Capítulo 9: Enlace de barra deslizante: Actualizar solo en arrastre finalizado

Parámetros

Parámetro	Detalle
Valor (flotador)	La propiedad vinculada a esta propiedad de dependencia se actualizará siempre que el usuario deje de arrastrar el control deslizante.

Observaciones

- Asegúrese de hacer referencia al ensamblado *System.Windows.Interactivity* , para que el analizador XAML reconozca la declaración *xmlns: i* .
- Tenga en cuenta que la instrucción *xmlns: b* coincide con el espacio de nombres donde reside la implementación del comportamiento
- El ejemplo supone un conocimiento práctico de las expresiones de enlace y XAML.

Examples

Implementación del comportamiento

```
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);
    }
}
}
```

Uso de 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>
```

Lea Enlace de barra deslizante: Actualizar solo en arrastre finalizado en línea: https://riptutorial.com/es/wpf/topic/6339/enlace-de-barra-deslizante--actualizar-solo-en-arrastre-finalizado

Capítulo 10: Estilos en WPF

Observaciones

Notas introductorias

En WPF, un **estilo** define los valores de una o más propiedades de dependencia para un elemento visual dado. Los estilos se utilizan en toda la aplicación para hacer que la interfaz de usuario sea más consistente (por ejemplo, para que todos los botones de diálogo tengan un tamaño uniforme) y para hacer que los cambios masivos sean más fáciles (por ejemplo, para cambiar el ancho de todos los botones).

Normalmente, los estilos se definen en un `ResourceDictionary` a un alto nivel en la aplicación (por ejemplo, en *App.xaml* o en un tema), por lo que está disponible para toda la aplicación, pero también pueden definirse para un solo elemento y sus elementos *secundarios* , por ejemplo, aplicar un Estilo a todos los elementos `TextBlock` dentro de 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>
```

Notas importantes

- La ubicación donde se define el estilo afecta a donde está disponible.
- Las referencias `StaticResource` no pueden ser resueltas por `StaticResource` . En otras palabras, si está definiendo un estilo que depende de otro estilo o recurso en un diccionario de recursos, debe definirse después / debajo del recurso del cual depende.
- `StaticResource` es la forma recomendada de hacer referencia a estilos y otros recursos (por razones de rendimiento y comportamiento) a menos que requiera específicamente el uso de `DynamicResource` , por ejemplo, para temas que se pueden cambiar en tiempo de ejecución.

Recursos

MSDN tiene artículos completos sobre estilos y recursos que tienen más profundidad de la que se

puede proporcionar aquí.

- Resumen de recursos
- Estilismo y plantillas
- Control de autoría general

Examples

Definiendo un estilo nombrado

Un estilo con nombre requiere que se establezca la propiedad `x:Key` y se aplica solo a los elementos que hacen referencia explícita por nombre:

```
<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>
```

Definiendo un estilo implícito.

Un estilo implícito se aplica a todos los elementos de un tipo dado dentro del alcance. Un estilo implícito puede omitir `x:Key` ya que es implícitamente lo mismo que la propiedad `TargetType` del estilo.

```
<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>
```

Heredando de un estilo

Es común que se necesite un estilo base que defina propiedades / valores compartidos entre varios estilos que pertenezcan al mismo control, especialmente para algo como `TextBlock` . Esto se logra mediante el uso de la propiedad `BasedOn` . Los valores se heredan y luego se pueden

anular.

```
<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>
```

En el ejemplo anterior, cualquier `TextBlock` use el estilo `WarningTextBlockStyle` se presentará como 12px Arial en rojo y negrita.

Debido a que los estilos implícitos tienen una `x:Key` implícita que coincide con su `TargetType` , también puede heredarlos:

```
<!-- 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>
```

Lea Estilos en WPF en línea: https://riptutorial.com/es/wpf/topic/4090/estilos-en-wpf

Capítulo 11: Extensiones de marcado

Parámetros

Método	Descripción
ProporcionarValor	La clase MarkupExtension solo tiene un método que debe ser anulado, el analizador XAML luego usa el valor proporcionado por este método para evaluar el resultado de la extensión de marcado.

Observaciones

Se puede implementar una extensión de marcado para proporcionar valores para las propiedades en un uso de atributo, propiedades en el uso de un elemento de propiedad o ambos.

Cuando se utiliza para proporcionar un valor de atributo, la sintaxis que distingue una secuencia de extensión de marcado a un procesador XAML es la presencia de la apertura y cierre de llaves ({y}). El tipo de extensión de marcado se identifica con el token de cadena que sigue a la llave de apertura.

Cuando se usa en la sintaxis del elemento de propiedad, una extensión de marcado es visualmente lo mismo que cualquier otro elemento utilizado para proporcionar un valor de elemento de propiedad: una declaración de elemento XAML que hace referencia a la clase de extensión de marcado como un elemento, encerrada entre paréntesis angulares (<>).

Para obtener más información, visite https://msdn.microsoft.com/en-us/library/ms747254(v=vs.110).aspx

Examples

Extensión de marcado utilizada con IValueConverter

Uno de los mejores usos de las extensiones de marcado es para facilitar el uso de IValueConverter. En el siguiente ejemplo, BoolToVisibilityConverter es un convertidor de valores, pero como es independiente de la instancia, se puede utilizar sin las molestias normales de un convertidor de valores con la ayuda de la extensión de marcado. En XAML solo usa

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

y puede establecer la visibilidad del elemento en bool value.

```
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;
        }
    }
```

Extensiones de marcado definidas por XAML

Hay cuatro extensiones de marcado predefinidas en XAML:

`x:Type` suministra el objeto Type para el tipo nombrado. Esta facilidad se utiliza más frecuentemente en estilos y plantillas.

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

`x:Static` produce valores estáticos. Los valores provienen de entidades de código de tipo de valor que no son directamente el tipo del valor de una propiedad de destino, pero se pueden evaluar para ese tipo.

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

`x:Null` especifica null como un valor para una propiedad y se puede usar para atributos o valores de elementos de propiedad.

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

`x:Array` proporciona soporte para la creación de arreglos generales en la sintaxis XAML, para casos en los que el soporte de recopilación proporcionado por los elementos base de WPF y los modelos de control no se usa deliberadamente.

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

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

Lea Extensiones de marcado en línea: https://riptutorial.com/es/wpf/topic/6619/extensiones-de-marcado

Capítulo 12: Gatillos

Introducción

Discusión sobre los diversos tipos de activadores disponibles en WPF, incluidos `Trigger` , `Trigger` , `Trigger DataTrigger` , `MultiTrigger MultiDataTrigger` y `EventTrigger` .

Los desencadenadores permiten que cualquier clase que se derive de `FrameworkElement` o `FrameworkContentElement` establezca o cambie sus propiedades en función de ciertas condiciones definidas en el desencadenante. Básicamente, si un elemento se puede diseñar, también se puede activar.

Observaciones

- Todos los activadores, excepto `EventTrigger` deben definirse dentro de un elemento `<Style>` . Un `EventTrigger` se puede definir en un elemento `<Style>` o en la propiedad `Triggers` un control.
- `<Trigger>` elementos `<Trigger>` pueden contener cualquier número de elementos `<Setter>` . Estos elementos son responsables de configurar las propiedades en el elemento que contiene cuando se cumple la condición del elemento `<Trigger>` .
- Si se define una propiedad en la marca del elemento raíz, el cambio de propiedad definido en el elemento `<Setter>` no tendrá efecto, incluso si se cumple la condición de activación. Considere el marcado `<TextBlock Text="Sample">` . La propiedad de `Text` del código de procedimiento nunca cambiará en función de un desencadenante porque las definiciones de propiedades de raíz tienen prioridad sobre las propiedades definidas en estilos.
- Al igual que los enlaces, una vez que se ha utilizado un activador, no se puede modificar.

Examples

Desencadenar

El más simple de los cinco tipos de activadores, el `Trigger` es responsable de establecer propiedades basadas en otras propiedades **dentro del mismo control** .

```
<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>
```

En este ejemplo, el color de primer plano del `TextBlock` volverá verde cuando su propiedad `Text` sea igual a la cadena `"Pass"` .

MultiTrigger

Un `MultiTrigger` es similar a un `Trigger` estándar en que solo se aplica a las propiedades **dentro del mismo control** . La diferencia es que un `MultiTrigger` tiene múltiples condiciones que deben cumplirse antes de que funcione el gatillo. Las condiciones se definen utilizando la etiqueta `<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>
```

Observe que el `MultiTrigger` no se activará hasta que se cumplan ambas condiciones.

Data Trigger

Un `DataTrigger` se puede adjuntar a cualquier propiedad, ya sea bajo su propio control, otro control, o incluso una propiedad en una clase que no sea UI. Considere la siguiente clase simple.

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

Que adjuntaremos como el `DataContext` en el siguiente `TextBlock` .

```
<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>
```

En el código anterior, la propiedad `TextBlock.Foreground` será verde. Si cambiamos la propiedad `StinkLevel` en nuestro XAML a algo distinto a 100, la propiedad `Text.Foreground` volverá a su valor predeterminado.

Lea Gatillos en línea: https://riptutorial.com/es/wpf/topic/9624/gatillos

Capítulo 13: Introducción al enlace de datos WPF

Sintaxis

- {Binding PropertyName} *es equivalente a* {Binding Path = PropertyName}
- {Ruta de enlace = SomeProperty.SomeOtherProperty.YetAnotherProperty}
- {Ruta de enlace = SomeListProperty [1]}

Parámetros

Parámetro	Detalles
Camino	Especifica la ruta a enlazar. Si no se especifica, se enlaza con el propio DataContext.
UpdateSourceTrigger	Especifica cuándo la fuente de enlace tiene su valor actualizado. El valor predeterminado es `LostFocus` . El valor más utilizado es `PropertyChanged` .
Modo	Típicamente `OneWay` o `TwoWay` . Si el enlace no lo especifica, el valor predeterminado es `OneWay` menos que el destino del enlace solicite que sea `TwoWay` . Se produce un error cuando `TwoWay` se utiliza para enlazar a una propiedad de sólo lectura, por ejemplo `OneWay` se debe configurar de forma explícita cuando se une una propiedad de cadena de sólo lectura a `TextBox.Text` .
Fuente	Permite utilizar un `StaticResource` como fuente de enlace en lugar del DataContext actual.
Fuente relativa	Permite utilizar otro elemento XAML como fuente de enlace en lugar del DataContext actual.
ElementName	Permite usar un elemento XAML nombrado como fuente de enlace en lugar del DataContext actual.
FallbackValue	Si el enlace falla, este valor se proporciona al objetivo de enlace.
TargetNullValue	Si el valor de origen de enlace es `null` , este valor se proporciona al destino de enlace.
Convertidor	Especifica el convertidor `StaticResource` que se utiliza para convertir el valor del enlace, por ejemplo, convertir un booleano en un elemento de enumeración de `Visibility` .

Parámetro	Detalles
Convertidor Parámetro	Especifica un parámetro opcional que se proporcionará al convertidor. Este valor debe ser estático y no se puede enlazar.
StringFormat	Especifica una cadena de formato que se utilizará al mostrar el valor enlazado.
Retrasar	(WPF 4.5+) Especifica un retraso en `milliseconds` para que el enlace actualice el `BindingSource` en el `ViewModel` . Esto debe usarse con `Mode=TwoWay UpdateSourceTrigger=PropertyChanged` y `UpdateSourceTrigger=PropertyChanged` para que tenga efecto.

Observaciones

UpdateSourceTrigger

De forma predeterminada, WPF actualiza el origen de enlace cuando el control pierde el foco. Sin embargo, si solo hay un control que puede enfocarse, algo que es común en los ejemplos, deberá especificar `UpdateSourceTrigger=PropertyChanged` para que las actualizaciones funcionen.

Querrá utilizar `PropertyChanged` como desencadenante en muchos enlaces de dos vías, a menos que la actualización de la fuente de enlace en cada pulsación de tecla sea costosa o la validación de datos en vivo no sea deseable.

El uso de `LostFocus` tiene un efecto secundario desafortunado: presionar intro para enviar un formulario usando un botón marcado como `IsDefault` no actualiza la propiedad que respalda su enlace, deshaciendo efectivamente sus cambios. Afortunadamente, existen algunas soluciones .

Tenga en cuenta también que, a diferencia de UWP, WPF (4.5+) también tiene la propiedad `Delay` en los enlaces, lo que podría ser suficiente para algunos Bindings con configuraciones de inteligencia secundaria solo o local, como algunas validaciones de `TextBox` .

Examples

Convertir un valor booleano a visibilidad

Este ejemplo oculta la casilla roja (borde) si la casilla de verificación no está marcada haciendo uso de un `IValueConverter` .

Nota: El `BooleanToVisibilityConverter` usa en el siguiente ejemplo es un convertidor de valores integrado, ubicado en el espacio de nombres 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>
```

Definiendo el DataContext

Para trabajar con enlaces en WPF, debe definir un **DataContext** . El DataContext le dice a los enlaces de dónde obtener sus datos de forma predeterminada.

```
<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>
```

También puede establecer el DataContext a través del código subyacente, pero vale la pena señalar que XAML IntelliSense es algo delicado: se debe establecer un DataContext fuertemente tipado en XAML para que IntelliSense sugiera las propiedades disponibles para el enlace.

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

Si bien existen marcos para ayudarlo a definir su DataContext de una manera más flexible (por ejemplo, MVVM Light tiene un localizador de modelos de vista que usa inversión de control), usamos el método rápido y sucio para los propósitos de este tutorial.

Puede definir un DataContext para casi cualquier elemento visual en WPF. El DataContext

generalmente se hereda de los antepasados en el árbol visual a menos que se haya anulado explícitamente, por ejemplo, dentro de un ContentPresenter.

Implementando INotifyPropertyChanged

`INotifyPropertyChanged` es una interfaz utilizada por las fuentes de enlace (es decir, el DataContext) para que la interfaz de usuario u otros componentes sepan que una propiedad ha sido cambiada. WPF actualiza automáticamente la interfaz de usuario cuando ve el evento `PropertyChanged` generado. Es deseable tener esta interfaz implementada en una clase base de la que puedan heredar todos sus modelos de vista.

En C # 6, esto es todo lo que necesitas:

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

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

Esto le permite invocar `NotifyPropertyChanged` de dos maneras diferentes:

1. `NotifyPropertyChanged()` , que `NotifyPropertyChanged()` el evento para el establecedor que lo invoca, gracias al atributo CallerMemberName .
2. `NotifyPropertyChanged(nameof(SomeOtherProperty))` , que generará el evento para SomeOtherProperty.

Para .NET 4.5 y superior utilizando C # 5.0, esto puede usarse en su lugar:

```
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));
        }
    }
}
```

En versiones de .NET anteriores a 4.5, debe conformarse con los nombres de propiedades como constantes de cadena o una solución utilizando expresiones .

Nota: es posible enlazar a una propiedad de un "objeto C # antiguo" (POCO) que no implementa `INotifyPropertyChanged` y observar que los enlaces funcionan mejor de lo esperado. Esta es una

característica oculta en .NET y probablemente debería evitarse. Especialmente porque causará pérdidas de memoria cuando el `Mode` de enlace no sea `OneTime` (ver aquí).

¿Por qué se actualiza el enlace sin implementar INotifyPropertyChanged?

Vincular a propiedad de otro elemento nombrado

Puede enlazar a una propiedad en un elemento con nombre, pero el elemento con nombre debe estar dentro del alcance.

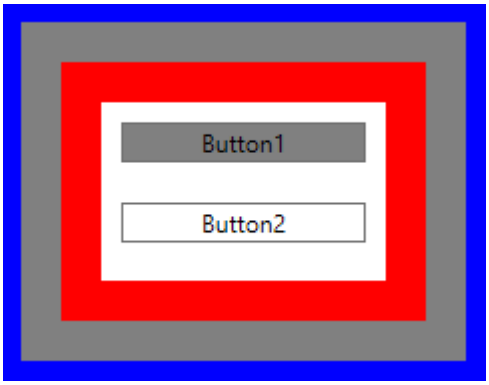
```
<StackPanel>
    <CheckBox x:Name="MyCheckBox" IsChecked="True" />
    <TextBlock Text="{Binding IsChecked, ElementName=MyCheckBox}" />
</StackPanel>
```

Vincular a la propiedad de un antepasado

Puede enlazar a una propiedad de un antepasado en el árbol visual utilizando un enlace `RelativeSource` . El control más cercano más alto en el árbol visual que tiene el mismo tipo o se deriva del tipo que especifique se usará como fuente del enlace:

```
<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>
```

En este ejemplo, *Button1* tiene un fondo gris porque el antepasado `Grid` más cercano tiene un fondo gris. *Button2* tiene un fondo blanco porque el antepasado más cercano derivado de `FrameworkElement` es el `StackPanel` blanco.



Vinculando múltiples valores con un MultiBinding

El MultiBinding permite vincular múltiples valores a la misma propiedad. En el siguiente ejemplo, varios valores están vinculados a la propiedad Texto de un cuadro de texto y se formatean utilizando la propiedad StringFormat.

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

Además de `StringFormat` , también se puede usar un `IMultiValueConverter` para convertir los valores de los enlaces en un valor para el objetivo de MultiBinding.

Sin embargo, MultiBindings no puede ser anidado.

Lea Introducción al enlace de datos WPF en línea:
https://riptutorial.com/es/wpf/topic/2236/introduccion-al-enlace-de-datos-wpf

Capítulo 14: Localización WPF

Observaciones

El contenido de los controles se puede localizar utilizando archivos de recursos, al igual que esto es posible en las clases. Para XAML hay una sintaxis específica, que es diferente entre una aplicación C # y una VB.

Los pasos son:

- Para cualquier proyecto de WPF: haga público el archivo de recursos, el valor predeterminado es interno.
- Para proyectos C # WPF use el XAML provisto en el ejemplo
- Para proyectos de VB WPF, use el XAML que se proporciona en el ejemplo y cambie la propiedad de la Herramienta personalizada a `PublicVbMyResourcesResXFileCodeGenerator` .
- Para seleccionar el archivo Resources.resx en un proyecto VB WPF:
    - Seleccione el proyecto en explorador de soluciones.
    - Seleccione "Mostrar todos los archivos"
    - Expandir mi proyecto

Examples

XAML para VB

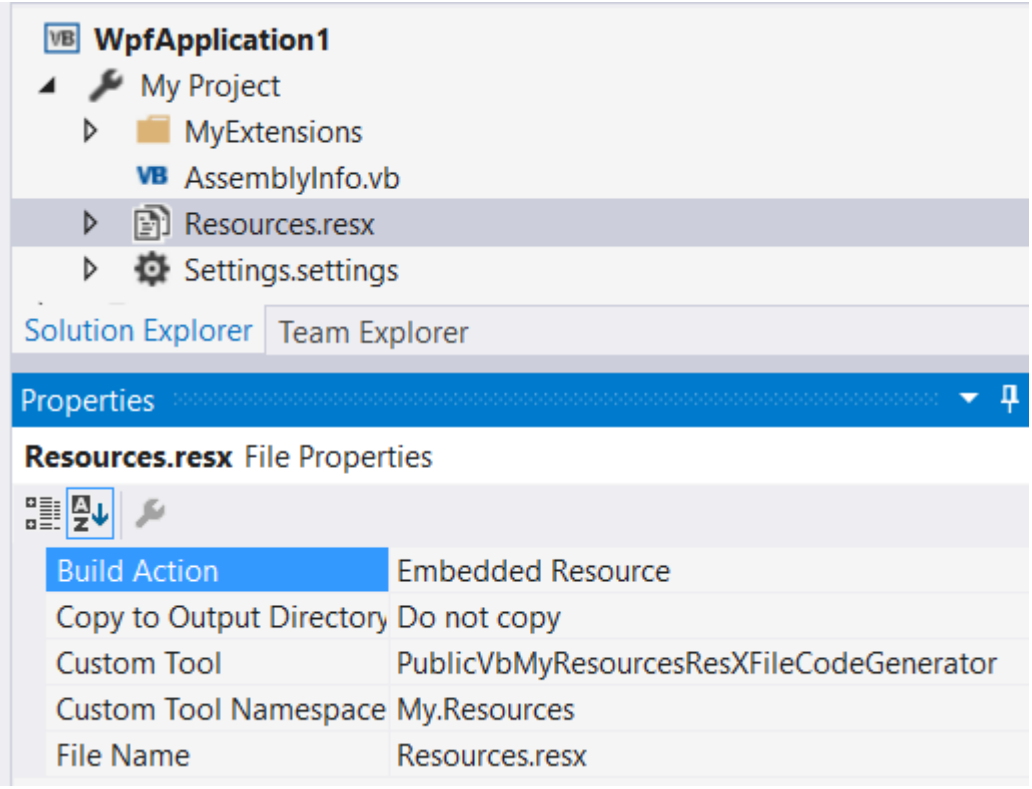
```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"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

Propiedades para el archivo de recursos en VB

De forma predeterminada, la propiedad Herramienta personalizada para un archivo de recursos de VB es `VbMyResourcesResXFileCodeGenerator` . Sin embargo, con este generador de código, la vista (XAML) no podrá acceder a los recursos. Para resolver este problema, agregue `Public` antes del valor de la propiedad Herramienta personalizada.

Para seleccionar el archivo Resources.resx en un proyecto VB WPF:

- Seleccione el proyecto en explorador de soluciones.
- Seleccione "Mostrar todos los archivos"
- Expandir "Mi proyecto"

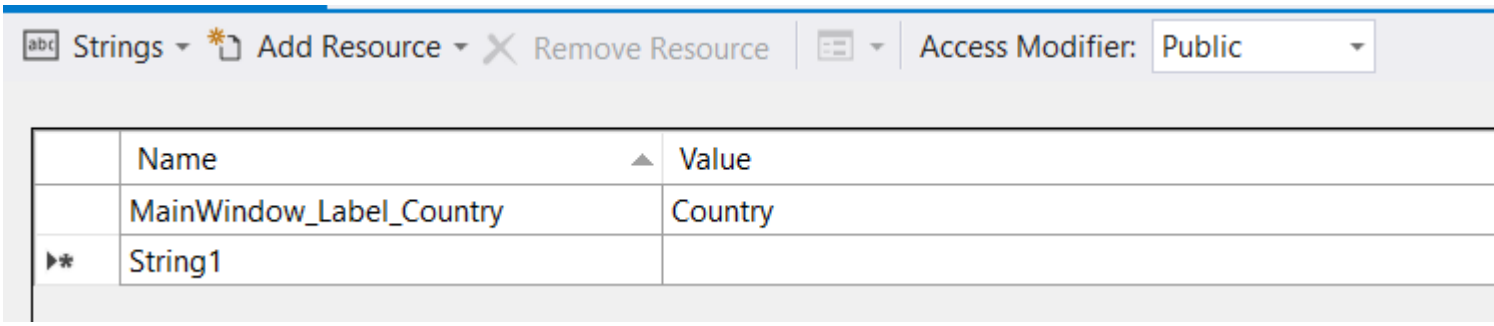


XAML para C

```
<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>
```

Hacer públicos los recursos.

Abra el archivo de recursos haciendo doble clic en él. Cambie el modificador de acceso a "Público".



Lea Localización WPF en línea: https://riptutorial.com/es/wpf/topic/3905/localizacion-wpf

Capítulo 15: MVVM en WPF

Observaciones

Modelos y modelos de vista

La definición de un modelo a menudo se debate ardientemente, y la línea entre un modelo y un modelo de vista se puede difuminar. Algunos prefieren no "contaminar" sus modelos con el `INotifyPropertyChanged` interfaz, y en lugar de duplicar las del modelo en la vista-modelo, que *hace* implementar esta interfaz. Como muchas cosas en el desarrollo de software, no hay una respuesta correcta o incorrecta. Sea pragmático y haga lo que se sienta bien.

Ver separación

La intención de MVVM es separar esas tres áreas distintas: Modelo, modelo de vista y Vista. Si bien es aceptable para la vista acceder al modelo de vista (VM) y (indirectamente) al modelo, la regla más importante con MVVM es que la VM no debe tener acceso a la vista ni a sus controles. La máquina virtual debe exponer todo lo que la vista necesita, a través de propiedades públicas. La máquina virtual no debe exponer ni manipular directamente los controles de la IU, como `TextBox` , `Button` , etc.

En algunos casos, puede ser difícil trabajar con esta separación estricta, especialmente si necesita poner en funcionamiento alguna funcionalidad de interfaz de usuario compleja. Aquí, es perfectamente aceptable recurrir al uso de eventos y controladores de eventos en el archivo de "código subyacente" de la vista. Si se trata de una funcionalidad de UI pura, entonces utilice los eventos en la vista. También es aceptable que estos manejadores de eventos llamen a métodos públicos en la instancia de VM, simplemente no le pasen referencias a los controles de UI ni nada de eso.

RelayCommand

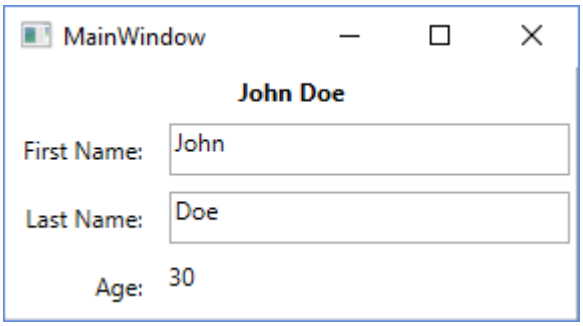
Desafortunadamente, la clase `RelayCommand` utilizada en este ejemplo no forma parte del marco de trabajo de WPF (¡debería haberlo sido!), Pero lo encontrará en casi todas las cajas de herramientas para desarrolladores de WPF. Una búsqueda rápida en línea revelará muchos fragmentos de código que puede levantar, para crear el suyo propio.

Una alternativa útil a `RelayCommand` es `ActionCommand` que se proporciona como parte de `Microsoft.Expression.Interactivity.Core` que proporciona una funcionalidad comparable.

Examples

Ejemplo básico de MVVM usando WPF y C

Este es un ejemplo básico para usar el modelo MVVM en una aplicación de escritorio de Windows, usando WPF y C #. El código de ejemplo implementa un simple diálogo de "información del usuario".



La vista

El 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>
```

y el código detrás

```
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;
    }
}
```

El modelo de vista

```
// 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));
        }
    }
}
```

El modelo

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

    public string LastName { get; set; }

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

El modelo de vista

El modelo de vista es la "VM" en MV **VM** . Esta es una clase que actúa como intermediario, expone los modelos a la interfaz de usuario (vista) y maneja las solicitudes desde la vista, como los comandos generados por los clics de los botones. Aquí hay un modelo de vista básico:

```
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
    }
}
```

El constructor crea un objeto de modelo de `Customer` y lo asigna a la propiedad `CustomerToEdit` , para que sea visible para la vista.

El constructor también crea un objeto `RelayCommand` y lo asigna a la propiedad `ApplyChangesCommand` , haciéndolo nuevamente visible para la vista. Los comandos de WPF se utilizan para manejar solicitudes desde la vista, como los clics en los botones o en los elementos del menú.

`RelayCommand` toma dos parámetros: el primero es el delegado al que se llama cuando se ejecuta el comando (por ejemplo, en respuesta a un clic del botón). El segundo parámetro es un delegado que devuelve un valor booleano que indica si el comando puede ejecutarse; en este ejemplo, está conectado a la propiedad `IsValid` del objeto del `IsValid` . Cuando esto devuelve falso, desactiva el botón o elemento de menú que está vinculado a este comando (otros controles pueden comportarse de manera diferente). Esta es una característica simple pero efectiva, evitando la necesidad de escribir código para habilitar o deshabilitar controles basados en diferentes condiciones.

Si tiene este ejemplo en funcionamiento, intente vaciar uno de los `TextBox` de `TextBox` (para colocar el modelo del `Customer` en un estado no válido). Cuando se aleje del `TextBox` , debería encontrar que el botón "Aplicar" se deshabilita.

Comentario sobre la creación del cliente

El modelo de vista no implementa `INotifyPropertyChanged` (INPC). Esto significa que si se asignara un objeto `Customer` diferente a la propiedad `CustomerToEdit` , entonces los controles de la vista no cambiarían para reflejar el nuevo objeto; el `TextBox` aún contendría el nombre y el apellido del cliente anterior.

El código de ejemplo funciona porque el `Customer` se crea en el constructor del modelo de vista, antes de que se asigne al `DataContext` la vista (en cuyo punto se conectan los enlaces). En una aplicación del mundo real, podría estar recuperando clientes de una base de datos en métodos distintos al constructor. Para admitir esto, la VM debe implementar INPC, y la propiedad `CustomerToEdit` debe cambiarse para usar el patrón "establecido" de obtención y establecimiento que se ve en el código de modelo de ejemplo, lo que provoca el evento `PropertyChanged` en el configurador.

El `ApplyChangesCommand` del modelo de `ApplyChangesCommand` no necesita implementar INPC ya que es muy poco probable que el comando cambie. Lo que *se* necesita para implementar este patrón si estuviera creando un lugar del comando que no sea el constructor, por ejemplo, algún tipo de `Initialize()` método.

La regla general es: implementar INPC si la propiedad está vinculada a cualquier control de vista *y* el valor de la propiedad puede cambiar en cualquier lugar que no sea el constructor. No

necesita implementar INPC si el valor de la propiedad solo se asigna en el constructor (y se ahorrará algo de escritura en el proceso).

El modelo

El modelo es la primera "M" en **M** VVM. El modelo suele ser una clase que contiene los datos que desea exponer a través de algún tipo de interfaz de usuario.

Aquí hay una clase de modelo muy simple que expone un par de propiedades:

```
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));
    }
}
```

Esta clase implementa la interfaz `INotifyPropertyChanged` que expone un evento `PropertyChanged` . Este evento debe producirse siempre que cambie uno de los valores de propiedad; puede verlo en acción en el código anterior. El evento `PropertyChanged` es una pieza clave en los mecanismos de enlace de datos de WPF, ya que sin él, la interfaz de usuario no podría reflejar los cambios realizados en el valor de una propiedad.

El modelo también contiene una rutina de validación muy simple a la que se llama desde los establecedores de propiedades. Establece una propiedad pública que indica si el modelo está o no en un estado válido. He incluido esta funcionalidad para demostrar una característica "especial" de los *comandos* de WPF, que veremos en breve. *El marco WPF proporciona una serie de enfoques más sofisticados para la validación, pero están fuera del alcance de este artículo* .

La vista

La vista es la "V" en M **V** VM. Esta es su interfaz de usuario. Puede usar el diseñador de arrastrar y soltar de Visual Studio, pero la mayoría de los desarrolladores finalmente terminan codificando el XAML en bruto, una experiencia similar a la de escribir HTML.

Aquí está el XAML de una vista simple para permitir la edición de un modelo de `Customer` . En lugar de crear una nueva vista, esto solo se puede pegar en el archivo `MainWindow.xaml` un proyecto WPF, entre las etiquetas `<Window ...>` y `</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>
```

Este código crea un formulario de entrada de datos simple que consta de dos `TextBox` es: uno para el nombre del cliente y otro para el apellido. Hay una `Label` encima de cada `TextBox` y un `Button` "Aplicar" en la parte inferior del formulario.

Localiza el primer `TextBox` y mira su propiedad de `Text` :

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

En lugar de establecer el `TextBox` del `TextBox` en un valor fijo, esta sintaxis especial de corchete es el enlace del texto a la "ruta" `CustomerToEdit.Forename` . ¿A qué se refiere este camino? Es el "contexto de datos" de la vista, en este caso, nuestro modelo de vista. La ruta de enlace, como podrá descubrir, es la propiedad `CustomerToEdit` del modelo de vista, que es de tipo `Customer` que a su vez expone una propiedad llamada `Forename` - por lo tanto, la notación de ruta "punteada".

De manera similar, si observa la XAML del `Button` , tiene un `Command` que está vinculado a la propiedad `ApplyChangesCommand` del modelo de vista. Eso es todo lo que se necesita para conectar un botón al comando de la máquina virtual.

El DataContext

Entonces, ¿cómo configura el modelo de vista para que sea el contexto de datos de la vista? Una forma es configurarlo en el "código subyacente" de la vista. Presione F7 para ver este archivo de código y agregue una línea al constructor existente para crear una instancia del modelo de vista y asignarla a la propiedad `DataContext` la ventana. Debería acabar luciendo así:

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

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

En los sistemas del mundo real, a menudo se utilizan otros enfoques para crear el modelo de

vista, como la inyección de dependencias o los marcos MVVM.

Comandando en MVVM

Los comandos se utilizan para manejar `Events` en WPF respetando el patrón MVVM.

Un `EventHandler` normal se vería así (ubicado en `Code-Behind`):

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

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

No para hacer lo mismo en MVVM usamos `Commands` :

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

> Recomiendo usar algún tipo de prefijo (`Cmd`) para las propiedades de tus comandos, porque principalmente los necesitarás en xaml, de esa forma son más fáciles de reconocer.

Ya que es MVVM, usted quiere manejar ese comando (para el `Button` "eq" `Button_Click`) en su `ViewModel` .

Para eso básicamente necesitamos dos cosas:

1. System.Windows.Input.ICommand
2. RelayCommand (por ejemplo, tomado de aquí .

Un **ejemplo** simple podría verse así:

```
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.
}
```

Entonces, ¿qué está haciendo esto en detalle:

El `ICommand` es a lo que se vincula el `Control` en xaml. `RelayCommand` dirigirá su comando a una `Action` (es decir, llama a un `Method`). La comprobación de nulos solo garantiza que cada `Command` solo se inicializará una vez (debido a problemas de rendimiento). Si ha leído el enlace de `RelayCommand` anterior, puede haber notado que `RelayCommand` tiene dos sobrecargas para su constructor. `(Action<object> execute)` y `(Action<object> execute, Predicate<object> canExecute)` .

Eso significa que puede (adicionalmente) agregar un segundo `Method` devolviendo un `bool` para decirle a `Control` el "Evento" puede activarse o no.

Una buena cosa para eso es que `Button` s, por ejemplo, se `Enabled="false"` si el `Method` devolverá `false`

CommandParameters

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

En este ejemplo, quiero pasar el `DataGrid.SelectedItem` al Click_Command en mi ViewModel.

Su Método debería tener este aspecto mientras que la implementación de ICommand se mantiene como se indica arriba.

```
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
    }
}
```

Lea MVVM en WPF en línea: https://riptutorial.com/es/wpf/topic/2134/mvvm-en-wpf

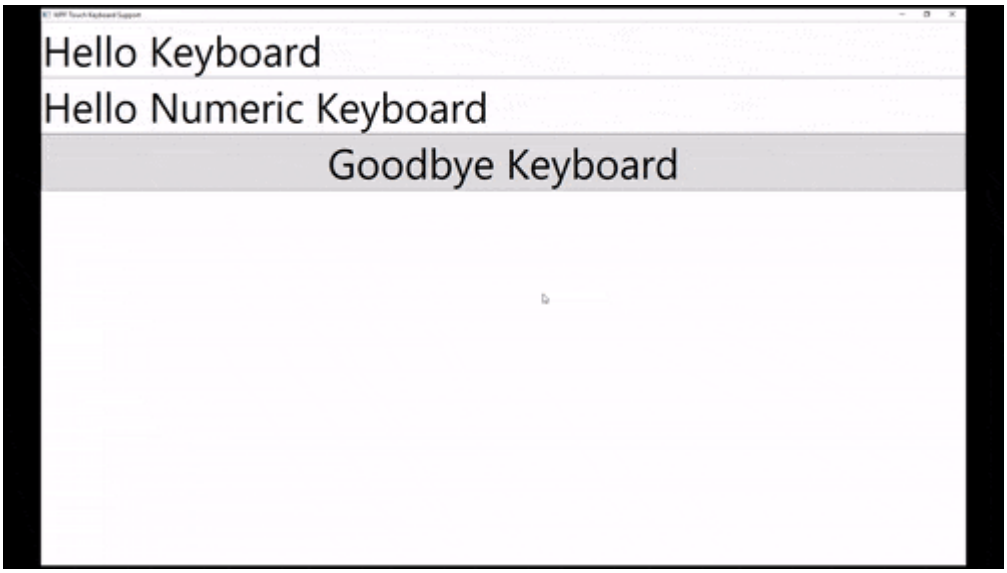
Capítulo 16: Optimización para la interacción táctil

Examples

Mostrando teclado táctil en Windows 8 y Windows 10

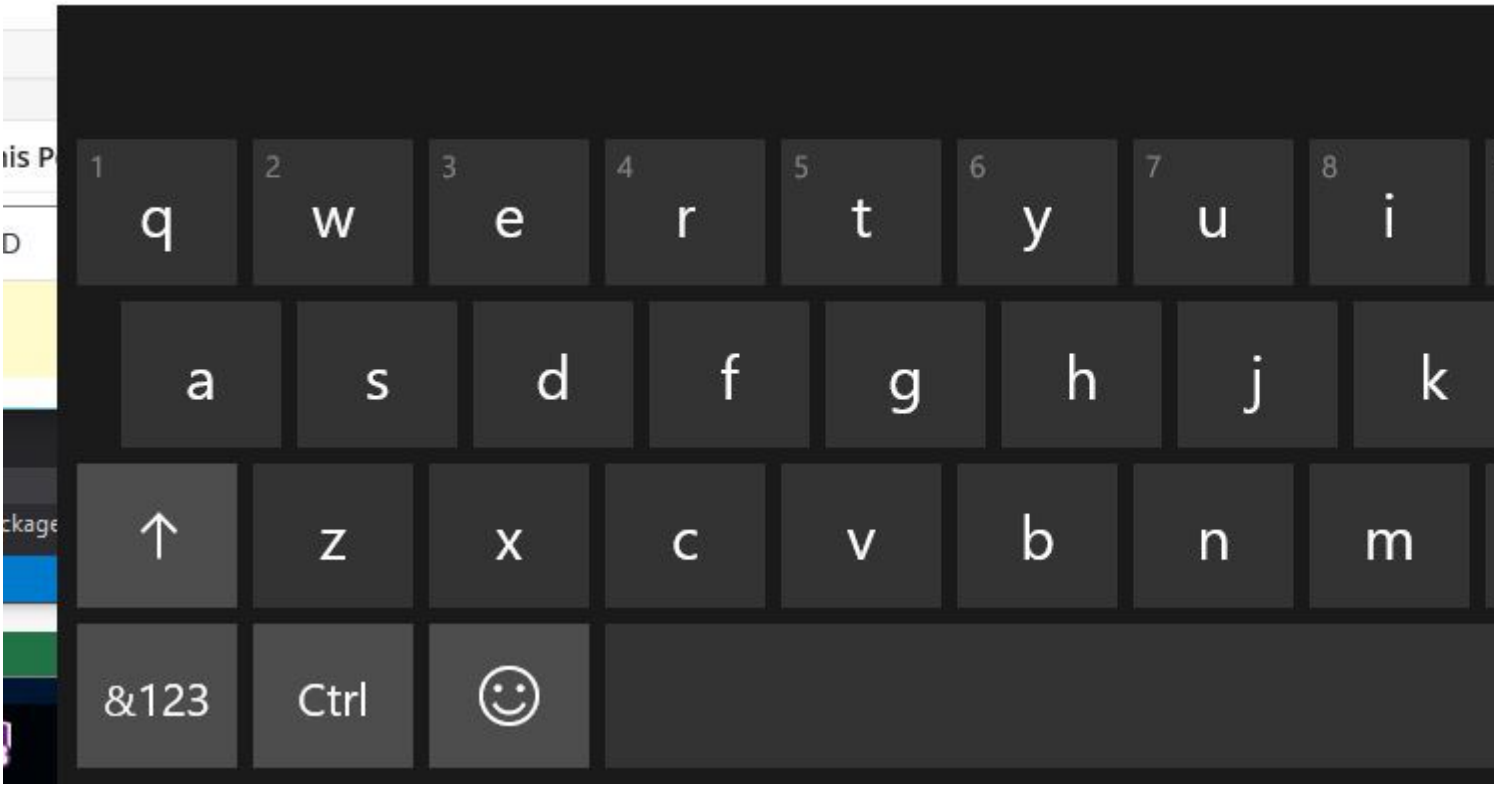
Aplicaciones WPF dirigidas a .NET Framework 4.6.2 y posteriores

Con las aplicaciones de WPF dirigidas a .NET Framework 4.6.2 (y posteriores), el teclado virtual se invoca y cierra automáticamente sin que sea necesario realizar ningún paso adicional.



Aplicaciones WPF dirigidas a .NET Framework 4.6.1 y anteriores

WPF no está habilitado principalmente para tocar, lo que significa que cuando el usuario interactúa con una aplicación de WPF en el escritorio, la aplicación **no mostrará automáticamente el teclado** cuando los controles de `TextBox` reciben el enfoque. Este es un comportamiento inconveniente para los usuarios de tabletas, obligándolos a abrir manualmente el teclado táctil a través de la barra de tareas del sistema.



Solución

El teclado táctil es en realidad una aplicación `exe` clásica que se puede encontrar en cada PC con Windows 8 y Windows 10 en la siguiente ruta: `C:\Program Files\Common Files\Microsoft Shared\Ink\TabTip.exe` .

En base a este conocimiento, puede crear un control personalizado derivado de `TextBox` , que escucha el evento `GotTouchCapture` (este evento se llama cuando el control se enfoca al tocar) e **inicia el proceso del teclado táctil** .

```
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 );
    }
}
```

Puede mejorar esto aún más si almacena en caché el proceso creado y luego lo mata después de que el control pierda el foco:

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

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

Ahora puedes conectar el evento `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;
   }
}
```

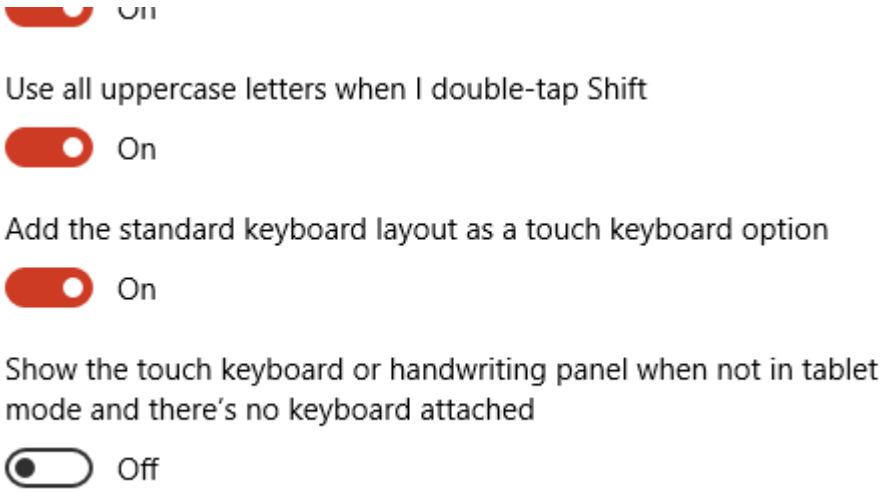
Nota sobre el modo tableta en Windows 10

Windows 10 introdujo un **modo de tableta** , que simplifica la interacción con el sistema cuando se usa la PC de manera táctil. Este modo, aparte de otras mejoras, garantiza que el **teclado táctil se muestre automáticamente** incluso para las aplicaciones de escritorio clásicas, incluidas las aplicaciones WPF.

Enfoque de configuración de Windows 10

Además del modo tableta, Windows 10 puede mostrar automáticamente el teclado táctil para aplicaciones clásicas incluso fuera del modo tableta. Este comportamiento, que está deshabilitado de forma predeterminada, se puede habilitar en la aplicación Configuración.

En la aplicación **Configuración** , vaya a la categoría **Dispositivos** y seleccione **Escritura** . Si se desplaza completamente hacia abajo, puede encontrar la opción "Mostrar el teclado táctil o el panel de escritura a mano cuando no está en el modo de tableta y no hay ningún teclado conectado", que pueda habilitar.



Cabe mencionar que esta configuración solo es visible en dispositivos con capacidades táctiles.

Lea Optimización para la interacción táctil en línea:
https://riptutorial.com/es/wpf/topic/6799/optimizacion-para-la-interaccion-tactil

Capítulo 17: Principio de diseño "La mitad del espacio en blanco"

Introducción

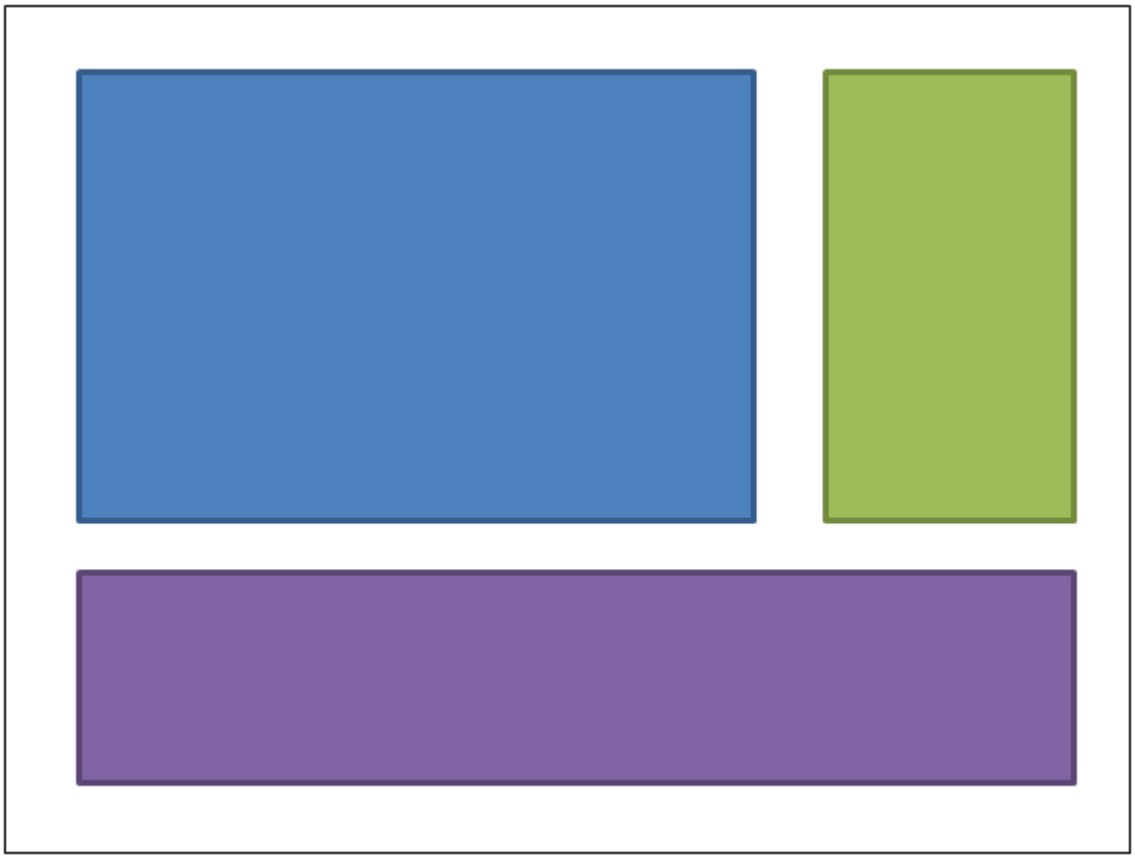
Al diseñar los controles, es fácil codificar valores específicos en márgenes y rellenos para que las cosas se ajusten al diseño deseado. Sin embargo, al codificar estos valores, el mantenimiento se vuelve mucho más caro. Si el diseño cambia, en lo que podría considerarse una forma trivial, entonces hay que trabajar mucho para corregir estos valores.

Este principio de diseño reduce el costo de mantenimiento del diseño al pensar en el diseño de una manera diferente.

Examples

Demostración del problema y la solución.

Por ejemplo, imagina una pantalla con 3 secciones, dispuestas así:



La caja azul podría tener un margen de 4,4,0,0. El cuadro verde podría tener un margen de 4,4,4,0. El margen de la caja púrpura sería 4,4,4,4. Aquí está el XAML: (Estoy usando una cuadrícula para lograr el diseño, pero este principio de diseño se aplica independientemente de

cómo elijas para lograr el diseño):

```
<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>
```

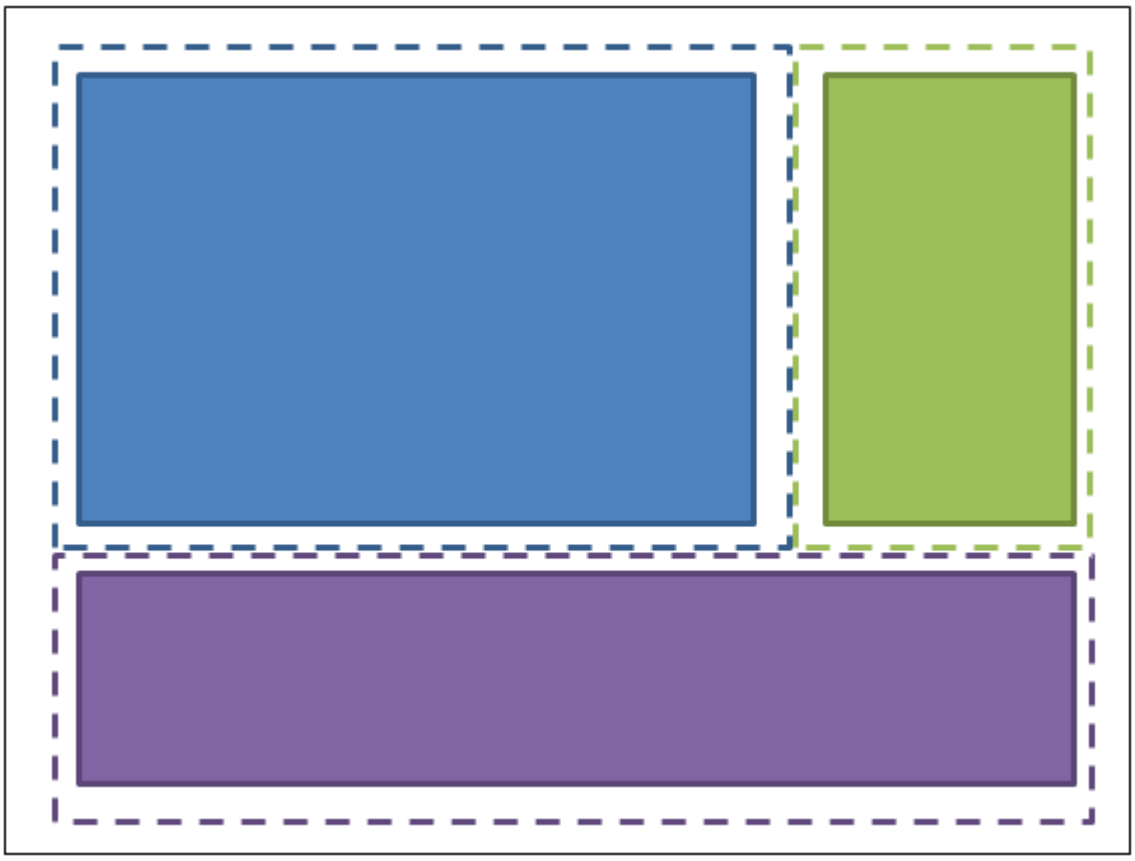
Ahora imagine que queremos cambiar el diseño, para colocar el cuadro verde a la izquierda del cuadro azul. Debería ser simple, ¿no? Excepto que cuando movemos esa caja, ahora necesitamos jugar con los márgenes. O bien podemos cambiar los márgenes de la caja azul a 0,4,4,0; o podríamos cambiar azul a 4,4,4,0 y verde a 4,4,0,0. Aquí está el 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>
```

Ahora vamos a poner la caja morada en la parte superior. Entonces los márgenes del azul se convierten en 4,0,4,4; El verde se convierte en 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>
```

¿No sería bueno si pudiéramos mover las cosas de manera que no tuviéramos que ajustar estos valores en absoluto? Esto se puede lograr simplemente pensando en el espacio en blanco de una manera diferente. En lugar de asignar todo el espacio en blanco a un control u otro, imagine la mitad del espacio en blanco que se asigna a cada cuadro: (mi dibujo no está del todo a escala: las líneas de puntos deben estar a medio camino entre el borde del cuadro y su vecino) .



Así que la caja azul tiene márgenes de 2,2,2,2; La caja verde tiene márgenes de 2,2,2,2; La caja morada tiene márgenes de 2,2,2,2. Y al contenedor en el que están alojados se le da un relleno (no margen) de 2,2,2,2. Aquí está el 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>
```

Ahora intentemos mover las cajas de la misma manera que antes ... Pongamos la caja verde a la izquierda de la caja azul. OK hecho. Y no hubo necesidad de cambiar ningún relleno o márgenes. Aquí está el 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>
```

Ahora vamos a poner la caja morada en la parte superior. OK hecho. Y no hubo necesidad de cambiar ningún relleno o márgenes. Aquí está el 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>
```

Cómo usar esto en código real.

Para generalizar lo que hemos demostrado anteriormente: las cosas individuales contienen un *margen* fijo de "mitad del espacio en blanco", y el contenedor en el que se encuentran debe tener un *relleno* de "mitad del espacio en blanco". Puede aplicar estos estilos en el diccionario de recursos de su aplicación, y ni siquiera necesitará mencionarlos en los elementos individuales. Así es como puedes definir "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}"/>
```

Luego puedo definir un estilo base para basar mis otros estilos de controles en: (esto también podría contener su FontFamily, FontSize, etc., por defecto)

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

Luego puedo definir mi estilo predeterminado para que TextBox use este margen:

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

Puedo hacer este tipo de cosas para DatePickers, etiquetas, etc., etc. (cualquier cosa que pueda estar dentro de un contenedor). Tenga cuidado con el estilo de TextBlock como este ... ese control es usado internamente por muchos controles. Te sugiero que crees tu propio control que simplemente se deriva de TextBlock. Puede aplicar estilo a *su* TextBlock para usar el margen predeterminado; y debe usar *su* TextBlock siempre que use explícitamente un TextBlock en su XAML.

Puede utilizar un enfoque similar para aplicar el relleno a contenedores comunes (por ejemplo, ScrollViewer, Border, etc.).

Una vez que hayas hecho esto, la *mayoría* de tus controles no necesitarán márgenes ni relleno, y solo necesitarás especificar los valores en los lugares donde quieras desviarte intencionalmente de este principio de diseño.

Lea Principio de diseño "La mitad del espacio en blanco" en línea: https://riptutorial.com/es/wpf/topic/9407/principio-de-diseno--la-mitad-del-espacio-en-blanco-

Capítulo 18: Propiedades de dependencia

Introducción

Las propiedades de dependencia son un tipo de propiedad que se extiende a una propiedad CLR. Mientras que una propiedad CLR se lee directamente de un miembro de su clase, una Propiedad de dependencia se resolverá dinámicamente al llamar al método GetValue () que su objeto obtiene a través de la herencia de la clase DependencyObject básica.

Esta sección desglosará las Propiedades de dependencia y explicará su uso tanto a nivel conceptual como a través de ejemplos de código.

Sintaxis

- DependencyProperty.Register (nombre de cadena, Type propertyType, Type ownerType)
- DependencyProperty.Register (nombre de cadena, Type propertyType, Type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.Register (nombre de cadena, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterAttached (nombre de cadena, tipo propertyType, tipo ownerType)
- DependencyProperty.RegisterAttached (nombre de cadena, Type propertyType, Type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterAttached (nombre de cadena, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterReadOnly (nombre de cadena, Type propertyType, Type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterReadOnly (nombre de cadena, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterAttachedReadOnly (nombre de cadena, Type propertyType, Type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterAttachedReadOnly (nombre de cadena, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)

Parámetros

Parámetro	Detalles
nombre	La representación en `String` del nombre de la propiedad.
tipo de propiedad	El `Type` de la propiedad, por ejemplo, `typeof(int)`
ownerType	El `Type` de la clase en la que se define la propiedad, por ejemplo, `typeof(MyControl)` o `typeof(MyAttachedProperties)` .

Parámetro	Detalles
typeMetadata	Instancia de `System.Windows.PropertyMetadata` (o una de sus subclases) que define valores predeterminados, devoluciones de llamada modificadas de propiedad, `FrameworkPropertyMetadata` permite definir opciones de enlace como `System.Windows.Data.BindingMode.TwoWay` .
validateValueCallback	Devolución de llamada personalizada que devuelve verdadero si el nuevo valor de la propiedad es válido, de lo contrario es falso.

Examples

Propiedades de dependencia estándar

Cuándo usar

Prácticamente todos los controles de WPF hacen un uso intensivo de las propiedades de dependencia. Una propiedad de dependencia permite el uso de muchas características de WPF que no son posibles solo con las propiedades CLR estándar, incluidas, entre otras, la compatibilidad con estilos, animaciones, enlace de datos, herencia de valores y notificaciones de cambios.

La propiedad `TextBox.Text` es un ejemplo simple de dónde se necesita una propiedad de dependencia estándar. Aquí, el enlace de datos no sería posible si el `Text` fuera una propiedad CLR estándar.

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

Como definir

Las propiedades de dependencia solo se pueden definir en clases derivadas de `DependencyObject` , como `FrameworkElement` , `Control` , etc.

Una de las formas más rápidas de crear una propiedad de dependencia estándar sin tener que recordar la sintaxis es usar el fragmento "propdp" escribiendo `propdp` y luego presionando `Tab` . Se insertará un fragmento de código que luego se puede modificar para satisfacer sus necesidades:

```
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));
}
```

Debe `desplazarse` por las diferentes partes del fragmento de código para realizar los cambios necesarios, incluida la actualización del nombre de la propiedad, el tipo de propiedad, el tipo de clase y el valor predeterminado.

Convenciones importantes

Hay algunas convenciones / reglas importantes a seguir aquí:

1. **Cree una propiedad CLR para la propiedad de dependencia.** Esta propiedad se utiliza en el código subyacente de su objeto o por otros consumidores. Debe invocar `GetValue` y `SetValue` para que los consumidores no tengan que hacerlo.

2. **Nombre la propiedad de dependencia correctamente.** El campo `DependencyProperty` debe ser `public static readonly` . Debe tener un nombre que se corresponda con el nombre de la propiedad CLR y termine con "Propiedad", por ejemplo, `Text` y `TextProperty Text` .

3. **No agregue lógica adicional al definidor de la propiedad CLR.** El sistema de propiedades de dependencia (y XAML específicamente) no hace uso de la propiedad CLR. Si desea realizar una acción cuando cambia el valor de la propiedad, debe proporcionar una devolución de llamada a través de `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.
   }
   ```

Modo de encuadernación

Para eliminar la necesidad de especificar `Mode=TwoWay` en los enlaces (similar al comportamiento de `TextBox.Text`), actualice el código para usar `FrameworkPropertyMetadata` lugar de `PropertyMetadata` y especifique la `TextBox.Text` correspondiente:

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

```
FrameworkPropertyMetadataOptions.BindsTwoWayByDefault));
```

Propiedades de dependencia adjuntas

Cuándo usar

Una propiedad adjunta es una propiedad de dependencia que se puede aplicar a cualquier `DependencyObject` para mejorar el comportamiento de varios controles o servicios que son conscientes de la existencia de la propiedad.

Algunos casos de uso para propiedades adjuntas incluyen:

1. Tener un elemento padre iterar a través de sus hijos y actuar sobre los niños de cierta manera. Por ejemplo, el control `Grid` usa las `Grid.Row` `Grid.Column` `Grid.Row` , `Grid.Column` , `Grid.RowSpan` y `Grid.ColumnSpan` para organizar los elementos en filas y columnas.
2. Agregar elementos visuales a controles existentes con plantillas personalizadas, por ejemplo, agregar marcas de agua a cuadros de texto vacíos en toda la aplicación sin tener que subclasificar `TextBox` .
3. Proporcionar un servicio o característica genérica a algunos o todos los controles existentes, por ejemplo, `ToolTipService` o `FocusManager` . Estos se conocen comúnmente como *comportamientos adjuntos* .
4. Cuando se requiere la herencia del árbol visual, por ejemplo, similar al comportamiento de `DataContext` .

Esto demuestra aún más lo que está sucediendo en el caso de uso de `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` no es una propiedad que exista en `Label` o `TextBox` . Más bien, el control de `Grid` examina sus elementos secundarios y los organiza de acuerdo con los valores de las propiedades adjuntas.

Como definir

Continuaremos usando `Grid` para este ejemplo. La definición de `Grid.Column` se muestra a continuación, pero se excluye `DependencyPropertyChangedEventHandler` por brevedad.

```
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));
}
```

Debido a que las propiedades adjuntas se pueden adjuntar a una amplia variedad de elementos, no se pueden implementar como propiedades CLR. Un par de métodos estáticos se introduce en su lugar.

Por lo tanto, a diferencia de las propiedades de dependencia estándar, las propiedades adjuntas también se pueden definir en clases que no se derivan de `DependencyObject` .

Las mismas convenciones de nomenclatura que se aplican a las propiedades de dependencia regulares también se aplican aquí: la propiedad de dependencia `RowProperty` tiene los métodos correspondientes `GetRow` y `SetRow` .

Advertencias

Como se documenta en MSDN :

> Aunque la herencia de valor de propiedad puede parecer que funciona para propiedades de dependencia no adjuntas, el comportamiento de herencia para una propiedad no asociada a través de ciertos límites de elementos en el árbol de tiempo de ejecución no está definido. Siempre use RegisterAttached para registrar propiedades donde especifique Herencias en los metadatos.

Propiedades de dependencia de solo lectura

Cuándo usar

Una propiedad de dependencia de solo lectura es similar a una propiedad de dependencia normal, pero está estructurada para no permitir que su valor se establezca desde fuera del control. Esto funciona bien si tiene una propiedad que es meramente informativa para los consumidores, por ejemplo, `IsMouseOver` o `IsKeyboardFocusWithin` .

Como definir

Al igual que las propiedades de dependencia estándar, una propiedad de dependencia de solo lectura debe definirse en una clase que se derive 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); }
    }
}
```

Las mismas convenciones que se aplican a las propiedades de dependencia regulares también se aplican aquí, pero con dos diferencias clave:

1. `DependencyProperty` se obtiene de un `DependencyPropertyKey` `private` .
2. El establecedor de propiedades CLR está `protected` o es `private` lugar de `public` .

Tenga en cuenta que el `MyPropertyPropertyKey` pasa `MyPropertyPropertyKey` y no `MyPropertyProperty` al método `SetValue` . Debido a que la propiedad se definió como de solo lectura, cualquier intento de usar `SetValue` en la propiedad debe usarse con una sobrecarga que recibe `DependencyPropertyKey` ; de lo contrario, se `InvalidOperationException` una `InvalidOperationException` .

Lea Propiedades de dependencia en línea: https://riptutorial.com/es/wpf/topic/2914/propiedades-de-dependencia

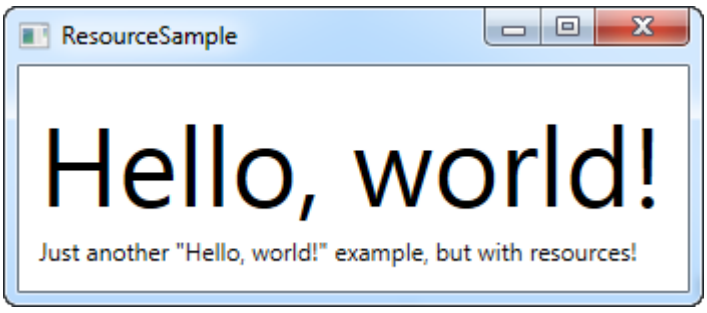
Capítulo 19: Recursos WPF

Examples

Hola Recursos

WPF presenta un concepto muy útil: la capacidad de almacenar datos como un recurso, ya sea localmente para un control, localmente para toda la ventana o globalmente para toda la aplicación. Los datos pueden ser prácticamente lo que desee, desde información real hasta una jerarquía de controles de WPF. Esto le permite colocar datos en un lugar y luego usarlos desde o varios otros lugares, lo cual es muy útil. El concepto se usa mucho para estilos y plantillas.

```
<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>
```



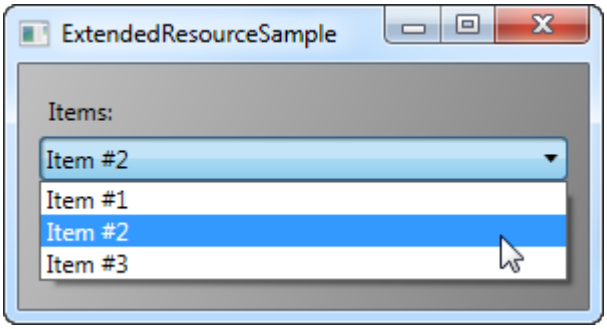
Los recursos reciben una clave, utilizando el atributo x: Key, que le permite hacer referencia a él desde otras partes de la aplicación utilizando esta clave, en combinación con la extensión de marcado StaticResource. En este ejemplo, simplemente almaceno una cadena simple, que luego uso de dos controles TextBlock diferentes.

Tipos de recursos

Compartir una cadena simple fue fácil, pero puedes hacer mucho más. En este ejemplo, también almacenaré una gama completa de cadenas, junto con un pincel de degradado que se utilizará para el fondo. Esto debería darle una buena idea de cuánto puede hacer con los recursos:

```
<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>
```



Esta vez, hemos agregado un par de recursos adicionales, de modo que nuestra Ventana ahora contiene una cadena simple, una matriz de cadenas y un LinearGradientBrush. La cadena se usa para la etiqueta, la matriz de cadenas se usa como elementos para el control ComboBox y el pincel de degradado se usa como fondo para toda la ventana. Entonces, como puedes ver, casi cualquier cosa puede ser almacenada como un recurso.

Recursos amplios locales y de aplicación

Si solo necesita un recurso determinado para un control específico, puede hacerlo más local agregándolo a este control específico, en lugar de a la ventana. Funciona exactamente de la misma manera, la única diferencia es que ahora solo puede acceder desde el ámbito del control donde lo coloca:

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

En este caso, agregamos el recurso al StackPanel y luego lo usamos desde su control

secundario, la Etiqueta. Otros controles dentro del StackPanel también podrían haberlo usado, al igual que los niños de estos controles infantiles podrían haber accedido a él. Sin embargo, los controles fuera de este StackPanel en particular no tendrían acceso a él.

Si necesita la posibilidad de acceder al recurso desde varias ventanas, esto también es posible. El archivo App.xaml puede contener recursos como la ventana y cualquier tipo de control WPF, y cuando los almacena en App.xaml, se puede acceder a ellos de forma global en todas las ventanas y controles de usuario del proyecto. Funciona exactamente de la misma manera que cuando se almacena y se usa desde una ventana:

```
<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>
```

Usarlo también es lo mismo: WPF subirá automáticamente el alcance, desde el control local a la ventana y luego a App.xaml, para encontrar un recurso determinado:

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

Recursos de Code-behind

En este ejemplo, accederemos a tres recursos diferentes de Code-behind, cada uno almacenado en un alcance diferente

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>
```

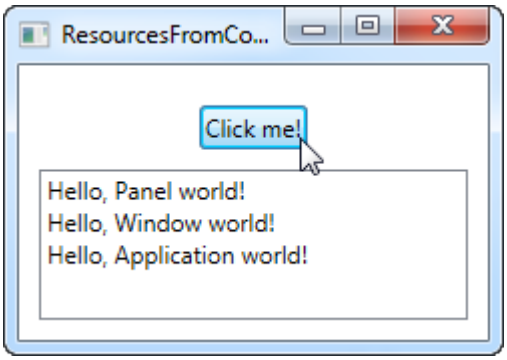
Ventana:

```
<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>
```

Código detrás:

```
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());
                }
        }
}
```



Entonces, como puedes ver, almacenamos tres "Hola, mundo!" mensajes: uno en App.xaml, uno dentro de la ventana y uno localmente para el panel principal. La interfaz consiste en un botón y un ListBox.

En Code-behind, manejamos el evento de clic del botón, en el que agregamos cada una de las cadenas de texto al ListBox, como se ve en la captura de pantalla. Usamos el método FindResource (), que devolverá el recurso como un objeto (si se encuentra), y luego lo convertimos en la cadena que sabemos que es usando el método ToString ().

Observe cómo utilizamos el método FindResource () en diferentes ámbitos: primero en el panel, luego en la ventana y luego en el objeto Aplicación actual. Tiene sentido buscar el recurso donde sabemos que está, pero como ya se mencionó, si no se encuentra un recurso, la búsqueda avanza hacia la jerarquía, por lo que, en principio, podríamos haber utilizado el método FindResource () en el panel en los tres casos, ya que habría continuado hasta la ventana y luego hasta el nivel de la aplicación, si no se hubiera encontrado.

Lo mismo no ocurre al revés: la búsqueda no navega hacia abajo en el árbol, por lo que no puede comenzar a buscar un recurso en el nivel de la aplicación, si se ha definido localmente para el control o para la ventana.

Lea Recursos WPF en línea: https://riptutorial.com/es/wpf/topic/4371/recursos-wpf

Capítulo 20: Síntesis del habla

Introducción

En el ensamblado `System.Speech` , Microsoft ha agregado **Speech Synthesis** , la capacidad de transformar texto en palabras habladas.

Sintaxis

1. SpeechSynthesizer speechSynthesizerObject = new SpeechSynthesizer ();

   speechSynthesizerObject.Speak ("Text to Speak");

Examples

Ejemplo de síntesis de voz - Hola mundo

```
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!");
                }
        }
}
```

Lea Síntesis del habla en línea: https://riptutorial.com/es/wpf/topic/8368/sintesis-del-habla

Capítulo 21: Soporta transmisión de video y asignación de píxeles a un control de imagen

Parámetros

Parámetros	Detalles
PixelHeight (System.Int32)	La altura de la imagen en unidades de píxeles de imagen.
PixelWidth (System.Int32)	El ancho de la imagen en unidades de píxeles de imagen.
PixelFormat (System.Windows.Media.PixelFormat)	El ancho de la imagen en unidades de píxeles de imagen.
Pixeles	Cualquier cosa que implemente IList <T>, incluida la matriz de bytes C #
DpiX	Especifica la Dpi horizontal - Opcional
DpiY	Especifica el Dpi vertical - Opcional

Observaciones

- Asegúrese de hacer referencia al *ensamblado System.Windows.Interactivity* , para que el analizador XAML reconozca la declaración *xmlns: i* .
- Tenga en cuenta que la instrucción *xmlns: b* coincide con el espacio de nombres donde reside la implementación del comportamiento
- El ejemplo supone un conocimiento práctico de las expresiones de enlace y XAML.
- Este comportamiento admite la asignación de píxeles a una imagen en forma de una matriz de bytes, incluso aunque el tipo de Propiedad de dependencia esté especificado como un *IList* . Esto funciona ya que la matriz de bytes C # implementa el *IList* interfaz.
- El comportamiento logra un rendimiento muy alto y se puede utilizar para transmisión de video
- No asigne la propiedad de dependencia de *origen de* la imagen: enlace a la propiedad de dependencia de *píxeles en su* lugar
- Las propiedades Pixels, PixelWidth, PixelHeight y PixelFormat deben asignarse para que los píxeles se procesen.
- Orden de Dependencia la asignación de propiedad no importa

Examples

Implementación del comportamiento

```
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;
    }
}
}
```

Uso de 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>
```

Lea Soporta transmisión de video y asignación de píxeles a un control de imagen en línea: https://riptutorial.com/es/wpf/topic/6435/soporta-transmision-de-video-y-asignacion-de-pixeles-a-un-control-de-imagen

Capítulo 22: System.Windows.Controls.WebBrowser

Introducción

Esto le permite poner un navegador web en su aplicación WPF.

Observaciones

Un punto clave a tener en cuenta, que no es obvio en la documentación, y que podría pasar años sin saberlo es que se comporta por defecto como InternetExplorer7, en lugar de su instalación más actualizada de InternetExplorer (consulte https: // weblog). west-wind.com/posts/2011/may/21/web-browser-control-specifying-the-ie-version).

Esto no se puede arreglar configurando una propiedad en el control; debe modificar las páginas que se muestran agregando una etiqueta Meta HTML, o aplicando una configuración de registro (!). (Los detalles de ambos enfoques están en el enlace de arriba).

Por ejemplo, este extraño comportamiento de diseño puede llevarlo a recibir un mensaje que dice "Error de secuencia de comandos" / "Ha ocurrido un error en la secuencia de comandos de esta página". Buscar en Google este error podría hacerte pensar que la solución es intentar suprimir el error, en lugar de comprender el problema real y aplicar la solución correcta.

Examples

Ejemplo de un WebBrowser dentro de un BusyIndicator

Tenga en cuenta que el control WebBrowser no simpatiza con su definición de XAML y se representa por encima de otras cosas. Por ejemplo, si lo pones dentro de un BusyIndicator que se ha marcado como ocupado, se seguirá mostrando por encima de ese control. La solución es vincular la visibilidad del WebBrowser al valor que está utilizando el BusyIndicator, y usar un convertidor para invertir el booleano y convertirlo en una visibilidad. Por ejemplo:

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

Lea System.Windows.Controls.WebBrowser en línea: https://riptutorial.com/es/wpf/topic/9115/system-windows-controls-webbrowser

Capítulo 23: Una introducción a los estilos WPF

Introducción

Un estilo permite la modificación completa de la apariencia visual de un control WPF. Aquí hay algunos ejemplos de algunos estilos básicos y una introducción a los diccionarios de recursos y animaciones.

Examples

Estilo de un botón

La forma más fácil de crear un estilo es copiar uno existente y editarlo.

Crea una ventana simple con dos botones:

```
<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>
```

En Visual Studio, la copia se puede hacer haciendo clic derecho en el primer botón del editor y seleccionando "Editar una copia ..." en el menú "Editar plantilla".

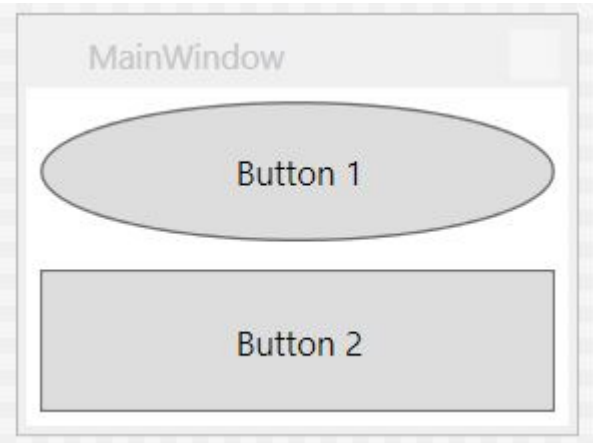
Definir en "Aplicación".

El siguiente ejemplo muestra una plantilla modificada para crear un botón de elipse:

```
<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>
```

El resultado:



Estilo aplicado a todos los botones

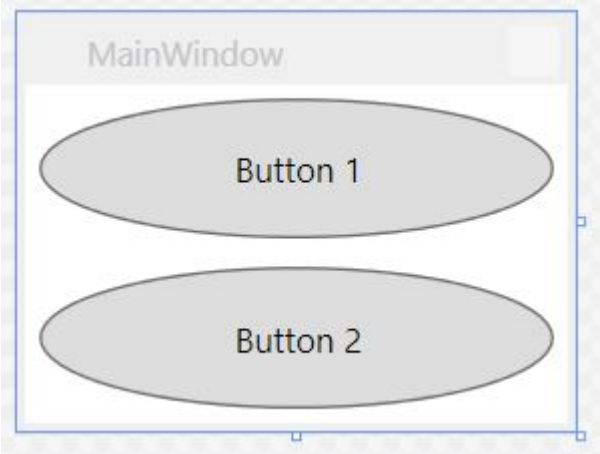
Tomando el ejemplo anterior, al eliminar el elemento x: Key del estilo, se aplica el estilo a todos los botones en el ámbito de la aplicación.

```
<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>
```

Tenga en cuenta que ya no es necesario especificar el estilo para botones individuales:

```
<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>
```

Ambos botones ahora están diseñados.



Diseñando un ComboBox

Comenzando con los siguientes `ComboBox` es:

```
<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>
```

Haga clic derecho en el primer `ComboBox` en el diseñador, elija "Editar plantilla -> Editar una copia". Definir el estilo en el ámbito de aplicación.

Hay 3 estilos creados:

```
ComboBoxToggleButton
ComboBoxEditableTextBox
ComboBoxStyle1
```

Y 2 plantillas:

```
ComboBoxTemplate
ComboBoxEditableTemplate
```

Un ejemplo de edición del estilo `ComboBoxToggleButton` :

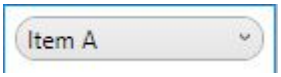
```
<SolidColorBrush x:Key="ComboBox.Static.Border" Color="#FFACACAC"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Border" Color="#FFABADB3"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Button.Background" 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>
```

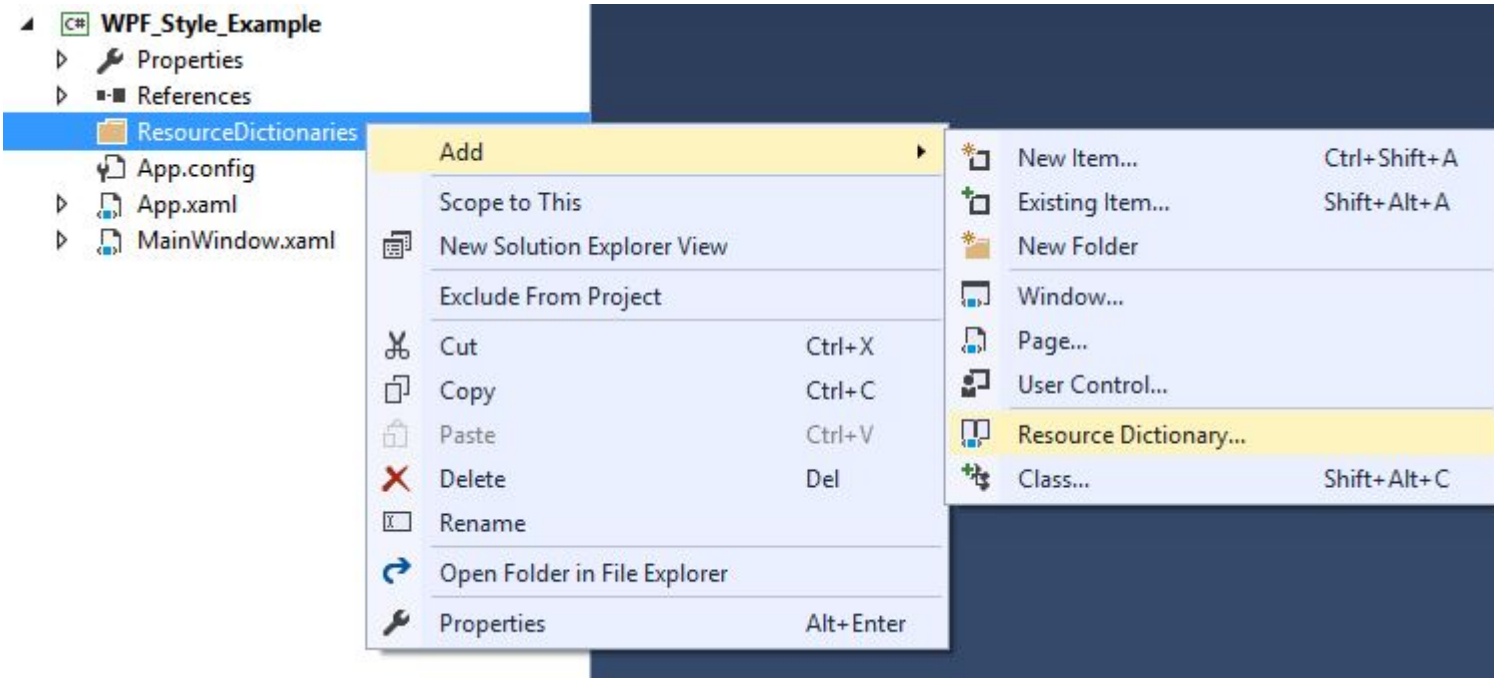
Esto crea un `ComboBox` redondeado que resalta el naranja en el mouse y se vuelve rojo cuando se presiona.



Tenga en cuenta que esto no cambiará el cuadro de control Editable debajo de él; modificación que requiere cambiar el estilo `ComboBoxEditableTextBox` o `ComboBoxEditableTemplate` .

Creando un Diccionario de Recursos

Tener muchos estilos en App.xaml se volverá rápidamente complejo, por lo que se pueden colocar en diccionarios de recursos separados.



Para poder utilizar el diccionario, se debe combinar con App.xaml. Entonces, en App.xaml, después de que se haya creado el diccionario de recursos:

```
<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>
```

Ahora se pueden crear nuevos estilos en Dictionary1.xaml y se puede hacer referencia a ellos como si estuvieran en App.xaml. Después de crear el proyecto, la opción también aparecerá en Visual Studio al copiar un estilo para ubicarlo en el nuevo diccionario de recursos.

Estilo de botón DoubleAnimation

Se ha creado la siguiente `Window` :

```
<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>
```

Se aplicó un estilo (creado en App.xaml) a los botones, que anima el ancho de 200 a 100 cuando el mouse ingresa al control y de 100 a 200 cuando sale:

```
<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>
```

Lea Una introducción a los estilos WPF en línea: https://riptutorial.com/es/wpf/topic/9670/una-

introduccion-a-los-estilos-wpf

Creditos

S. No	Capítulos	Contributors
1	Empezando con wpf	Community, Derpcode, Gusdor, Matthew Cargille, Nasreddine, Sam, Stephen Wilson
2	Afinidad de hilos que accede a los elementos de la interfaz de usuario	Mert Gülsoy
3	Arquitectura WPF	Adi Lester
4	Comportamientos de WPF	Bradley Uffner
5	Control de cuadrícula	Alexander Mandt, vkluge
6	Convertidores de Valor y Multivalor	Adi Lester, Arie, Dalstroem, galakt, ltiveron
7	Creación de UserControls personalizados con enlace de datos	ltiveron, Mage Xy
8	Creando la pantalla de bienvenida en WPF	Grx70, Sam
9	Enlace de barra deslizante: Actualizar solo en arrastre finalizado	Eyal Perry
10	Estilos en WPF	Guttsy, Jakub Lokša
11	Extensiones de marcado	Alexander Pacha, Emad
12	Gatillos	John Strit, Maxim
13	Introducción al enlace de datos WPF	Adi Lester, Arie, Gabor Barat, Guttsy, Ian Wold, Jirajha, vkluge, wkl
14	Localización WPF	Dabblernl
15	MVVM en WPF	Andrew Stephens, Athafoud, Dutts, Felix D., Felix Too, H.B., James LaPenn, Kcvin, kowsky, Matt Klein, RamenChef, STiLeTT, TrBBoI, vkluge
16	Optimización para la interacción táctil	Martin Zikmund
17	Principio de diseño "La mitad del espacio en blanco"	Richardissimo

18	Propiedades de dependencia	Adi Lester, auticus, Clemens, Guttsy
19	Recursos WPF	Elangovan, John Strit, SUB-HDR
20	Síntesis del habla	BKO
21	Soporta transmisión de video y asignación de píxeles a un control de imagen	Eyal Perry
22	System.Windows.Controls.WebBrowser	Richardissimo
23	Una introducción a los estilos WPF	J R