ACTIVIDAD GUIADA 4
Controles básicos

Contenidos

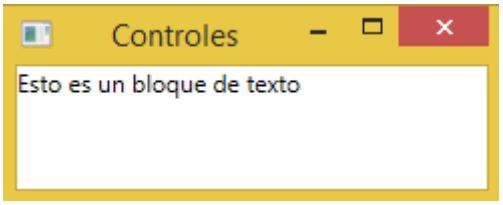
Guarda los ejercicios de esta actividad dentro de un proyecto llamado ***Controles****, a su vez dentro de una solución denominada* ***Ejemplos WPF***

Bloques de texto

NOTA: El bloque de texto no es estrictamente un control, pues no hereda de la clase Control, sin embargo se utiliza como cualquier otro control de WPF, de modo que lo trataremos como tal.

Los bloques de texto (**`TextBlock`**) permiten mostrar texto en la pantalla, de modo similar a una etiqueta (**`Label`**), pero de modo más simple y consumiendo menos recursos. Por otra parte, mientras que las etiquetas habitualmente se utilizan para textos cortos y pueden mostrar imágenes, los bloques de texto funcionan bien para bloques de varias líneas, aunque sólo pueden contener textos (**`string`**).

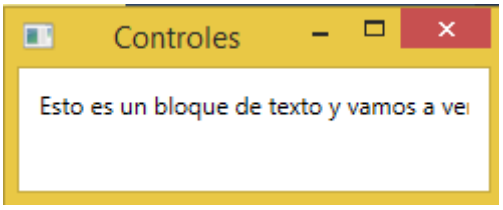
```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="100" Width="250">
    <Grid>
        <TextBlock>Esto es un bloque de texto</TextBlock>
    </Grid>
</Window>
```



El texto entre las etiquetas **`TexBlock`** es simplemente un atajo para establecer la propiedad **`Text`** del **`TextBlock`**.

En el siguiente ejemplo, vamos a mostrar un texto más largo para comprobar como lo trata el **`TextBlock`**. También hemos añadido un pequeño margen para mejorar la presentación:

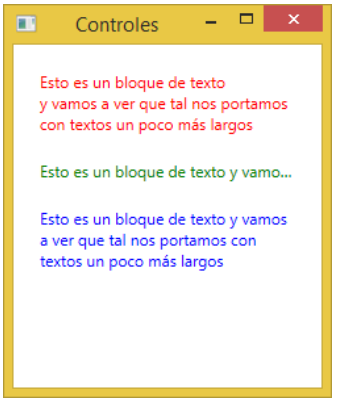
```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="100" Width="250">
    <Grid>
        <TextBlock Margin="10">Esto es un bloque de texto y vamos a ver cómo
nos portamos con textos un poco más largos</TextBlock>
    </Grid>
</Window>
```



Como se puede ver en la captura, el componente es capaz de tratar textos largos, pero por defecto no hace nada para mostrarlo dentro de la pantalla. Hay varios modos de tratar esto. Lo veremos mediante otro ejemplo.

Definimos tres controles **TextBlock**, cada uno con un color distinto (usando la propiedad **Foreground**) para visualizar las diferencias claramente.

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="300" Width="250">
    <StackPanel Margin="10">

        <TextBlock Foreground="Red" Margin="10">Esto es un bloque de texto
                   <LineBreak /> y vamos a ver que tal nos portamos
                   <LineBreak />con textos un poco más largos
        </TextBlock>

        <TextBlock Foreground="Green" Margin="10"
                   TextTrimming="CharacterEllipsis">
                   Esto es un bloque de texto  y vamos a ver que tal nos
                   portamos con textos un poco más largos
        </TextBlock>

        <TextBlock Foreground="Blue" Margin="10" TextWrapping="Wrap">
                    Esto es un bloque de texto  y vamos a ver que tal nos
                    portamos con textos un poco más largos
        </TextBlock>
    </StackPanel>
</Window>
```



Manejan el texto largo de distintos modos:

- El bloque rojo usa la etiqueta **LineBreak** para insertar un salto de línea manualmente en una posición determinada. Esto nos da un control absoluto sobre donde queremos colocar el salto de línea, pero no es demasiado flexible (si el usuario incrementa el tamaño de la pantalla el salto sigue en la misma posición).
- El bloque verde usa la propiedad **TextTrimming** con el valor **CharacterEllipsis** para hacer que el bloque de texto muestre puntos suspensivos si no puede dimensionar el texto en el control. Es un método común si queremos mostrar que hay más texto pero no hay espacio para mostrarlo, si por ejemplo no queremos usar más de una línea. Una alternativa a **CharacterEllipsis** es **WordEllipsis**, que muestra el texto hasta la última palabra, evitando que ésta quede cortada.
- El bloque azul utiliza la propiedad **TextWrapping** con el valor **Wrap**, para hacer que el bloque de texto salte a la siguiente línea cuando no podemos ajustarlo a la línea anterior. El ajuste se realiza de manera automática y se adapta al redimensionamiento de la ventana, luego generalmente es un enfoque mejor que el usado en el primer bloque.

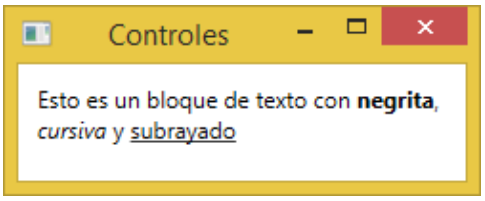
Definiendo el aspecto

Los bloques de texto soportan contenido en línea. Se trata de construcciones que heredan de la clase **Inline**, lo que significa que pueden ser mostrados en línea, como parte de un texto mayor. Veamos algunos de los principales:

- **Negrita, cursiva y subrayado**

Son los tipos más simples de elementos en línea. Veámoslos en un ejemplo:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="100" Width="250">
    <StackPanel >
        <TextBlock Margin="10" TextWrapping="Wrap">
            Esto es un bloque de texto con <Bold>negrita</Bold>,
            <Italic>cursiva</Italic> y <Underline>subrayado</Underline>
        </TextBlock>
    </StackPanel>
</Window>
```



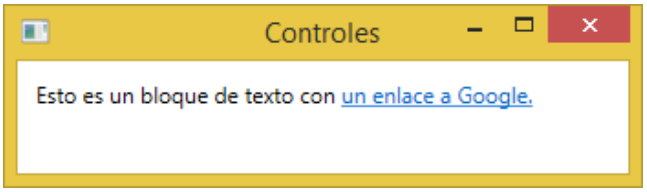
De forma muy parecida a lo que se hace con HTML, al rodear un texto con una etiqueta **Bold**, el texto se pone en negrita. Lo mismo ocurre con **Italic** y **Underline**.

Las tres etiquetas son clases hijas del elemento **Span**, cada uno estableciendo una propiedad específica de **Span** para crear el efecto deseado.

- **Hipervínculos**

El elemento **Hyperlink** permite añadir enlaces al texto. Se muestra con un estilo que se corresponde con el tema de Windows que tenemos instalado, si bien habitualmente se trata de texto subrayado en color azul con un efecto rojo al pasar el ratón por encima. Podemos usar la propiedad **NavigateUri** para definir la URL la cual queremos ir al pulsar en el enlace:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="100" Width="350">
    <StackPanel >
        <TextBlock  Margin="10" TextWrapping="Wrap">
            Esto es un bloque de texto con
           <Hyperlink NavigateUri="http://www.google.es"
                      RequestNavigate="Hyperlink_RequestNavigate">
             un enlace a Google.
           </Hyperlink>
        </TextBlock>
    </StackPanel>
</Window>
```



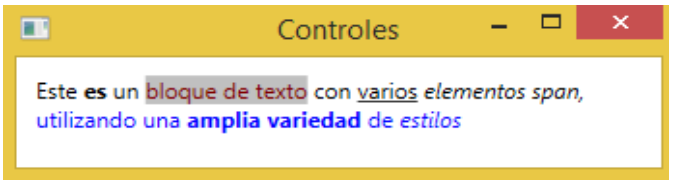
Por sí mismo el enlace no hace nada, por eso tenemos que tratar el evento **RequestNavigate**. En este caso vamos a utilizarlo para lanzar el navegador por defecto desde el archivo de *Code-Behind.*

```
private void Hyperlink_RequestNavigate(object sender, RequestNavigateEventArgs e)
        {
            System.Diagnostics.Process.Start(e.Uri.AbsoluteUri);
        }
```

- **Span**

El elemento **Span** nos permite establecer cualquier tipo de presentación, incluyendo tamaño de letra, estilo y anchura, colores de fondo y de letra, etc. Lo mejor del elemento **Span** es que permite otros elementos *inline* dentro de él, haciendo fácil realizar combinaciones avanzadas de texto y estilo. En el ejemplo siguiente se utilizan varios elementos **Span** para mostrar algunas de sus posibilidades:

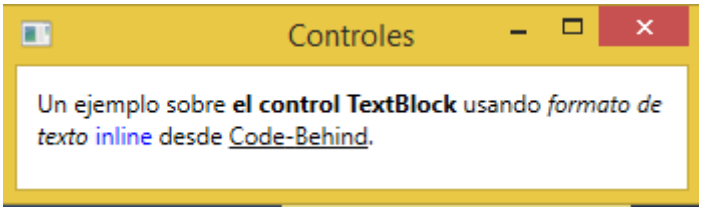
```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="100" Width="350">
    <StackPanel >
        <TextBlock Margin="10" TextWrapping="Wrap">
            Este <Span FontWeight="Bold">es</Span> un
                 <Span Background="Silver" Foreground="Maroon">bloque de texto</Span>
                 con <Span TextDecorations="Underline">varios</Span>
                <Span FontStyle="Italic">elementos span,</Span>
                <Span Foreground="Blue">utilizando una <Bold>amplia variedad</Bold>
                  de <Italic>estilos</Italic>
                </Span>
        </TextBlock>
    </StackPanel>
</Window>
```



Formateando el texto desde C#

Aunque el formateado de texto con XAML es muy sencillo, en ocasiones puede resultar preferible hacerlo desde C#/*Code Behind*. En lugar de utilizar elementos **Span**, se usan elementos **Run**, que son completamente equivalentes a los **Span** con la diferencia de que un **Run** únicamente puede contener texto plano, mientras que un elemento **Span** puede contener otros elementos *inline*. En cualquier caso, por lo general es preferible formatear desde XAML si se puede:

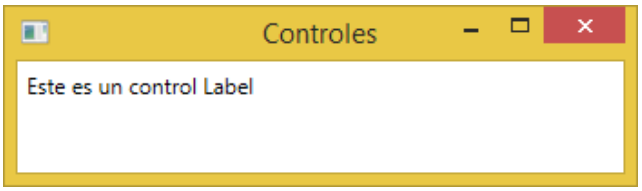
```
namespace Controles
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            TextBlock tb = new TextBlock();
            tb.TextWrapping = TextWrapping.Wrap;
            tb.Margin = new Thickness(10);
            tb.Inlines.Add("Un ejemplo sobre ");
            tb.Inlines.Add(new Run("el control TextBlock ") {
               FontWeight = FontWeights.Bold });
            tb.Inlines.Add("usando ");
            tb.Inlines.Add(new Run("formato de texto ") {
               FontStyle = FontStyles.Italic });
            tb.Inlines.Add(new Run("inline ") {
               Foreground = Brushes.Blue });
            tb.Inlines.Add("desde ");
            tb.Inlines.Add(new Run("Code-Behind") {
               TextDecorations = TextDecorations.Underline });
            tb.Inlines.Add(".");
            this.Content = tb;

        }
    }
}
```



Etiquetas

El control **`Label`**, en su forma más simple, es muy parecido al control **`TextBox`**. En lugar de una propiedad **`Text`**, el **`Label`** tiene una propiedad **`Content`**. La razón del cambio es que el **`Label`** puede albergar cualquier tipo de contenido, en lugar de sólo texto.

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="100" Width="350">
    <Grid>
        <Label Content="Este es un control Label" />
    </Grid>
</Window>
```



Un aspecto a tener en cuenta es que la etiqueta, por defecto, tiene algo de margen, lo que establece una pequeña separación del resto de elementos. Este no es el caso del bloque de texto, donde tenemos que especificarlo de manera manual.

Sin embargo ésta no parece suficiente razón para tener componentes diferenciados. Habitualmente usaremos una etiqueta si estamos en uno de los siguientes casos:

- Queremos especificar un borde.
- Queremos mostrar otros controles, como por ejemplo una imagen.
- **Queremos usar teclas mnemónicos (*Access Keys*) para resaltar controles relacionados.**

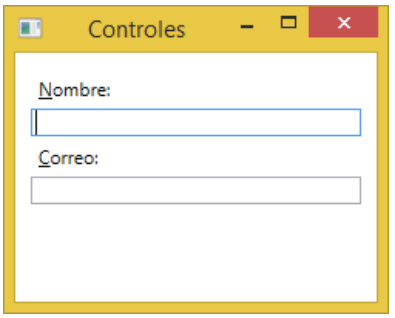
El último punto es una de las razones principales para decantarnos por este control.

Por el contrario, si únicamente queremos mostrar texto plano, usamos el control **`TextBlock`**, pues es más ligero y se comporta mejor que **`Label`** en la mayoría de los casos. El **`Label`** suele usarse, como su nombre indica, para actuar como etiqueta de otros controles.

Uso de mnemónicos

En la mayor parte de S.O. es una práctica común que podamos acceder a los controles de un diálogo pulsando una tecla determinada ([Alt en Windows] y presionando un carácter que se corresponde con el control al que queremos acceder. Los bloques de texto no permiten esta funcionalidad, pero las etiquetas sí. Veamos un ejemplo en funcionamiento:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="200" Width="250">
    <StackPanel Margin="10">
       <Label Content="_Nombre:" Target="{Binding ElementName=txtNombre}" />
       <TextBox Name="txtNombre" />
       <Label Content="_Correo:" Target="{Binding ElementName=txtCorreo}" />
       <TextBox Name="txtCorreo" />
    </StackPanel>
</Window>
```



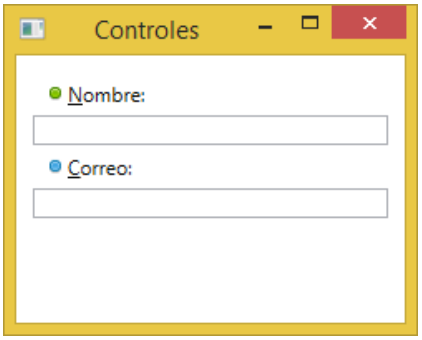
La captura de pantalla muestra el aspecto del diálogo de ejemplo cuando la tecla Alt está presionada. Al ejecutarlo, si pulsamos Alt+N o Alt+C, veremos que el foco se mueve entre los distintos controles. Hay varios conceptos nuevos:

- Por una parte, **definimos las teclas de acceso anteponiendo un carácter de subrayado** (_) antes del carácter en concreto. No tiene por qué ser el primer carácter, puede colocarse previamente a cualquiera de los caracteres de la etiqueta. La práctica común es usar el primer carácter que no se está usando en una tecla de acceso para otro control.
- Usamos la propiedad **Target** para conectar la etiqueta y el control correspondiente. Para ello, usamos un enlace (*binding*) WPF estándar, usando la propiedad **`ElementName`**.

Uso de controles como contenido de etiquetas

Como ya se comentó, el control **`Label`** permite albergar otros controles en lugar de solo texto. Vamos un ejemplo donde albergamos tanto una imagen como un texto dentro de la etiqueta, a la vez que mantenemos una tecla de acceso para cada etiqueta.

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="200" Width="250">
    <StackPanel Margin="10">
        <Label Target="{Binding ElementName=txtNombre}">
            <StackPanel Orientation="Horizontal">
                <Image
Source="http://cdn1.iconfinder.com/data/icons/fatcow/16/bullet_green.png" />
                <AccessText Text="_Nombre:" />
            </StackPanel>
        </Label>
        <TextBox Name="txtNombre" />

        <Label Target="{Binding ElementName=txtCorreo}">
            <StackPanel Orientation="Horizontal">
                <Image
Source="http://cdn1.iconfinder.com/data/icons/fatcow/16/bullet_blue.png" />
                <AccessText Text="_Correo:" />
            </StackPanel>
        </Label>
        <TextBox Name="txtMail" />
    </StackPanel>
</Window>
```



Se trata únicamente de una versión extendida del ejemplo anterior, donde en lugar de mostrar un único **`String`**, la etiqueta alberga tanto una imagen como una pieza de texto:

- Para albergar el texto usamos el control **AccessText**, que nos permite usar la tecla de acceso para la etiqueta.
- Ambos controles se encuentran dentro de un contenedor **`StackPanel`** horizontal, dado que la etiqueta, como cualquier otro derivado de **`ContentControl`**, únicamente permite albergar **un hijo**.

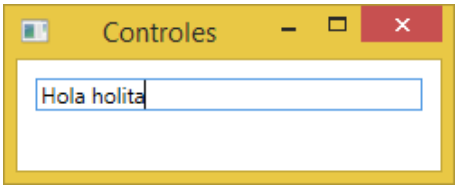
Para este ejemplo hemos usado una imagen remota, si bien esto es algo que no suele hacerse en aplicaciones reales para no depender de la conexión a Internet a la hora de mostrar correctamente la aplicación.

Cajas de texto

El control **TextBox** es el modo más básico de entrada de texto en WPF, permitiendo al usuario escribir texto plano, ya sea para una única línea (un diálogo) o varias líneas (un editor).

Si no establecemos ninguna propiedad en el componente, obtenemos una caja de texto completamente editable que ocupa todo el ancho de la ventana:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="100" Width="250">
    <StackPanel Margin="10">
        <TextBox />
    </StackPanel>
</Window>
```



En el ejemplo hemos añadido el texto una vez que la aplicación ya estaba corriendo, pero podríamos haber añadido un texto por defecto mediante marcado para tener un texto prefijado:

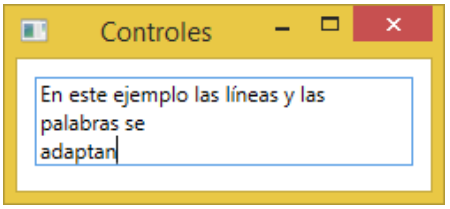
```
<TextBox Text="Hola holita" />
```

Si pulsamos con el botón derecho sobre el **TextBox** veremos que se muestra un menú contextual con acciones que nos permiten interactuar con el Portapapeles de Windows. Además, también añade atajos para deshacer (Control+Z) y rehacer (Control+Y).

Cajas de texto de varias líneas

En el ejemplo anterior podemos ver que el **TextBox** por defecto es de una sola línea. No pasa nada si pulsamos *Enter*, y si añadimos más texto del que cabe en una única línea, el componente únicamente hace *scroll* hacia la derecha. Sin embargo, es fácil convertir el **TextBox** en un editor multilínea:

```
<TextBox AcceptsReturn="True" TextWrapping="Wrap" />
```



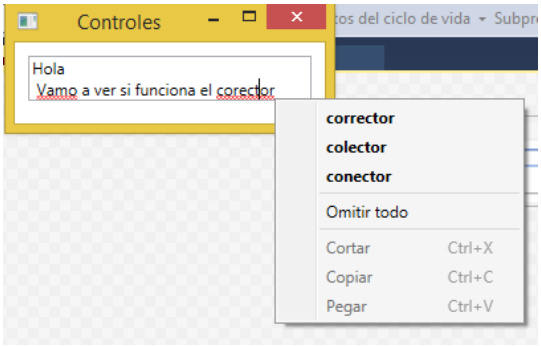
Añadimos un par de propiedades:

- **AcceptsReturn** convierte el **TextBox** en un control multilínea permitiendo el uso de la tecla Enter/Return para ir a la siguiente línea.
- **TextWrapping** permite que el texto se ajuste automáticamente a las líneas inferiores cuando se alcanza el fin de línea.

Control ortográfico

El **TextBox** permite añadir un corrector ortográfico para los lenguajes inglés, francés, alemán y español. Funciona de manera similar al Word, donde los errores son subrayados y podemos pulsar con el botón derecho sobre ellos para sugerir alternativas. Es muy fácil configurar el control ortográfico. Simplemente añadimos una propiedad de la clase **SpellCheck** llamada **IsEnabled**, y la propiedad **Language**.

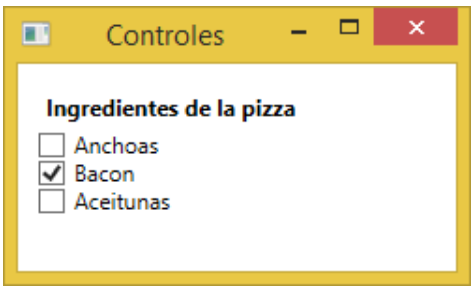
```
<TextBox AcceptsReturn="True" TextWrapping="Wrap"
SpellCheck.IsEnabled="True" Language="es-ES" />
```



Casillas de verificación

Las casillas de verificación **CheckBox** permiten al usuario seleccionar una opción, lo que normalmente se traduce en un valor booleano en el *Code-Behind*. Veamos como siempre en primer lugar un ejemplo sencillo:

```
<Window x:Class="Controles.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="150" Width="250">
    <StackPanel Margin="10">
        <Label FontWeight="Bold">Ingredientes de la pizza</Label>

        <CheckBox>Anchoas</CheckBox>
        <CheckBox IsChecked="True">Bacon</CheckBox>
        <CheckBox>Aceitunas</CheckBox>

    </StackPanel>
</Window>
```

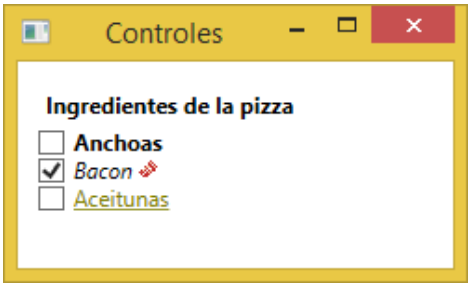


En el segundo **CheckBox**, se usa la propiedad **IsChecked** para dejarlo seleccionado por defecto, pero aparte de este atributo y el nombre por lo general no necesitamos usar más. La propiedad **IsChecked** también se puede usar desde el *Code-Behind* para comprobar si determinada casilla está marcada o no.

Contenido a medida

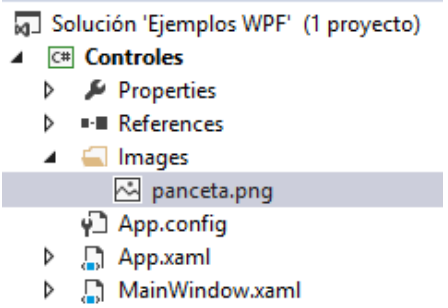
El **CheckBox** hereda de la clase **ContentControl**, lo que implica que puede tomar contenido a medida y mostrarlo al lado de la casilla. Si únicamente especificamos un fragmento de texto, como en el ejemplo, WPF lo coloca en un **TextBlock** y lo muestra, pero podemos usar cualquier tipo de control en la casilla como muestra el siguiente ejemplo:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="150" Width="250">
    <Window.Resources>
        <BitmapImage x:Key="imagenBacon" UriSource="/Images/panceta.png" />
    </Window.Resources>
    <StackPanel Margin="10">
        <Label FontWeight="Bold">Ingredientes de la pizza</Label>
        <CheckBox>
            <TextBlock>
               <Span FontWeight="Bold">Anchoas</Span>
            </TextBlock>
        </CheckBox>
        <CheckBox IsChecked="True">
            <TextBlock>
               <Span FontStyle="Italic">Bacon</Span>
               <Image Source="{StaticResource imagenBacon}" Width="10" Height="10"/>
            </TextBlock>
        </CheckBox>
        <CheckBox>
            <TextBlock>
               <Span Foreground="Olive" TextDecorations="Underline">Aceitunas</Span>
            </TextBlock>
        </CheckBox>
    </StackPanel>
</Window>
```



Como se puede ver, en cada caja de texto realizamos una opción distinta con el texto. En la casilla correspondiente al ingrediente *Bacon* mostramos una imagen. Independientemente de que pulsemos en el texto o en la imagen correspondiente, el control se activará/desactivará a medida que pulsamos sobre él.

Para añadir la imagen, hemos añadido un recurso estático de tipo **BitMapImage** para el cual hemos definido la propiedad **UriSource** con la ruta de la imagen (que se ha añadido en la subcarpeta **Images**). La ruta comienza en /, que representa la raíz del proyecto:

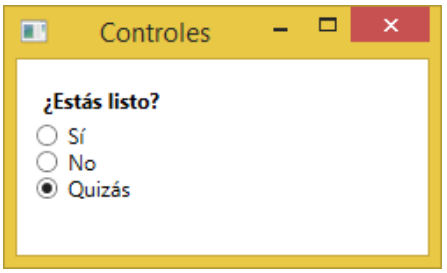


Botones de radio

Los botones de radio (**RadioButton**) proporcionan al usuario una lista de posibles opciones, donde únicamente podemos escoger una a la vez.

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="150" Width="250">

    <StackPanel Margin="10">
        <Label FontWeight="Bold">¿Estás listo?</Label>
        <RadioButton>Sí</RadioButton>
        <RadioButton>No</RadioButton>
        <RadioButton IsChecked="True">Quizás</RadioButton>
    </StackPanel>
</Window>
```

Únicamente añadimos un **Label** con una pregunta, y tres **RadioButton**, cada uno con una posible respuesta. Se define una respuesta por defecto usando la propiedad **IsChecked** en el último botón, que el usuario puede cambiar simplemente pulsando en el resto de botones. **Esta propiedad es la que deberíamos usar desde el *Code-Behind* para comprobar si el botón está seleccionado o no.**

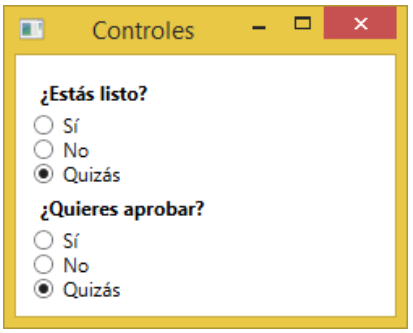


Grupos de botones

En el ejemplo anterior se comprueba que sólo se puede pulsar un botón al mismo tiempo. Sin embargo, puede que queramos tener varios grupos de botones, cada uno con su selección individual. Para ello usamos la propiedad **GroupName**, que permite especificar qué botones se agrupan conjuntamente:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="200" Width="250">

    <StackPanel Margin="10">
        <Label FontWeight="Bold">¿Estás listo?</Label>
        <RadioButton GroupName="listo">Sí</RadioButton>
        <RadioButton GroupName="listo">No</RadioButton>
        <RadioButton IsChecked="True" GroupName="listo">Quizás</RadioButton>

        <Label FontWeight="Bold">¿Quieres aprobar?</Label>
        <RadioButton GroupName="aprobar">Sí</RadioButton>
        <RadioButton GroupName="aprobar">No</RadioButton>
        <RadioButton IsChecked="True" GroupName="aprobar">
         Quizás</RadioButton>
    </StackPanel>
</Window>
```

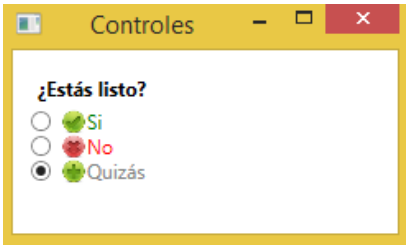
En este caso cada una de las opciones se puede elegir por separado. De lo contrario, sólo podríamos escoger una de las seis:



Contenido a medida

Los **`RadioButton`** heredan de la clase **`ContentControl`**, lo que significa que pueden tomar contenido a medida al igual que hicimos en el ejemplo con las casillas de verificación. Ejemplificamos este comportamiento en el siguiente ejemplo:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="150" Width="250">
    <Window.Resources>
        <BitmapImage x:Key="imgSi" UriSource="/Images/si.png" />
        <BitmapImage x:Key="imgNo" UriSource="/Images/no.png" />
        <BitmapImage x:Key="imgQuizas" UriSource="/Images/quizas.png" />
    </Window.Resources>

    <StackPanel Margin="10">
        <Label FontWeight="Bold">¿Estás listo?</Label>
        <RadioButton GroupName="listo">
            <WrapPanel>
                <Image Source="{StaticResource imgSi}" Width="16" />
                <TextBlock Foreground="Green">Si</TextBlock>
            </WrapPanel>
        </RadioButton>
        <RadioButton GroupName="listo">
            <WrapPanel>
                <Image Source="{StaticResource imgNo}" Width="16" />
                <TextBlock Foreground="Red">No</TextBlock>
            </WrapPanel>
        </RadioButton>
        <RadioButton IsChecked="True" GroupName="listo">
            <WrapPanel>
                <Image Source="{StaticResource imgQuizas}" Width="16" />
                <TextBlock Foreground="Gray">Quizás</TextBlock>
            </WrapPanel>
        </RadioButton>
    </StackPanel>
</Window>
```



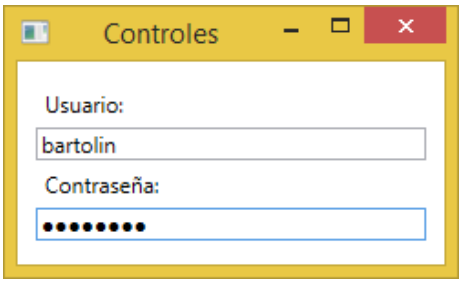
Contraseñas

Para editar contraseñas podemos utilizar un control específico llamado **PasswordBox** cuya funcionalidad es casi la misma que en un **TextBox**, pero se muestra algo distinto a los caracteres reales al introducir la contraseña, lo que evita que miradas no deseadas la vean.

Su utilización es tan sencilla como la de los **TextBox**, tal y como muestra el siguiente ejemplo:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="150" Width="250">
    <StackPanel Margin="10">
        <Label>Usuario:</Label>
        <TextBox />
        <Label>Contraseña:</Label>
        <PasswordBox />
    </StackPanel>
</Window>
```

El siguiente ejemplo hace uso del campo de contraseña. Hemos introducido el mismo texto en ambas cajas, pero sólo se visualiza su contenido en la caja superior:



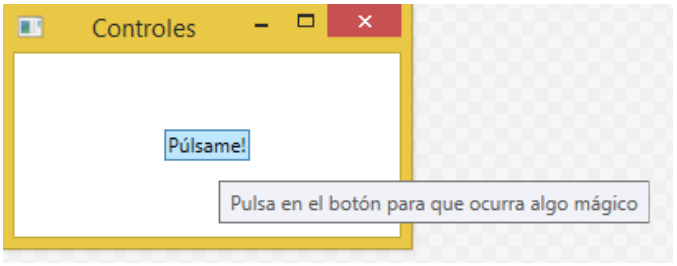
Podemos controlar qué carácter se utiliza para ocultar la contraseña con la propiedad **PasswrdChar**. También podemos controlar la longitud máxima del campo con la propiedad **MaxLength**, como muestra el siguiente ejemplo:

```
<PasswordBox PasswordChar="X" MaxLength="6"/>
```

Tooltips

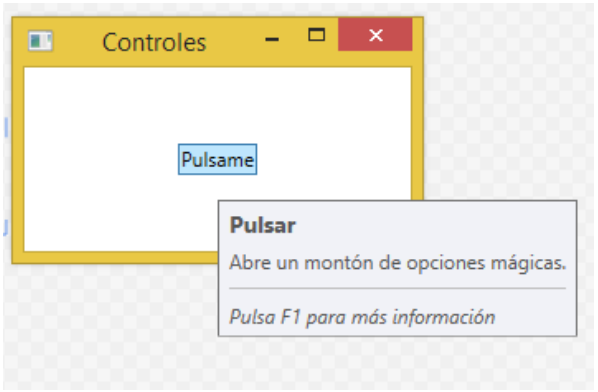
Los *tooltips* nos permiten dar información extra sobre un control específico o un enlace poniendo el ratón sobre el mismo. WPF soporta este concepto utilizando la propiedad **ToolTip** que se encuentra en la clase **FrameworkElement**, de la cual la mayor parte de los controles WPF heredan. Especificar un *tooltip* es muy fácil, como muestra el siguiente ejemplo:

```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="150" Width="250">
    <Grid VerticalAlignment="Center" HorizontalAlignment="Center">
        <Button ToolTip="Pulsa en el botón para que ocurra algo mágico">
         Púlsame!</Button>
    </Grid>
</Window>
```



WPF permite un comportamiento más avanzado para los *tooltips*, dado que la propiedad en realidad no es un **`String`**, sino un objeto, de modo que podemos colocar en él lo que queramos.

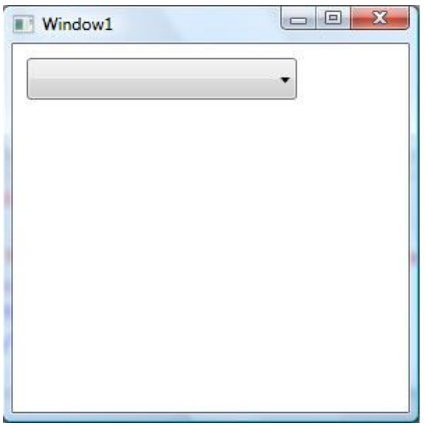
```
<Window x:Class="Controles.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Controles" Height="150" Width="250">
    <Grid VerticalAlignment="Center" HorizontalAlignment="Center">
        <Button>
            <Button.Content>
                <TextBlock>Pulsame</TextBlock>
            </Button.Content>
            <Button.ToolTip>
                <StackPanel>
                    <TextBlock FontWeight="Bold" FontSize="14" Margin="0,0,0,5">
                      Pulsar
                    </TextBlock>
                    <TextBlock>
                        Abre un montón de opciones mágicas.
                    </TextBlock>
                    <Border BorderBrush="Silver" BorderThickness="0,1,0,0"
                            Margin="0,8" />
                    <WrapPanel>
                        <TextBlock FontStyle="Italic">
                             Pulsa F1 para más información
                        </TextBlock>
                    </WrapPanel>
                </StackPanel>
            </Button.ToolTip>
        </Button>
    </Grid>
</Window>
```



ComboBox

Para representar un ComboBox con XAML tenemos la etiqueta <ComboBox>. El siguiente código fija además su nombre, anchura, altura y alineación:

```
<ComboBox Name="ComboBox1" Width="200" Height="30"
          VerticalAlignment="Top" HorizontalAlignment="Left">
</ComboBox>
```



Para añadirle componentes vía XML, utilizamos *ComboBoxItem*. Este elemento tiene una propiedad *isSelected* que indicará el elemento por defecto.

```
<ComboBoxItem Content="Coffie" IsSelected="True" />
```

También es posible añadir elementos por código, a través del método *Add*:

```
ComboBox1.Items.Add("Tea");
```

Si queremos asignar directamente una colección al combo, modificaremos su propiedad *ItemSource*:

```
String [] lista = {"uno","dos","tres"};
ComboBox1.ItemsSource = lista;
```

Y para recuperar el valor seleccionado pueden interesarnos:

- La propiedad *Text* representa el texto del ítem.
- *SelectemItem* representa el primer ítem seleccionado.
- *SelectedValue* representa el valor del ítem seleccionado.
- *SelectedIndex* representa la posición del ítem seleccionado.

El siguiente código prueba las propiedades citadas:

```
string str = ComboBox1.Text;
ComboBoxItem cbi = (ComboBoxItem)ComboBox1.SelectedItem;
string str1 = cbi.Content.ToString();
string val = ComboBox1.SelectedValue.ToString();
```