Otros controles comunes II

Contenidos

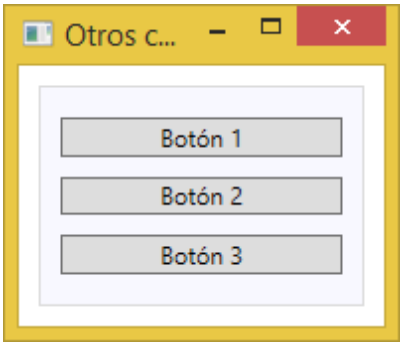
1. Bordes y Fondos

Border es un control que nos permite dibujar un borde o un fondo (o ambos) alrededor de otro elemento. Así se vence la restricción de los paneles WPF, que no permiten dibujar un borde alrededor de las esquinas.

A continuación se muestra un ejemplo de uso:

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Otros controles" Height="170" Width="200">
    <Grid Margin="10" >
        <Border Background="GhostWhite"
                 BorderBrush="Gainsboro" BorderThickness="1">
            <StackPanel VerticalAlignment="Center" Margin="10">
                <Button>Botón 1</Button>
                <Button Margin="0,10">Botón 2</Button>
                <Button>Botón 3</Button>
            </StackPanel>
        </Border>
    </Grid>
</Window>
```

El Border no tiene estilo hasta que definimos un fondo (**Background**) o un borde (color [**BorderBrush**] y grosor [**BorderThickness**])

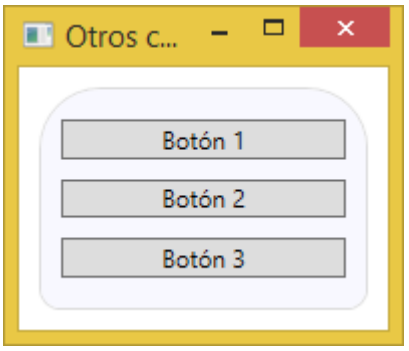


1.1. Otros aspectos a definir

Otros aspectos que podemos definir en los bordes (los aplicaremos sobre el ejemplo anterior) son:

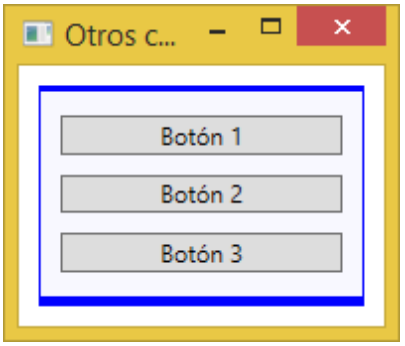
- **Esquinas redondeadas**. Se definen con la propiedad **CornerRadius**, que permiten especificar el radio de curvatura de las esquinas. Podemos especificar un único valor (4 esquinas), dos valores (uno para las esquinas superiores y otro para las inferiores) o 4 valores.

```
<Border Background="GhostWhite" BorderBrush="Gainsboro"
          BorderThickness="1" CornerRadius="30,10">
```



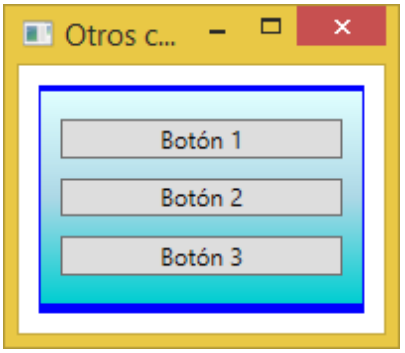
- **Color y grosor de los bordes**. El borde del ejemplo anterior es muy discreto, pero puede cambiar fácilmente regulando el color y/o la anchura. Dado que la propiedad **BorderThickness** es de tipo **Thickness**, podemos manipular cada anchura del borde de manera individual, de dos en dos o de 4 en 4

```
<Border Background="GhostWhite" BorderBrush="Blue"
BorderThickness="1,3,1,5" >
```



- **Color de fondo**. El color de fondo es de tipo **Brush**, lo que abre un montón de posibilidades. Por ejemplo, podemos especificar gradientes. En este caso vamos a especificar un gradiente lineal (**LinearGradientBrush**). Sin entrar a detalles de definición del mismo, para entender el control nos basta con saber que podemos definir varias paradas de color, que son colores por los que pasará el gradiente:

```
<Border BorderBrush="Blue" BorderThickness="1,3,1,5" >
    <Border.Background>
     <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
            <GradientStop Color="LightCyan" Offset="0.0"/>
            <GradientStop Color="LightBlue" Offset="0.5"/>
            <GradientStop Color="DarkTurquoise" Offset="1"/>
        </LinearGradientBrush>
    </Border.Background>
```

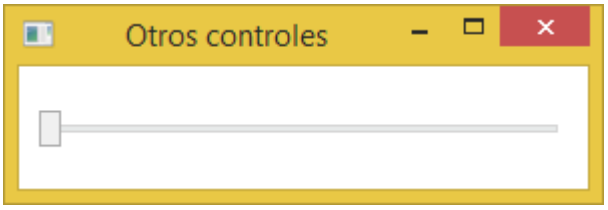


2. Deslizadores

El control **`Slider`** permite tomar un valor numérico arrastrando un cursor a lo largo de una línea horizontal o vertical. En el siguiente ejemplo se muestra en funcionamiento:

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Otros controles" Height="100" Width="300">
    <StackPanel VerticalAlignment="Center" Margin="10">
        <Slider Maximum="100" />
    </StackPanel>
</Window>
```

El presente deslizador permite al usuario escoger un valor desplazando el cursor a lo largo de la línea.



Para que tenga un poco más de sentido, agregaremos los elementos descritos a continuación

2.1. Marcas

En el ejemplo anterior es difícil ver el valor exacto. Una manera de hacerlo es habilitar marcas (*ticks*), que nos permiten apreciar mejor la posición del deslizador:

```
<Slider Maximum="100" TickPlacement="BottomRight" TickFrequency="10"/>
```

Las marcas se habilitan estableciendo la propiedad **`TickPlacement`** con una propiedad distinta a **`None`**, que es el valor por defecto. Además de bajo la línea (**`BottomRight`**), podemos ponerlas encima (**`TopLeft`**) o incluso a ambos (**`Both`**).

También se usa la propiedad **`TickFrequency`** para establecer la frecuencia. Su valor por defecto es 1.

2.2. Ajustando a las marcas

En el ejemplo anterior se puede comprobar que hemos dejado el cursor en un valor entre marcas. Esto es posible, dado que hay 10 valores entre cada marca. El valor por defecto es un doble, lo que significa que el valor puede ser no entero (y es muy probable que lo sea).

Este comportamiento puede cambiarse con el atributo booleano **`IsSnapToTickEnabled`**

```
<Slider Maximum="100" TickPlacement="BottomRight" TickFrequency="10"
  IsSnapToTickEnabled="True" >
```

Esto nos asegura que el cursor sólo puede ir en un valor de los establecidos en una etiqueta (en este ejemplo 0,10,20,30,40… hasta 100.

2.3. Valor del deslizador

Hasta ahora se ha usado el Slider con fines ilustrativos, pero el objeto real del mismo es leer su valor y usarlo para algo. Para este fin, el control tiene una propiedad `Value` a la cual podemos tanto acceder desde el *Code-Behind* como enlazar desde otro control.

Un escenario común consiste en combinar el Slider con un **`TextBox`**, que permite al usuario comprobar el valor que hemos seleccionado, además de cambiarlo introduciendo un número en lugar de arrastrar el cursor.

Un modo de lograr este comportamiento es ejecutar eventos de cambio tanto en el Slider como en el **`TextBox`** y luego actualizarlos. Sin embargo, la misma tarea puede realizarse fácilmente con un **`Binding`**:

```
    <Window x:Class="OtrosControles.VentanaPrincipal"
            xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            xmlns:system="clr-namespace:System;assembly=mscorlib"
            Title="Otros controles" Height="100" Width="300">
        <DockPanel VerticalAlignment="Center" Margin="10">
            <TextBox DockPanel.Dock="Right" TextAlignment="Right" Width="40"
Text="{Binding ElementName=sliValor, Path=Value, UpdateSourceTrigger=PropertyChanged}" />
            <Slider Name="sliValor" Maximum="100"
          TickPlacement="BottomRight" TickFrequency="10" IsSnapToTickEnabled="True"/>
        </DockPanel>
    </Window>
```

Ahora podemos cambiar el valor, bien usando el deslizador o introduciendo un valor en el **TextBox**, lo que se reflejará inmediatamente en el otro control. Un valor añadido es que obtenemos una validación simple sin necesidad de añadir código, por ejemplo en el caso de introducir valores no numéricos:

2.4. Respondiendo a los valores cambiados

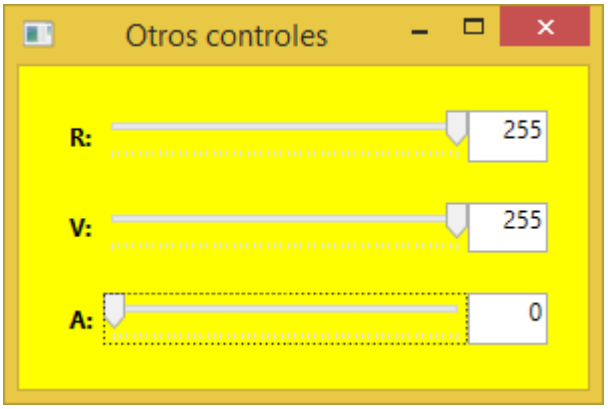
Aunque los *bindings* facilitan mucho las cosas, podemos responder a los cambios en el Slider desde el *Code-Behind*. El control añade un evento **ValueChanged** que nos permite compobar si el valor ha cambiado.

Para ilustrar este comportamiento, se ha creado un ejemplo con 3 deslizadores que representan los valores rojo, verde y azul (RGB) de un color:

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Otros controles" Height="200" Width="300">
    <StackPanel Margin="10" VerticalAlignment="Center">
        <DockPanel VerticalAlignment="Center" Margin="10">
            <Label DockPanel.Dock="Left" FontWeight="Bold">R:</Label>
            <TextBox
Text="{Binding ElementName=sliRojo, Path=Value, UpdateSourceTrigger=PropertyChanged}"
             DockPanel.Dock="Right" TextAlignment="Right" Width="40" />
            <Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5"
 IsSnapToTickEnabled="True" Name="sliRojo" ValueChanged="ColorSlider_ValueChanged" />
        </DockPanel>
```

```
            <DockPanel VerticalAlignment="Center" Margin="10">
                <Label DockPanel.Dock="Left" FontWeight="Bold">V:</Label>
                <TextBox
Text="{Binding ElementName=sliVerde, Path=Value, UpdateSourceTrigger=PropertyChanged}"
                  DockPanel.Dock="Right" TextAlignment="Right" Width="40" />
                <Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5"
       IsSnapToTickEnabled="True" Name="sliVerde"  ValueChanged="ColorSlider_ValueChanged"
      />
            </DockPanel>

            <DockPanel VerticalAlignment="Center" Margin="10">
                <Label DockPanel.Dock="Left" FontWeight="Bold">A:</Label>
                <TextBox      Text="{Binding      ElementName=sliAzul,      Path=Value,
UpdateSourceTrigger=PropertyChanged}"       DockPanel.Dock="Right"      TextAlignment="Right"
Width="40" />
                <Slider   Maximum="255"   TickPlacement="BottomRight"   TickFrequency="5"
      IsSnapToTickEnabled="True" Name="sliAzul" ValueChanged="ColorSlider_ValueChanged" />
            </DockPanel>
        </StackPanel>
      </Window>
```

```
      namespace OtrosControles
      {

          public partial class VentanaPrincipal : Window
          {
              public VentanaPrincipal()
              {
                  InitializeComponent();
              }

              private void ColorSlider_ValueChanged(object sender,
                                      RoutedPropertyChangedEventArgs<double> e)
              {
                   Color color = Color.FromRgb((byte)sliRojo.Value,
                                               (byte)sliVerde.Value,
                                                (byte)sliAzul.Value);
                               this.Background = new
                              SolidColorBrush(color);

              }
```



Tenemos tres **DockPanel**, cada uno con una etiqueta, un deslizador y una caja de texto. Cada caja de texto se ha enlazado a su control correspondiente.

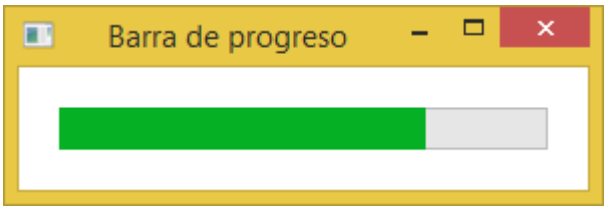
Cada deslizador se suscribe al mismo evento **ValueChanged**, en el cual creamos una nueva instancia de Color basándonos en los valores seleccionados. Posteriormente usamos dicho color para crear un nuevo **SolidColorBrush** que asignamos al fondo (**Background**) de la ventana.

3. Barras de progreso

WPF incluye un control que permite mostrar el progreso de una tarea llamado **ProgressBar**. El control permite establecer un valor mínimo y máximo y posteriormente un valor, que permite dar un indicativo visual acerca del progreso del proceso en que nos encontramos

El siguiente ejemplo básico muestra una barra de progreso:

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Barra de progreso" Height="100" Width="300">
    <Grid Margin="20">
        <ProgressBar Minimum="0" Maximum="100" Value="75" />
    </Grid>
</Window>
```



Se ha usado un enfoque bastante estándar de mostrar un porcentaje (un valor entre 0 y 100%), dando un valor inicial de 75. Otra opción es dar un valor máximo y mínimo de las tareas que estamos realizando. Por ejemplo, si estamos comprobando un conjunto de archivos, podemos establecer el atributo **Minimum** a 0 y **Maximum** al número de archivos que procesamos. A continuación incrementamos la barra a medida que iteramos sobre los archivos.

3.1. Mostrando el progreso de una tarea

En muchas ocasiones queremos usar la barra de progreso para una tarea que puede llevar tiempo. Sin embargo esto puede acarrear un problema: Si llevamos a cabo un proceso pesado en el hilo que maneja la interfaz de usuario, mientras intentamos actualizar la misma a la vez (por ejemplo para actualizar la barra) veremos que no es posible llevar ambas a cabo en el mismo hilo.

Veámoslo con el siguiente ejemplo:

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Barra de progreso" Height="100" Width="300"
        ContentRendered="Window_ContentRendered">
    <Grid Margin="20">
        <ProgressBar Minimum="0" Maximum="100" Name="pbEstado"/>
    </Grid>
</Window>
```

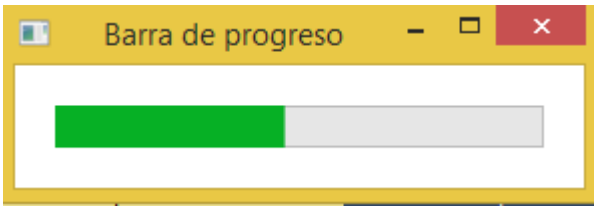
```
using System.Threading;

namespace OtrosControles
{
    public partial class VentanaPrincipal : Window
    {
        public VentanaPrincipal()
        {
            InitializeComponent();
        }

        private void Window_ContentRendered(object sender, EventArgs e)
        {
            for (int i = 0; i < 100; i++)
            {
                pbEstado.Value = i;
                Thread.Sleep(20);
            }
        }
    }
}
```

En este ejemplo, en cuanto se ha cargado la ventana hacemos un bucle que va de 0 a 100. En cada iteración incrementamos el valor de la barra de progreso. Dado que la velocidad sería demasiado rápida (no nos daríamos cuenta de que se realiza el bucle), añadimos en cada iteración un pequeño retardo de 20 milisegundos. Sin embargo, si ejecutamos el ejemplo veremos no ocurre nada hasta que el bucle ha acabado, tras lo cual vemos directamente la barra llena. Es decir, no vemos el progreso de la operación

Para evitar este comportamiento, tendremos que realizar la tarea en un hilo específico, y notificar las actualizaciones al hilo encargado de la interfaz de usuario, que será capaz de procesar y mostrar las actualizaciones. Para ello podemos usar una clase llamada **`BackgroundWorker`** cuyo uso se muestra a continuación (no se muestra el archivo XAML por ser igual al del ejemplo anterior:

```
using System.Threading;
using
System.ComponentModel;

namespace OtrosControles
{

    public partial class VentanaPrincipal : Window
    {
        public VentanaPrincipal()
        {
            InitializeComponent();
        }

        private void Window_ContentRendered(object sender, EventArgs e)
        {
            BackgroundWorker worker = new BackgroundWorker();
            worker.WorkerReportsProgress  =  true;
            worker.DoWork += worder_DoWork;
            worker.ProgressChanged += worker_Progress_Changed;

            worker.RunWorkerAsync();
        }

     private void worker_Progress_Changed(object sender, ProgressChangedEventArgs e)
        {
            pbEstado.Value = e.ProgressPercentage;
        }

        private void worder_DoWork(object sender, DoWorkEventArgs e)
        {
            for (int i = 0; i < 100; i++)
            {
                (sender as BackgroundWorker).ReportProgress(i);
                Thread.Sleep(50);
            }
        }
    }
```

Como se puede ver, ahora la barra se actualiza poco a poco:



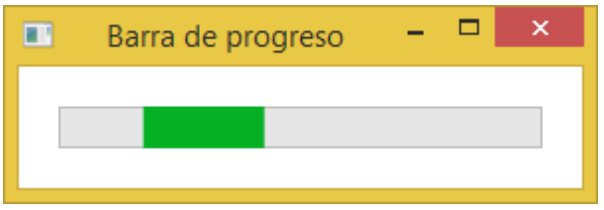
3.2. Indeterminación

Para algunas tareas, expresar el progreso como un porcentaje no es posible (o simplemente no sabemos cuánto tiempo llevará). Para estas situaciones, se utilizan barras indeterminadas, donde una animación hace saber al usuario que algo está pasando, a la vez que se indica que el tiempo de ejecución no puede ser especificado.

Este tipo de barras se especifican a través del atributo **`IsIndeterminate`**:

```
      <Window x:Class="OtrosControles.VentanaPrincipal"
              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
              " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
              xmlns:system="clr-namespace:System;assembly=mscorlib"
              Title="Barra de progreso" Height="100" Width="300">
          <Grid Margin="20">
<ProgressBar Minimum="0" Maximum="100" Name="pbEstado" IsIndeterminate="True"/>
          </Grid>
      </Window>
```

En este caso el indicador de progreso no va fijado a ninguno de los lados: Se mueve libremente de izquierda a derecha y posteriormente comienza de nuevo:



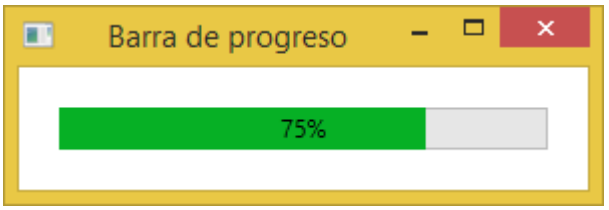
3.3. Barra de progreso con texto

La flexibilidad de WPF hace que sea muy fácil representar un texto asociado a la barra de proceso, como muestra el siguiente ejemplo:

```
    <Window x:Class="OtrosControles.VentanaPrincipal"
            xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
            " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            xmlns:system="clr-namespace:System;assembly=mscorlib"
            Title="Barra de progreso" Height="100" Width="300">
        <Grid Margin="20">
            <ProgressBar Minimum="0" Maximum="100" Name="pbEstado" Value="75" />
            <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
Text="{Binding ElementName=pbEstado, Path=Value, StringFormat={}{0:0}% }" />
        </Grid>
    </Window>
```

En este ejemplo ponemos una barra de progreso y un bloque de texto en el mismo **`Grid`**, sin especificar filas o columnas. Esto muestra ambos controles en el mismo lugar, que es lo que queremos dado que el **`TextBlock`** tiene un fondo transparente por defecto.

Por otra parte usamos un *binding* para asegurar que el texto del **`TextBlock`** es el mismo que el valor del **`ProgressBar`**. Nótese la sintaxis usada para definir la propiedad **`StringFormat`**, que permite mostrar el valor con el porcentaje.



4. El control WebBrowser

WPF añade un control **`WebBrowser`** que permite albergar un navegador Web completo en la aplicación. El navegador es una versión simplificada de Internet Explorer, pero dado que es una parte de Windows, la aplicación debería funcionar en máquinas Windows sin requerir componentes adicionales.

Vamos a ver un ejemplo donde se muestra el control:

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Navegador" Height="600" Width="900">
    <Window.CommandBindings>
        <CommandBinding Command="NavigationCommands.BrowseBack"
       CanExecute="BrowseBack_CanExecute" Executed="BrowseBack_Executed" />
        <CommandBinding Command="NavigationCommands.BrowseForward"
       CanExecute="BrowseForward_CanExecute" Executed="BrowseForward_Executed" />
        <CommandBinding Command="NavigationCommands.GoToPage"
       CanExecute="GoToPage_CanExecute" Executed="GoToPage_Executed" />
    </Window.CommandBindings
                >
    <DockPanel>
        <ToolBar DockPanel.Dock="Top">
            <Button Command="NavigationCommands.BrowseBack">
                <Image Source="/OtrosControles;component/Images/arrow_left.png"
                    Width="16" Height="16" />
            </Button>
            <Button Command="NavigationCommands.BrowseForward">
                <Image Source="/OtrosControles;component/Images/arrow_right.png"
                    Width="16" Height="16" />
            </Button>
            <Separator />
            <TextBox Name="txtURL" Width="300" KeyUp="txtURL_KeyUp" />
            <Button Command="NavigationCommands.GoToPage">
                <Image Source="/OtrosControles;component/Images/world_go.png"
                    Width="16" Height="16" />
            </Button>
        </ToolBar>
        <WebBrowser Name="wbEjemplo"
             Navigating="wbEjemplo_Navigating"></WebBrowser>
    </DockPanel>
</Window>
```

```
namespace OtrosControles
{
    public partial class VentanaPrincipal : Window
    {
        public VentanaPrincipal()
        {
            InitializeComponent();
            wbEjemplo.Navigate("http://www.ies1.com");
        }
        private void txtURL_KeyUp(object sender, KeyEventArgs e)
        {
            if (e.Key == Key.Enter)
                wbEjemplo.Navigate(txtURL.Text);
        }
        private void wbEjemplo_Navigating(object sender, NavigatingCancelEventArgs e)
        {
            txtURL.Text = e.Uri.OriginalString;
        }
        private void BrowseBack_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute=((wbEjemplo!=null) && (wbEjemplo.CanGoBack));
        }
        private void BrowseForward_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = ((wbEjemplo != null) && (wbEjemplo.CanGoForward));
        }
        private void GoToPage_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
        private void GoToPage_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            wbEjemplo.Navigate(txtURL.Text);
        }
        private void BrowseForward_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            wbEjemplo.GoForward();
        }
        private void BrowseBack_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            wbEjemplo.GoBack();
        }
    }
}
```

En el ejemplo se han usado iconos de la librería Silk Icon Set, que ya se ha mencionado anteriormente.

En el XAML vemos una barra de herramientas con un par de botones para ir adelante y atrás, además de una barra de direcciones que nos va a permitir introducir y mostrar la URL, y un botón para navegar a la URL introducida. En los botones hacemos uso de comandos predefinidos para tratar las acciones.

Bajo la barra de herramientas tenemos el control **`WebBrowser`**. Desde el mismo nos suscribimos al evento **`Navigating`**, que ocurre en cuanto el navegador comienza a navegar a una URL.

En el *Code Behind* comenzamos navegando a una URL indicada en el constructor de la ventana, para así tener algo que mostrar en lugar de un control en blanco. También tenemos el evento **`txtURL_KeyUp`**, que utilizamos para comprobar si el usuario ha pulsado Enter en la barra de direcciones. De ser así, navegamos a la URL introducida.

El evento **`wbEjemplo_Navigating`** actualiza la barra de herramientas cada vez que comienza la navegación. Esto es importante, dado que queremos mostrar la URL independientemente de que el usuario haya comenzado la navegación introduciendo una URL o pulsando un enlace en la página.

La última parte del *Code-Behind* simplemente sirve para tratar los comandos:

- Para tratar los botones de *Atrás* y *Adelante* usamos las propiedades **`CanGoBack`** y **`CanGoForward`** de **`WebBrowser`** que permiten determinar si se pueden ejecutar. Los métodos del mismo nombre realizan el trabajo.
- Para el último botón, permitimos ejecutarlo siempre, y cuando lo hace usamos de nuevo el método **`Navigate()`**.

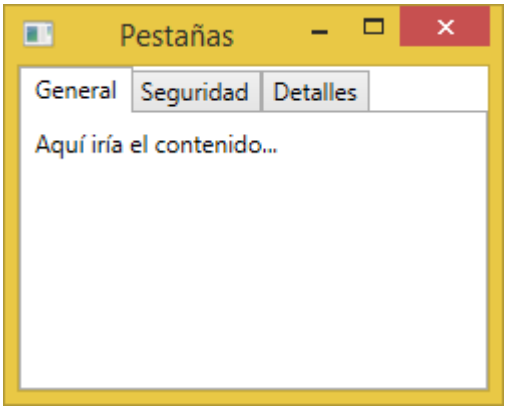
5. Uso de pestañas

El componente **TabControl** nos permite dividir la interface en distintas áreas, cada una accesible pulsando la cabecera de la pestaña (que habitualmente se posiciona en la parte superior del control). Este tipo de controles suelen usarse en aplicaciones Windows o también en las propias interfaces de Windows, como el diálogo de propiedades para archivos/directorios, etc..

Como ocurre con la mayor parte de controles WPF, es muy fácil comenzar con el **TabControl**:

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Pestañas" Height="200" Width="250">

    <Grid>
        <TabControl>
            <TabItem Header="General">
                <Label Content="Aquí iría el contenido..." />
            </TabItem>
            <TabItem Header="Seguridad" />
            <TabItem Header="Detalles" />
        </TabControl>
    </Grid>
</Window>
```

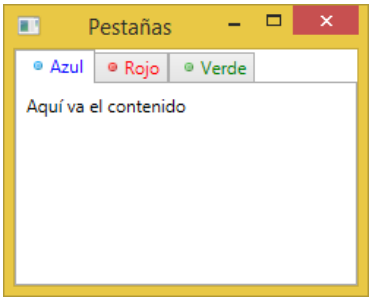
Cada pestaña se representa con un elemento **TabItem**, donde el texto que se muestra se controla con la propiedad **Header**. El **TabItem** viene de la clase **ContentControl**, lo que implica que podemos definir un único elemento dentro de él que se muestra si el elemento está activo. En este ejemplo hemos usado una única etiqueta, pero si queremos ubicar más de un elemento en la pestaña usamos un panel con elementos hijos dentro.



5.1. Cabeceras a medida

Como siempre, WPF es muy flexible cuando queremos personalizar el aspecto de las pestañas. Esto se refiere tanto al contenido de las pestañas como a las cabeceras, como muestra el siguiente ejemplo:

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        " xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Pestañas" Height="200" Width="250">

        <Grid>
        <TabControl>
            <TabItem>
                <TabItem.Header>
                    <StackPanel Orientation="Horizontal">
                   <Image Source="/OtrosControles;component/Images/bullet_blue.png" />
                            <TextBlock Text="Azul" Foreground="Blue" />
                    </StackPanel>
                </TabItem.Header>
                <Label Content="Aquí va el contenido" />
            </TabItem>
            <TabItem>
                <TabItem.Header>
                    <StackPanel Orientation="Horizontal">
                  <Image Source="/OtrosControles;component/Images/bullet_red.png" />
                            <TextBlock Text="Rojo" Foreground="Red" />
                    </StackPanel>
                </TabItem.Header>
            </TabItem>
            <TabItem>
                <TabItem.Header>
                    <StackPanel Orientation="Horizontal">
                   <Image Source="/OtrosControles;component/Images/bullet_green.png"
                                                    />
                            <TextBlock Text="Verde" Foreground="Green" />
                    </StackPanel>
                </TabItem.Header>
            </TabItem>
        </TabControl>
    </Grid>
</Window>
```



Pese a la cantidad de código, una vez que nos adentramos en él es simple. Cada atributo **Header** contiene un tiene un **StackPanel**, que contiene a su vez una imagen y un **TextBlock**. Esto nos permite tener una imagen en cada pestaña, además de personalizar el color del texto.

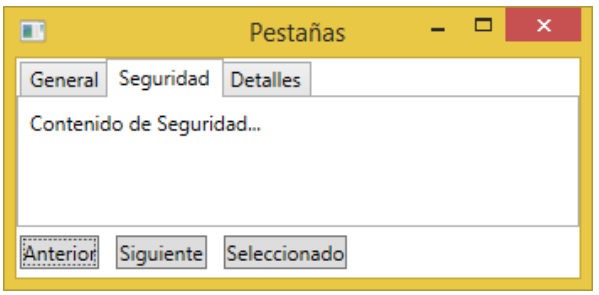
5.2. Controlando las pestañas

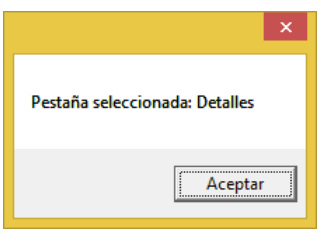
En ocasiones queremos controlar qué pestaña se habilita de manera programática, u obtener información adicional acerca de la pestaña seleccionada. El control **TabControl** tiene algunas propiedades que permiten esto, incluyendo **SelectedIndex** y **SelectedItem**. En el siguiente ejemplo hemos añadido unos ejemplos al primer ejemplo que nos permiten manejar el **TabControl** (se muestra primero el *CodeBehind* y luego el XAML):

```
namespace OtrosControles
{
    public partial class VentanaPrincipal : Window
    {
        public VentanaPrincipal()
        {
            InitializeComponent();
        }
        private void btnAnterior_Click(object sender, RoutedEventArgs e)
        {
            int indiceNuevo = tcEjemplo.SelectedIndex - 1;
            if (indiceNuevo < 0)
                indiceNuevo = tcEjemplo.Items.Count-1;
            tcEjemplo.SelectedIndex = indiceNuevo;
        }
        private void btnSiguiente_Click(object sender, RoutedEventArgs e)
        {
            int indiceNuevo = tcEjemplo.SelectedIndex + 1;
            if (indiceNuevo >= tcEjemplo.Items.Count)
                indiceNuevo = 0;
            tcEjemplo.SelectedIndex = indiceNuevo;
        }
        private void btnSeleccion_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("
          Pestaña seleccionada: " + (tcEjemplo.SelectedItem as TabItem).Header);
        }
    }
}
```

```
<Window x:Class="OtrosControles.VentanaPrincipal"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="Pestañas" Height="200" Width="350">
        <DockPanel>
            <StackPanel Orientation="Horizontal"
                      DockPanel.Dock="Bottom" Margin="2,5">
                <Button Name="btnAnterior"
              Click="btnAnterior_Click">Anterior</Button>
                <Button Name="btnSiguiente" Margin="10,0"
               Click="btnSiguiente_Click">Siguiente</Button>
                <Button Name="btnSeleccion"
               Click="btnSeleccion_Click">Seleccionado</Button>
            </StackPanel>
            <TabControl Name="tcEjemplo">
                <TabItem Header="General">
                    <Label Content="Contenido de General..." />
                </TabItem>
            <TabItem Header="Seguridad" >
                <Label Content="Contenido de Seguridad..." />
            </TabItem>

            <TabItem Header="Detalles">
                <Label Content="Contenido de Detalles..." />
            </TabItem>
        </TabControl>

        </DockPanel>
    </Window>
```

Los dos primeros botones utilizan la propiedad **`SelectedIndex`** para determinar cuál es la pestaña actual, tras lo cual se resta o suma una unidad al valor, asegurándonos de que no pasamos el límite inferior o superior de elementos disponibles. El tercer botón usa **`SelectedItem`** para obtener una referencia al elemento actual. Como se puede ver, hay que hacer una propiedad de cast (conversión) para poder acceder a la propiedad **`Header`**, dado que **`SelectedItem`** es de tipo objeto por defecto.





5.3. Posicionando el control

Por defecto, las pestañas de un **`TabControl`** se posicionan en la parte superior del control. Sin embargo, usando el atributo **`TabStripPlacement`** podemos cambiar este comportamiento fácilmente. La propiedad puede tomar los valores **`Top`** (por defecto), **`Bottom`**, **`Left`** o **`Right`**:

```
<TabControl TabStripPlacement="Bottom" Name="tcEjemplo">
```