TEORÍA

WPF - Windows Presentation Foundation

Contenido

Introducción

WPF o “Windows Presentation Foundation” se introdujo con Windows Vista e inicialmente compartía características con SilverLight, por aquel entonces Flash era la referencia. Toma características tanto de las aplicaciones de escritorio como de las aplicaciones web permitiéndole a WPF crear aplicaciones tanto para escritorio como para navegador.

WPF amplía las posibilidades de WinForms, ahora podemos realizar aplicaciones multimedia como reproductores de audio o de video, aplicaciones que sintetizan voz o procesan comandos verbales, documentos enriquecidos al estilo de word, animaciones, gráficos en 2D y en 3D.

Una mejora respecto a WinForms es que ahora la representación gráfica de la aplicación la realiza la GPU a través de Direct3D, con esto descargamos a la CPU de carga de trabajo y a la vez tenemos aplicaciones visualmente más atractivas. Los gráficos ahora son vectoriales y pueden escalar sin perdida pudiendo adaptándose la aplicación a distintas resoluciones sin necesidad de ajustes.

Como punto negativo descubrirás que WPF ya no soporta aplicaciones MDI, aunque con la combinación de un dock panel y controles de usuario simulando ventanas se pueden desarrollar aplicaciones similares.

En cuanto a la forma de codificar WPF separa el código del Frontend, ficheros XAML, del Backend, ficheros de código .NET. Ambos ficheros están interrelacionados pudiendo manipular mediante código cualquier característica de la presentación. Este modelo de dos vistas favorece un modelo desarrollo basado en el patrón MVC donde la vista es en fichero XAML y el controlador el fichero de respaldo .NET.

Una parte importante de WPF es el lenguaje XAML, acrónimo de “eXtensible Application Markup Language” o lenguaje extensible de marcado de aplicación. Este lenguaje se basa en XML y es de tipo declarativo, es decir, decimos que queremos representar y no como representarlo.

El XAML copia características del HTML, verás que algunas propiedades tienen los mismos nombres y formas de asignar valores. Las etiquetas tienen etiquetas de apertura y cierre, y algunos elementos se anidan dentro de otros de forma jerarquicé para definir la vista. Es importante que recuerdes que en **XAML se diferencian MAYUSCULAS de minúsculas**.

Aunque nosotros trabajemos con ficheros XAML al compilar el proyecto se generan fichero binarios “.baml” que se incrustan en el ensamblado. También se incrustan el resto de recursos como los iconos o imágenes que usemos.

Estructura etiquetas AXML

Vamos a ver como trabajar con las etiquetas o controles XAML. La siguiente etiqueta define un botón sin ninguna propiedad por lo que su tamaño lo definirá el contenedor.

```
<Button />
```

Es equivalente a

```
<Button></Button>
```

Ahora vamos a darle un contenido, en este caso el texto que se muestra en el botón.

```
<Button>Texto del botón</Button>
```

Es equivalente a (más la fuente en negrita)

```
<Button FontWeight=”Bold” Content=”Texto del botón” />
```

También se podría escribir así, fíjate en que podemos sacar las propiedades a una etiqueta hija.

```
<Button>
      <Button.FontWeight>Bold</Button.FontWeight>
      <Button.Content>Texto del botón</Button.Content>
</Button>
```

Podemos cambiar el Content por cualquier elemento que queramos para dar formato a la presentación. WrapPanel apila en vertical el contenido.

```
<Button>
    <Button.FontWeight>Bold</Button.FontWeight>
    <Button.Content>
        <WrapPanel>
            <TextBlock Foreground="Blue">Multi</TextBlock>
            <TextBlock Foreground="Red">Color</TextBlock>
            <TextBlock>Botón</TextBlock>
        </WrapPanel>
    </Button.Content>
</Button>
```

IMPORTANTE: la propiedad Content solamente puede contener un nodo hijo. En los casos con más de un elemento lo debemos envolver con el panel “WrapPanel”. ESTO PASA CON TODOS LOS ELEMENTOS QUE DERIVAN DE LA CLASE “**ContentControl**”.

Una forma simplificada de escribir lo mismo sería

```
<Button FontWeight="Bold">
    <WrapPanel>
        <TextBlock Foreground="Blue">Multi</TextBlock>
        <TextBlock Foreground="Red">Color</TextBlock>
        <TextBlock>Botón</TextBlock>
    </WrapPanel>
</Button>
```

Si quisiéramos podríamos hacer lo mismo por código.

```
Button btn = new Button();
btn.FontWeight = FontWeights.Bold;

WrapPanel pnl = new WrapPanel();

TextBlock txt = new TextBlock();
txt.Text = "Multi";
txt.Foreground = Brushes.Blue;
pnl.Children.Add(txt);

txt = new TextBlock();
txt.Text = "Color";
txt.Foreground = Brushes.Red;
```

```
pnl.Children.Add(txt);

txt = new TextBlock();
txt.Text = "Botón";
pnl.Children.Add(txt);

btn.Content = pnl;
pnlMain.Children.Add(btn);
```

Eventos

Podemos asignar eventos de 3 maneras. Desde el editor de propiedades, desde el editor AXML escribiendo el nombre del evento y dejando que nos ayude intelisense pulsando TAB, y por último desde el código con ".Evento += Manejador"

Descripción general

Ventana

El control Window es el equivalente a Form en WinForms, esté control deriva de ContentControl por lo que únicamente tendrá un control hijo que será alguno de los tipos de paneles disponibles.

Una ventana consta de un fichero XAML y un fichero de código asociado. La relación entre ambos es a través del atributo "**x:Class**". Por defecto el panel hijo es de tipo "Grid", del que se podría decir que es un contenedor neutro que puede albergar tantos hijos como queramos.

```
<Window x:Class="Demo1.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:Demo1"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <TextBlock HorizontalAlignment="Center" TextWrapping="Wrap"
VerticalAlignment="Center" Height="419" Width="792" FontSize="72">
                  <Run Text="Hola Mundo!"/>
                        <LineBreak/>
                  <Run/>
        </TextBlock>

    </Grid>
</Window>
```

En el fichero de código de respaldo cuenta con un constructor que invoca al método "InitializeComponents", su codificación es distinta a la de WinForms. Échale un ojo a la implementación y ubicación de esta función (Si no lo encuentras, compila el proyecto).

```
namespace Demo1
{
    /// <summary>
    /// Lógica de interacción para MainWindow.xaml
    /// </summary>
```

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

Propiedades de Window

- ResizeMode: ajusta la forma en la que se redimensiona la ventana.
  - NoResize
  - CanResize
  - CanResizeWithGrip -> Igual que CanResize pero con un control abajo a la derecha para redimensionar
  - CanMinimize
- SizeToContent: ajusta el tamaño de la ventana al tamaño del contenido.
  - Manual (defecto)
  - Width
  - Height
  - WidthAndHeight

App.xaml

Este fichero es el punto de entrada de la aplicación, tanto para los elementos XAML como para el código fuente.

Si quisiéramos podríamos cambiar la ventana inicial por la llamada a la función manejadora, para ello cambiamos la propiedad **StartupUri="MainWindows.xaml"** por el evento **Startup="Application_Startup"** y definir el código en la clase de apoyo App.xaml.cs con la firma **"private void Application_Startup(object sender, StartupEventArgs e)"**.

Está funcionalidad puede ser útil para manejar los parámetros de la aplicación, por ejemplo, imagínate aplicaciones al estilo visores de fotografías o editores que reciben al hacer doble click abren el documento indicado.

```
public partial class App : Application
{
      private void Application_Startup(object sender, StartupEventArgs
e)
      {
            MainWindow wnd = new MainWindow();
            if(e.Args.Length == 1)
                  MessageBox.Show("Abriendo: " + e.Args[0]);
            wnd.Show();
      }
}
```

NOTA: para configurar un parámetro de ejecución debemos ir a propiedades del proyecto, opción de depuración y establecer un valor en la propiedad de "Argumentos de la línea de comandos".

Recursos

Los recursos permiten almacenar dentro de las definiciones XAML de las vistas o de la aplicación diversos tipos de elementos. Algunos tipos son: datos, bloques de código xaml reutilizables, referencias a comandos y referencias a clases de apoyo estáticas. Por ejemplo, si reutilizáramos un menú contextual en una ventana con varias listas podríamos definir el código xaml del menú como un recurso y luego referenciarlo en cada lista.

La gracia, y uso común de los recursos, es combinarlo con la definición de estilos. De esta forma podemos definir en un único lugar los estilos de nuestra aplicación y aplicarlos sin tener que editar los controles uno a uno. Esta aproximación no permite crear una especie de hoja de estilos o CSS al estilo de una página web.

Si los estilos se definen en el elemento raíz, en la aplicación, estarán disponibles en toda la aplicación. En cambio, si los definimos por ejemplo en una ventana específica únicamente estarán disponibles en dicha ventana.

Ámbitos de estilo

- Globales, en App.xaml, se verán en toda la aplicación de forma global. En este caso por rendimiento se recomienda envolver con la etiqueta “ResourceDictionay”
- Locales, en el fichero .xaml de cada ventana.
- Dentro de cada etiqueta podemos rellenar la propiedad Style.
- Dentro de cada etiqueta ajustando propiedades una a una.

Por último, indicar que podemos asignar estilos explícitamente, a través de una clave, e implícitamente, ajustando los valores por defecto de un control especifico.

Ejemplo de estilos globales. El primero con clave y el segundo a todos los objetos de un tipo.

```
<Application.Resources>
      <!--El diccionario hace que se cacheen los recursos, útil para
los estilos globales-->
      <ResourceDictionary>
            <!--Declaramos un estilo con key, hay que asignarlo
explícitamente con la propiedad Style-->
            <Style TargetType="Button" x:Key="BotonRojo">
                  <Setter Property="Background" Value="Red" />
            </Style>
            <!--Declaramos un estilo global, todos los label tendrán el
estilo aplicado implícitamente-->
            <Style TargetType="Label">
                  <Setter Property="FontSize" Value="42"/>
                  <Setter Property="FontWeight" Value="Bold"/>
            </Style>
      </ResourceDictionary>
</Application.Resources>
```

Podemos asignar los recursos, en este caso estilos, en XAML de dos formas distintas:

- **StaticResource**, el estilo carga en el evento Load y no se modifica, opción habitual.
- DynamicResource, el estilo se carga como una expresión, si el recurso cambia los elementos enlazados también cambia.

Ejemplo de asignación en línea dentro del control **Style=”{StaticResource NombreEstilo}”**

```
<Button x:Name="btnAplicar" Content="Aplicar por código"
Style="{StaticResource BotonRojo}" />
```

No olvides que también podemos manipular los estilos desde el código.

Los estilos globales se encuentras en “App.Current.Resources” y los locales en “this.Resources”. Es una colección por lo que puedes leerla como un array, Resources[“NombreEstilo”], el objeto devuelto normalmente será de tipo Style por lo que habrá que castearlo a la propiedad .Style del objeto.

```
//App.Current.Resources
Style estilo = (Style)this.Resources["FondoAzul"];
this.lblNombre.Style = estilo;
```

NOTA: los elementos únicamente tienen un estilo, si queremos aplicar varios debemos combinarlos en un único y nuevo estilo.

Documentación

https://docs.microsoft.com/en-us/dotnet/desktop/wpf/fundamentals/styles-templates-create-apply-style?view=netdesktop-5.0

Gestión de errores

Igual que siempre, mediante los bloques try, catch, finally. Las excepciones no capturadas se pueden recoger en el evento “DispatcherUnhandledException” de la clase App. Como siempre lo definimos en el fichero app.xaml y dejamos que autocompletar cree la firma por nosotros.

Cuando se dispare el evento de excepción no controlada el segundo argumento contendrá en “e.Exception” la excepción no gestionada.

Controles básicos.

TextBlock

Se parece al label pero permite textos de múltiples líneas y formatos. Para textos cortos o si necesitamos incluir imágenes debemos emplear “label”. Este control es importante y se emplea para dar formato a los textos dentro de otros controles.

NOTA: podemos establecer contenido rellenado la propiedad **.Text**, en este caso texto plano, o a través de la propiedad .**Inlines**, en este caso para incluir otras etiquetas.

OJO, no hereda de ContentControl.

Propiedades

- Text, texto mostrado. Si el control tiene hijos con texto formateado esta propiedad no aplica.
- **Inlines**, sólo para código, texto mostrado.
- TextTrimming, recorta el exceso de texto, añade los puntos al final “…”. Valores:
    - CharacterEllipsis, recorta en medio de una palabra si hace falta.
    - WordEllipsis, recorta al final de la última palabra posible.
- TextWrapping, ajusta el texto al espacio disponible. Valores:
    - NoWrap

- Wrap, mete saltos de línea.
- WrapWithOverflow, trata de ajustar pero corta cuando se sobrepasa el tamaño del bloque.

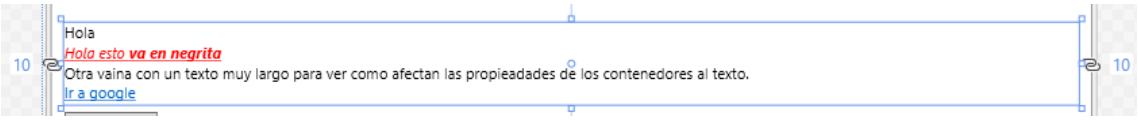
TextBlock. Formato en línea.

El control TextBlock admite elementos en línea, es decir, podemos añadir a su contenido otras etiquetas o controles en línea permitiéndonos personalizar la apariencia del texto. A continuación, la lista de controles en línea admitidos.

- LineBreak, salto de línea.
- Bold, negrita.
- Italic, cursiva.
- Underline, subrayado.
- Run, texto con formato.
- Span, texto y elementos con formato.
- Hyperlink, hipervínculo.
- AnchoredBlock, bloque anclado. Similar a un "DIV" en HTML.
- InlineUIContainer, contenedor de controles, podríamos por ejemplo añadir un combo.

Por ejemplo.

```
<TextBlock x:Name="tbSegundo" Margin="10,82,10,0" Text="TextBlock"
VerticalAlignment="Top" Height="66" TextTrimming="CharacterEllipsis">
      <Run>Hola</Run>
      <LineBreak />
      <Span Foreground="Red" TextDecorations="Underline"
FontStyle="Italic" >Hola esto <Bold>va en negrita</Bold></Span>
      <LineBreak/>
      <Span>Otra vaina con un texto muy largo para ver cómo afectan
las propiedades de los contenedores al texto.</Span>
      <LineBreak />
      <Hyperlink NavigateUri="https://www.google.com"
RequestNavigate="Hyperlink_RequestNavigate">Ir a google</Hyperlink>
</TextBlock>
```



Y en código

```
this.tbCodigo.Inlines.Clear();

Span span = new Span();
span.Inlines.Add(new Run("Esto es un span y no admite texto sin
elemento en línea"));
span.Inlines.Add(new LineBreak());
span.Inlines.Add(new Run("Otra línea"));

span.FontFamily = new FontFamily("Comic Sans MS");
span.Inlines.ElementAt(0).FontWeight = FontWeights.UltraBold;
span.Inlines.ElementAt(2).Background = new
SolidColorBrush(Colors.Yellow);

this.tbCodigo.Inlines.Add(span);
```

Label

La etiqueta permite mostrar contenidos sencillos, dicho esto permite algunas opciones adicionales interesantes.

- Se le puede asociar una tecla de acceso rápido
- Admite bordes
- Puede renderizar otros controles como imágenes
- Uso de plantillas a través de la propiedad .ContentTemplate

NOTA: hereda de ContentControl.

La sintaxis básica.

```
<Label Content="Texto de la etiqueta" />
```

Añadir una imagen a la etiqueta.

Este truco es común. En este caso deberemos añadir un StackPanel y añadir como elementos hijo la imagen y el textblock.

```
<Label HorizontalAlignment="Left">
      <StackPanel Orientation="Horizontal">
            <Image Source="procedure_16xMD.png" Height="26" Width="27"
HorizontalAlignment="Left" />
            <TextBlock Text="Calendario" Width="62" Margin="0,0,0,0"
VerticalAlignment="Center"/>
      </StackPanel>
</Label>
```

TextBox

Muestra una caja en la que se puede editar texto. No hereda de “ContentControl” por lo que trabajaremos con la propiedad “.Text”

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

Propiedades

- Text, texto editable.
- **AcceptsReturn**, multilínea.
- TextWrapping, distribuye la línea en la siguiente línea al alcanzar el borde del control.
- SpellCheck.IsEnabled="True", corrección ortográfica.
- Otras propiedades similares al control de WinForms

```
<TextBox AcceptsReturn="True" TextWrapping="Wrap"
SpellCheck.IsEnabled="True" />
```

Button

También hereda de “ContentControl”, aunque el uso habitual es ponerle un texto y olvidarnos podemos rellenar la propiedad “content” para que se ajuste a nuestras necesidades, por ejemplo, una imagen más texto.

```
<Button x:Name="btnFormato" Margin="10,209,0,176"
HorizontalAlignment="Left" >
      <StackPanel Orientation="Horizontal">
            <Image Source="procedure_16xMD.png" Height="16"/>
            <TextBlock Padding="10,5">Botón formateado</TextBlock>
      </StackPanel>
</Button>
```

NOTA: el botón tiene las propiedades "IsCancel" e "IsDefault" para cuando se pulsan las teclas "Esc" e "Intro". Ya no son propiedades del formulario (en este caso de la ventana).

CheckBox

Control activable y desactivable. Tiene un comportamiento ligeramente distinto al de WinForms.

OJO, el habitual evento CheckedChanged de WinForms ya no está disponible, ahora debemos emplear el evento Click, o Checked y Unchecked.

OJO, con el estado "null" los eventos Checked y Unchecked no saltan.

Fíjate que puede resultar útil envolverlo en un StackPanel evitando tener que ajustar cada control interno en la ventana.

```
<StackPanel Margin="20,5">
      <CheckBox Name="chkOpcion1" Click="chkOpcion_Click">Habilitar
opción 1</CheckBox>
      <CheckBox Name="chkOpcion2" IsChecked="True"
Click="chkOpcion_Click">Habilitar opción 2</CheckBox>
      <CheckBox Name="chkOpcion3" Click="chkOpcion_Click">Habilitar
opción 3</CheckBox>
</StackPanel>
```

RadioButton

Conjunto de controles que permiten seleccionar un único valor. El usuario puede seleccionar un valor pero no deseleccionarlo.

Como siempre, todos los radio buttons que están en la misma ventana la aplicación los ve en el mismo grupo. A diferencia de WinForms aquí podemos emplear la propiedad "**GroupName**" y agrupar varios radio buttons.

Trabajamos con los eventos Checked, Unchecked y Click, el primero salta con los valores por defecto, y el click cada vez que pinchamos en el control, aunque este ya este activo.

Podemos trabajar con el nombre de cada Radio Button, emplear la propiedad Tag o jugárnosla con el Content.

Fíjate que puede resultar útil envolverlo en un StackPanel evitando tener que ajustar cada control interno en la ventana.

```
<StackPanel>
      <Label FontWeight="Bold">Segundo grupo</Label>
      <RadioButton GroupName="grupo2" Click="RadioButton_Click"
Tag="si">Sí</RadioButton>
      <RadioButton GroupName="grupo2" Click="RadioButton_Click"
Tag="no">No</RadioButton>
```

```
      <RadioButton GroupName="grupo2" IsChecked="True"
Click="RadioButton_Click" Tag="quizas">Quizás</RadioButton>
</StackPanel>
```

Y el código

```
private void RadioButton_Click(object sender, RoutedEventArgs e)
{
      string valor = (string)((RadioButton)e.Source).Tag;
      MessageBox.Show("Ha pulsado: " + valor);
}
```

Image

Permite mostrar imágenes, las propiedades interesantes son:

- Source, para indicar la fuente, un Bitmap.
- Stretch (Estirar), para indicar la manera en la que se deforma la imagen para adaptarse al control. None, Fill, Uniform, UniformToFill

Para trabajar con código hay que envolver las rutas con la clase "Uri"

```
OpenFileDialog openFileDialog = new OpenFileDialog();
if (openFileDialog.ShowDialog() == true)
{
      Uri archivoUri = new Uri(openFileDialog.FileName);
      this.iVisor.Source = new BitmapImage(archivoUri);
}
```

En el caso de querer cargar un recurso incrustado en el proyecto deberemos indicar que la Uri tiene una ruta relativa. En este caso el proyecto tiene una carpeta "images" para los recursos incrustados. Automáticamente se incluyen o incrustan en el ejecutable.

```
Uri recursoUri = new Uri("/images/foto2.png", UriKind.Relative);
this.iVisor.Source = new BitmapImage(recursoUri);
```

Conceptos sobre controles

ToolTip

La propiedad tooltip permite mostrar una ayuda contextual sobre el control cada vez que el usuario deja el puntero del ratón inmóvil sobre el control.

Todos los controles que hereden de "FrameworkElement" cuentan con la propiedad ToolTip.

NOTA: la propiedad ToolTip es de tipo object por lo que nada te impide meterla un panel con los contenidos que quieras. <Button.ToolTip> LO QUE SEA </Button.ToolTip>

A través de "ToolTipService" puedes acceder a propiedades de configuración del tooltip, en el ejemplo siguiente se cambia la duración, con que recuerdes que tienes más opciones disponibles bajo esta propiedad es suficiente.

```
<Button ToolTip="Crea un nuevo archivo"
ToolTipService.ShowDuration="5000" Content="Abrir" />
```

Renderizado de textos

Al tener un modo de renderizado distinto de WinForms (GDI) es posible que con textos pequeños las fuentes se vean mal.

En estos casos podemos ajustar el comportamiento del motor de renderizado actuando sobre la propiedad ".TextOptions" que cuenta con dos propiedades hijas:

- .TextOptions.TextFormattingMode, para definir el modo de renderizado (Ideal o Display).
- .TextOptions.TextRenderingMode, para definir el algoritmo de antialising.

Orden de tabulación

Permite indicar el orden del control dentro de la ventana al emplear la tecla TAB para navegar por la ventana.

Es similar a WinForms, en este caso tenemos dos propiedades:

- **TabIndex**, indica el orden de tabulación.
- **IsTabStop**, indica que no se debe parar en el control. Por ejemplo, útil para evitar los campos de solo lectura o deshabilitados.

```
<TextBox TabIndex="5" IsReadOnly="True" IsTabStop="False" />
```

Atajos de teclado

NOTA: Lo indicado en el manual de referencia ya no tira. NO LO HE PROBADO

https://stackoverflow.com/questions/3574405/implement-keyboard-shortcuts

Hay que implementar dos propiedades en la ventana (**CommandBindings** e **InputBindings**) y posteriormente asignarse una opción a un botón.

```
<Window.CommandBindings>
    <CommandBinding Command="Settings" CanExecute="SettingsCanExecute"
Executed="SettingsExecuted" />
</Window.CommandBindings>

<Window.InputBindings>
    <KeyBinding Command="Settings" Key="S" Modifiers="Alt" />
</Window.InputBindings>
```

Y aquí le asignamos a un botón la acción previamente configurada.

```
<Button Height="50" Width="50" Margin="50,5,0,0" Command="Settings" />
```

Otra opción es enchufarle un InputBinding directamente.

```
<Menu>
    <MenuItem Name="New" Header="New Whatever" Click="click event"
Command="a command">
        <MenuItem.InputBindings>
            <KeyBinding Key="N" Modifiers="Control" Command="New"/>
        </MenuItem.InputBindings>
    </MenuItem>
</Menu>
```

Introducción a los paneles WPF

Muchos tipos de paneles, descripción breve:

- **Canvas**, comportamiento similar a WinForms, pongo los controles manualmente en la posición deseada.
- **WrapPanel**, coloca los hijos uno detrás de otro hasta que se queda sin espacio y salta de línea. Por defecto orientación horizontal, aunque se puede cambiar.
- **StackPanel**, como el WrapPanel con la diferencia que los controles hijos tratan de ocupar todo el espacio disponible en el contenedor.
- **DockPanel**, permite acoplar los controles a los lados del panel que se le indiquen. Si hay varios controles el último ocupara todo el espacio disponible. Se puede hacer lo mismo con Grid aunque requiere una mayor configuración por lo que puede ser útil para casos sencillos.
- **Grid**, permite definir filas y columnas, a las que se les puede indicar lo que ocupan en pixeles, porcentaje o que se ajusten automáticamente. Lo controles hijos se pueden posicionar en el lugar indicado indicando la fila y la columna.
- **UniformGrid**, como el Grid pero en este caso el tamaño de las filas y de las columnas es siempre el mismo.

Canvas

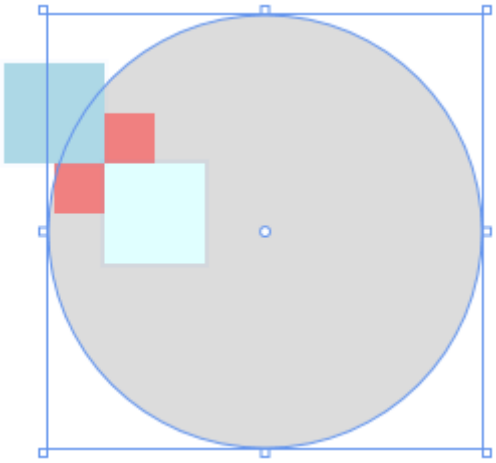
Comportamiento similar al de WinForms, los controles se posicionan manualmente en la posición deseada.

Con este panel podríamos construir aplicaciones en las que el usuario pueda dibujar o representar formas libres.

Propiedades (ELEMENTO HIJOS):

- Canvas.Left, número de pixeles desde la izquierda.
- Canvas.Top, número de pixeles desde arriba.
- Canvas.Right, número de pixeles desde la derecha, no incluido en propiedades.
- Canvas.Bottom, número de pixeles desde abajo, no incluido en propiedades.
- Panel.ZIndex, índice Z, orden de superposición de las capas.

```
<Canvas>
<Ellipse Panel.ZIndex="2" Fill="Gainsboro" Height="216"
Canvas.Left="47" Canvas.Top="41" Width="216" />
</Canvas>
```



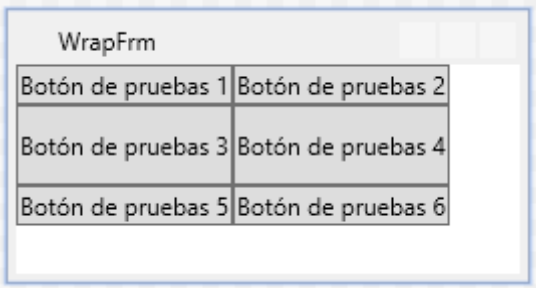
NOTA: La imagen anterior incluye otras formas geométricas en el canvas.

WrapPanel

Coloca los elementos hijos uno detrás de otro hasta que se queda sin espacio y salta de línea. Por defecto emplea una orientación horizontal, aunque se puede cambiar por vertical.

NOTA: con la orientación horizontal la fila adaptará su altura a la del elemento más alto, con orientación vertical la columna adaptará su anchura a la del elemento más ancho. Para evitar este comportamiento asignar el ancho y el alto al elemento.

```
<WrapPanel Orientation="Horizontal">
      <Button>Botón de pruebas 1</Button>
      <Button>Botón de pruebas 2</Button>
      <Button>Botón de pruebas 3</Button>
      <Button Height="40">Botón de pruebas 4</Button>
      <Button>Botón de pruebas 5</Button>
      <Button>Botón de pruebas 6</Button>
</WrapPanel>
```



StackPanel

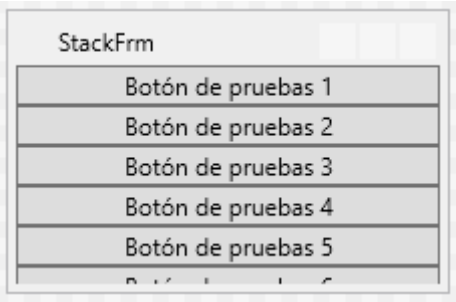
Como el "WrapPanel" con la diferencia que los controles hijos tratan de ocupar todo el espacio disponible en el contenedor.

Fíjate que este panel no hace scroll, si se queda sin espacio los controles hijos no se ven.

Por defecto la orientación es vertical, los hijos se apilan uno encima del otro, fíjate que en este caso se les estira para ocupar todo el ancho disponible. Con la orientación horizontal los hijos se posicionan de izquierda a derecha ocupando en este caso todo el alto disponible.

Este comportamiento de estirar los hijos se puede modificar con las propiedades de alineación en los NODOS HIJOS.

```
<StackPanel Orientation="Vertical">
      <Button HorizontalAlignment="Left">Botón de pruebas 1</Button>
      <Button HorizontalAlignment="Center">Botón de pruebas 2</Button>
      <Button HorizontalAlignment="Right">Botón de pruebas 3</Button>
      <Button HorizontalAlignment="Left">Botón de pruebas 4</Button>
      <Button HorizontalAlignment="Center">Botón de pruebas 5</Button>
      <Button HorizontalAlignment="Right">Botón de pruebas 6</Button>
</StackPanel>
```



DockPanel

Permite acoplar los controles a los lados del panel que se le indiquen. Si hay varios controles el último ocupara todo el espacio disponible. Se puede hacer lo mismo con Grid aunque requiere una mayor configuración por lo que puede ser útil para casos sencillos.

Propiedades

- **LastChildFill**="False", hace que el último elemento añadido no ocupe todo el espacio disponible.

Propiedades (ELEMENTOS HIJOS)

- **DockPanel.Dock**, establece el lugar al que se acopla, Left, Top, Right, Bottom. NO HAY valor para FILL.

```
<DockPanel LastChildFill="False">
      <Button DockPanel.Dock="Top" Height="50">Arriba</Button>
      <Button DockPanel.Dock="Bottom" Height="50">Abajo</Button>
      <Button DockPanel.Dock="Left" Width="60">Izquierda</Button>
      <Button DockPanel.Dock="Left" Width="60">Izquierda</Button>
      <Button DockPanel.Dock="Right" Width="60">Derecha</Button>
      <Button DockPanel.Dock="Right" Width="60">Derecha</Button>
</DockPanel>
```

Grid

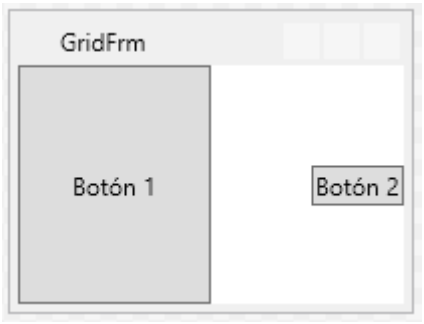
Permite definir filas y columnas, a las que se les puede indicar lo que ocupan en pixeles, proporcional (*) o que se ajusten automáticamente. Lo controles hijos se pueden posicionar en el lugar indicado indicando la fila y la columna.

Si no se indica la posición de los elementos es posible que se sobrepongan. **Si no se indica nada el elemento supondrá que se encuentra en la columna 0 y en la fila 0**.

A la hora de posicionar un elemento deberemos indicar la fila, columna o ambas. No existe el concepto de celda como tal.

Los elementos que incluyamos en una "celda" trataran de ocupar todo el espacio disponible salvo que se juegue con la alineación del contenido. En el siguiente ejemplo el botón de la primera columna ocupa todo su espacio, en cambio, el segundo botón posicionado en segunda columna se posiciona en centro derecha ocupando el mínimo espacio posible.

```
<Grid>
      <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Button>Botón 1</Button>
      <Button Grid.Column="1" VerticalAlignment="Center"
HorizontalAlignment="Right">Botón 2</Button>
</Grid>
```



Podemos asignar el espacio de las filas y de las columnas de 3 formas distintas.

- Absoluta: indicamos el número de pixeles
- Estrella *: tamaño relativo, la estrella simboliza el espacio correspondiente que le toca al elemento tras repartir de forma igualitaria. Tomando esta referencia podemos poner 2* para que toque el doble, 1* para lo justo, 10* para 10 veces lo correspondiente. Fíjate que el tamaño correspondiente final depende también de lo que pidan el resto de elementos.
- Auto: espacio necesario por el control.

El grid en primer lugar asigna tamaño a los elementos absolutos (los "auto" también tienen un tamaño fijo) y después distribuye el espacio que quede disponible.

```
<Grid>
      <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2*" />
            <ColumnDefinition Width="1*" />
            <ColumnDefinition Width="1*" />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="1*" />
            <RowDefinition Height="1*" />
      </Grid.RowDefinitions>
      <Button>Botón 1</Button>
      <Button Grid.Column="1">Botón 2</Button>
      <Button Grid.Column="2">Botón 3</Button>
      <Button Grid.Row="1">Botón 4</Button>
      <Button Grid.Column="1" Grid.Row="1">Botón 5</Button>
      <Button Grid.Column="2" Grid.Row="1">Botón 6</Button>
      <Button Grid.Row="2">Botón 7</Button>
      <Button Grid.Column="1" Grid.Row="2">Botón 8</Button>
      <Button Grid.Column="2" Grid.Row="2">Botón 9</Button>
</Grid>
```

Grid, combinar celdas

Una vez definida la red de filas y columnas es posible que necesitemos que un elemento ocupe varias celdas. Esto es posible con las propiedades de elemento **Grid.ColumnSpan** y **Grid.RowSpan**. A partir de la posición de comienzo de la celda podemos cuantas columnas queremos adherir (Grid.ColumnSpan) y cuantas filas (Grid.RowSpan).

```
<Grid>
      <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
```

```
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Button Grid.ColumnSpan="2">Botón 1</Button>
        <Button Grid.Column="3">Botón 2</Button>
        <Button Grid.Row="1">Botón 3</Button>
        <Button Grid.Column="1" Grid.Row="1" Grid.RowSpan="2"
Grid.ColumnSpan="2">Botón 4</Button>
        <Button Grid.Column="0" Grid.Row="2">Botón 5</Button>
</Grid>
```

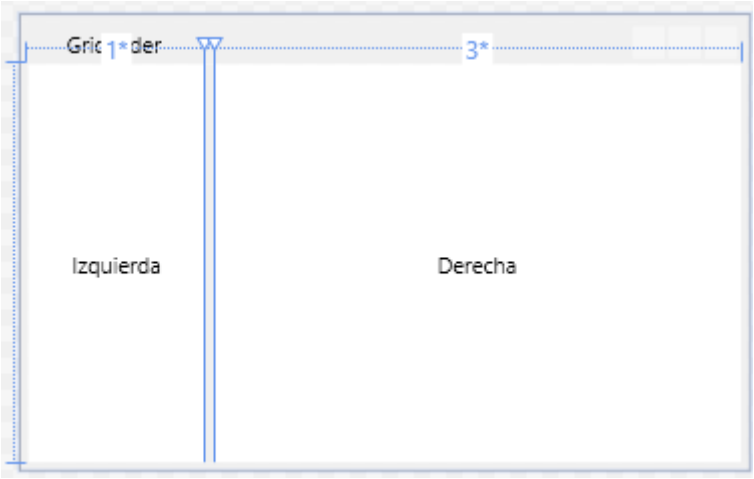
GridSplitter

Permite dividir la vista al estilo del "SplitContainer" de WinForms. En este caso el control "GridSplitter" **únicamente es un separador y nosotros deberemos construir un grid con 3 partes, la izquierda, la central o divisoria y la derecha**.

Podemos orientarla verticalmente u horizontalmente, para ello deberemos construir un grid con columnas o con filas según sea el caso. Después únicamente debemos indicar que la columna central se corresponde con un GridSplitter. Las otras dos columnas pueden ser cualquier cosa, un panel, un textblock, etc…

NOTA: es importante definir el tamaño de la columna separadora, tanto en el grid como en el splitter. En el primer caso para que no ocupe el tamaño por defecto (*) y en el segundo para que pueda ser seleccionada en tiempo de ejecución (sin ancho no se verá).

```
<Grid>
        <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="5"/>
                <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Column="0" VerticalAlignment="Center"
HorizontalAlignment="Center">Izquierda</TextBlock>
        <GridSplitter Grid.Column="1" Width="5"/>
        <TextBlock Grid.Column="2" VerticalAlignment="Center"
HorizontalAlignment="Center">Derecha</TextBlock>
</Grid>
```



Sería más correcto que el ancho de la columna 0 tuviera un tamaño mínimo o un control "StackPanel" relleno. La columna para los datos tiene un ancho "Auto" para adaptarse a la redimensión de la ventana.

Consideraciones finales Grid.

No se trata, pero si fuera necesario desplegar la vista en un navegador o incluso definir un formulario que se adapte a la resolución deberemos considerar dividir la vista con un Grid para posicionar los controles.

Controles de usuario y controles personalizados

La idea de un control personalizado es la de agrupar la funcionalidad de algunos controles existentes adaptándolos a nuestras necesidades y con la finalidad de poder reutilizarlo fácilmente en nuestros proyectos.

Nuestros propios controles contaran con 2 partes, el fichero .XAML y el archivo de Code Behind. Para crear un nuevo control debemos heredar de alguna de las clases preparadas para ello. En función de lo que deseemos construir heredaremos de las clases "UserControl", "ContentControl", "Control", "FrameworkElement", o "UIElement". El orden de estos controles es relevante porque de izquierda a derecha tendremos de menos a más libertad (y también de menos a más trabajo de codificación).

Si te fijas en el proyecto a la hora de agregar tenemos la opción de incluir un control de usuario WPF.

NOTA: Después de desarrollar el control hay que COMPILARLO para que no casque al usarlo.

Espacios de nombres.

- x: hace referencia a los controles de XAML.
- d: hace referencia al espacio de nombres de los diseñadores por ejemplo al desarrollar el control tenemos dos propiedades de tiempo de diseño, d:DesignHeight="200" d:DesignWidth="300. Estas medidas solo tienen efecto a la hora de desarrollar, en ejecución aplica el Height y el Width que indiquemos.

- local: hace referencia al proyecto en el que estamos trabajando, si hemos desarrollado los controles en el mismo proyecto de la aplicación principal accederemos con este a nuestros nuevos controles.
- common: si empaquetamos los controles en una librería y la añadimos como referencia la incluiremos con esta variable. Esta es la recomendación de Microsoft aunque otras fuentes emplean “uc” como referencia a los controles de usuario.

https://docs.microsoft.com/es-es/windows/uwp/xaml-platform/xaml-namespaces-and-namespace-mapping

Atributos

Para la clase

- [ContentProperty("Content")] o [ContentProperty("Inlines")]
- [DefaultProperty("Content")]

Para las propiedades

- [Browsable(false)] -> indica si la propiedad sale en la ventana de propiedades
- [ReadOnly(true)] -> indica si la propiedad es de solo lectura, útil para que el diseñador no trate de escribir una propiedad con únicamente un setter
- [Bindable(true)] -> indica que la propiedad puede enlazarse
- [Category("Común")] -> indica en que grupo de la caja de propiedades aparece, si no existe lo crea PROBADA
- [Description(“Obtiene o establece el valor de propiedad”)] -> ayuda que se muestra en el diseñador.
- [DefaultValue("Titulo control")] -> valor por defecto

Para indicar como se escribe en el control.

- [DesignerSerializationVisibility(DesignerSerializationVisibility.Content)]

Extendiendo WPF, documentación antigua

https://docs.microsoft.com/en-us/previous-versions/bb546938(v=vs.110)?redirectedfrom=MSDN

Código

```
namespace Demo8
{
    /// <summary>
    /// Lógica de interacción para UCEntradaLimitada.xaml
    /// </summary>
    public partial class UCEntradaLimitada : UserControl
    {
        public UCEntradaLimitada()
        {
            InitializeComponent();
            //El datacontext establece el objeto al que se le pide
información en un binding
```

```
            //En este caso permite leer las propiedades Titulo y
LongitudMaxima
            this.DataContext = this;
        }

        [Category("Común")]
        [Description("Obtiene o estable el valor para el título")]
        public string Titulo { get; set; }

        [Category("Propias control")]
        [Description("Obtiene o estable el tamaño máximo de entrada")]
        public int LongitudMaxima { get; set; }
    }
}
```

Maquetado

```
<UserControl x:Class="Demo8.UCEntradaLimitada"

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:local="clr-namespace:Demo8"
             mc:Ignorable="d"
             d:DesignHeight="200" d:DesignWidth="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Label Content="{Binding Titulo}"/>
        <Label Grid.Column="1">
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding
ElementName=txtEntradaLimitada, Path=Text.Length}" />
                <TextBlock Text="/"/>
                <TextBlock Text="{Binding LongitudMaxima}"/>
            </StackPanel>
        </Label>
        <TextBox MaxLength="{Binding LongitudMaxima}" Grid.Row="1"
Grid.ColumnSpan="2" Name="txtEntradaLimitada"
ScrollViewer.VerticalScrollBarVisibility="Auto" TextWrapping="Wrap" />
    </Grid>
</UserControl>
```

Para usarlo en una ventana

```
<local:UCEntradaLimitada Titulo="Introduce un texto:"
LongitudMaxima="60" Height="44" VerticalAlignment="Top">
</local:UCEntradaLimitada>
```

Ligado de datos en WPF

En WPF los binding son una herramienta de uso más común que en WinForms, vamos a ver poco a poco que se puede hacer. A modo de resumen podremos enlazar un control con otro control, un control con una propiedad de la clase de respaldo mediante la propiedad datacontext, y de lo último, asignación manual de la fuente de datos, podemos deducir que el origen será cualquier objeto.

El aspecto de un Binding es el siguiente

{Binding Path=Text, ElementName=txtValor}

El campo "Path" es opcional cuando lo que se quiere mostrar la propiedad por defecto (atributo DefaultProperty), por lo que en este caso es redundante.

Lo normal es que la propiedad .Text contenga los datos, en el siguiente ejemplo vamos a asignar a un control los datos de otro control.

```
<TextBox Name="txtValor" />
<WrapPanel Margin="0,10">
      <TextBlock Text="Valor: " FontWeight="Bold" />
      <TextBlock Text="{Binding Path=Text, ElementName=txtValor}" />
</WrapPanel>
```

Usando la fuente de datos o "DataContext"

La clase FrameworkElement define la propiedad DataContext, por defecto con valor null. La ventana la hereda por lo que podemos emplearla fácilmente desde la clase Code Behind. Por ejemplo si le asignamos el valor "this" haciendo referencia a la propia ventana podremos leer desde el XAML cualquier propiedad de la clase.

Maquetado

```
<StackPanel Margin="15">
      <WrapPanel>
            <TextBlock Text="Título ventana:  " />
            <TextBox Text="{Binding Title,
UpdateSourceTrigger=PropertyChanged}" Width="150" />
      </WrapPanel>
      <WrapPanel Margin="0,10,0,0">
            <TextBlock Text="Dimensiones ventana: " />
            <TextBox Text="{Binding Width}" Width="50" />
            <TextBlock Text=" x " />
            <TextBox Text="{Binding Height}" Width="50" />
      </WrapPanel>
</StackPanel>
```

Código

```
public partial class Bind2Frm : Window
{
      public Bind2Frm()
      {
            InitializeComponent();
            //La propia ventana es fuente de datos
            this.DataContext = this;
      }
```

```
}
```

En este caso **"UpdateSourceTrigger=PropertyChanged"** no sería obligatorio, al añadirlo conseguimos que cada vez que se escribe un carácter se actualice el título de la ventana. De omitirlo la actualización se produciría al perder el foco.

Crear un binding mediante código

Con contenido dinámico podría ser útil establecer en el Code Behind un enlace a datos. El código para ello es sencillo aunque poco intuitivo por lo que lo describimos.

Ten en cuenta que al instanciar el Binding le pasamos el parámetro "Path", la fuente en este caso es otro control. Y al hacer el binding tenemos que pasarle una propiedad definida en cada clase derivada FrameworkElement.

```
public Bind3Frm()
{
      InitializeComponent();
      Binding enlace = new Binding("Text"); //no haría falta el text
      enlace.Source = this.txtValor;
      lblValor.SetBinding(TextBlock.TextProperty, enlace);
}
```

Y el XAML correspondiente.

```
<StackPanel Margin="10">
      <TextBox Name="txtValor" />
      <WrapPanel Margin="0,10">
            <TextBlock Text="Valor: " FontWeight="Bold" />
            <TextBlock Name="lblValor" />
      </WrapPanel>
</StackPanel>
```

La propiedad UpdateSourceTrigger

Ya la vimos anteriormente pero vamos a detallar su comportamiento, esta propiedad del objeto encargado del binding se encarga de definir la forma en la que se actualizan los datos tras una actualización.

Los valores disponibles para esta propiedad son:

- Default, comportamiento establecido en el control.
- **PropertyChanged**, cuando cambia el contenido de la propiedad enlazada.
- LostFocus, actualiza al perder el foco.
- **Explicit**, para cuando queramos gestionar manualmente la actualización.

Documentación adicional

https://docs.microsoft.com/es-es/dotnet/api/system.windows.data.binding.updatesourcetrigger?view=net-5.0

XAML

```
<StackPanel Margin="15">
      <WrapPanel>
            <TextBlock Text="Título ventana:  " />
            <TextBox Name="txtTituloVentana" Text="{Binding Title,
UpdateSourceTrigger=Explicit}" Width="150" />
            <Button Name="btnActualizarFuente"
Click="btnActualizarFuente_Click" Margin="5,0"
Padding="5,0">*</Button>
      </WrapPanel>
      <WrapPanel Margin="0,10,0,0">
            <TextBlock Text="Dimensiones ventana: " />
            <TextBox Text="{Binding Width,
UpdateSourceTrigger=LostFocus}" Width="50" />
            <TextBlock Text=" x " />
            <TextBox Text="{Binding Height,
UpdateSourceTrigger=PropertyChanged}" Width="50" />
      </WrapPanel>
</StackPanel>
```

Código

```
private void btnActualizarFuente_Click(object sender, RoutedEventArgs
e)
{
      BindingExpression enlace =
this.txtTituloVentana.GetBindingExpression(TextBox.TextProperty);
      enlace.UpdateSource();
}
```

NOTA: no olvidar "**this.DataContext = this**" en el constructor para enlazar el propio objeto de la ventana con los controles.

Respondiendo a los cambios

Normalmente trabajamos con datos y colecciones de datos, cuando estos se modifican debemos estar atentos para refrescar la información mostrada. Existe la posibilidad de emplear colecciones de datos que nos notifiquen cuando alguno de los elementos contenidos se ha modificado. Esta colección se denomina **ObservableCollection<T>**, el único inconveniente es que el tipo T deberá implementar la interfaz **INotifyPropertyChanged**. Esta interfaz requiere que por cada propiedad modificable invoquemos al evento "PropertyChanged".

```
public class Usuario : INotifyPropertyChanged
{
      private string nombre;
      public string Nombre
      {
            get { return this.nombre; }
            set
            {
                  this.nombre = value;
                  this.OnPropertyChanged("Nombre");
            }
      }

      public event PropertyChangedEventHandler PropertyChanged;
```

```
        private void OnPropertyChanged(string nombrePropiedad)
        {
                if(this.PropertyChanged != null)
                {
                        this.PropertyChanged(this, new
PropertyChangedEventArgs(nombrePropiedad));
                }
        }
}
```

El constructor de la ventana establecerá el ItemsSource al objeto correspondiente y a partir de ese momento cualquier cambio en la colección refrescará automáticamente la información.

```
private ObservableCollection<Usuario> usuarios = new
ObservableCollection<Usuario>();

public Bind5Frm()
{
        InitializeComponent();
        this.lbUsuarios.ItemsSource = usuarios;
}
```

En este caso el XAML es un listbox y debemos indicar que propiedad del objeto mostrara.

```
<ListBox Name="lbUsuarios" DisplayMemberPath="Nombre"></ListBox>
```

NOTA: tenemos dos fuentes de datos, “DataSource” e “ItemsSource”, es importante entender la diferencia entre ellos. DataSource espera un objeto y e ItemsSourve espera un IEnumerable. El primero lo usaremos cuando tengamos que compartir la información con los nodos hijos (ventana con controles hijos por ejemplo), el segundo cuando va directo a un control, por ejemplo una lista. Además datasource requiere de los bindings con cada uno de los elementos hijos el segundo no requiere configurar la plantilla.

Conversión de valores con IValueConverter

Se puede plantear el caso de que el dato almacenado no tenga el formato correcto para la interfaz, por ejemplo los booleanos, true o false mostrados como sí o no, una fecha con un formato no esperado, un separador de decimales y miles con el carácter incorrecto o realizar alguna conversión de unidades, por ejemplo cambiar los 6 dígitos de un millón por una M.

Las clases a emplear son **IValueConverter** e **IMultiValueConverter**, ambas implementan los métodos **Convert**() y **ConvertBack**() para definir las conversiones en ambos sentidos.

Código

```
public class SiNoABooleanoConverter : IValueConverter
{
        public object Convert(object value, Type targetType, object
parameter, CultureInfo culture)
        {
                switch (value.ToString())
                {
                        case "yes":
                        case "si":
```

```
                        return true;
                case "not":
                case "no":
                        return false;
            }
            return false;
        }

        public object ConvertBack(object value, Type targetType, object
parameter, CultureInfo culture)
        {
            if(value is bool)
            {
                if((bool)value == true)
                {
                    return "si";
                }
                else
                {
                    return "no";
                }
            }
            return "no";
        }
}
```

XAML (recuerda compilar para que no se produzcan errores con los recursos)

```
<Window.Resources>
        <local:SiNoABooleanoConverter
x:Key="SiNoABooleanoConverter"></local:SiNoABooleanoConverter>
</Window.Resources>
<StackPanel Margin="10">
        <TextBox Name="txtValor" />
        <WrapPanel Margin="0,10">
            <TextBlock Text="El valor actual es: " />
            <TextBlock Text="{Binding ElementName=txtValor, Path=Text,
Converter={StaticResource SiNoABooleanoConverter}}"></TextBlock>
        </WrapPanel>
        <CheckBox IsChecked="{Binding ElementName=txtValor, Path=Text,
Converter={StaticResource SiNoABooleanoConverter}}" Content="Sí" />
</StackPanel>
```

Formateador de salida StringFormat

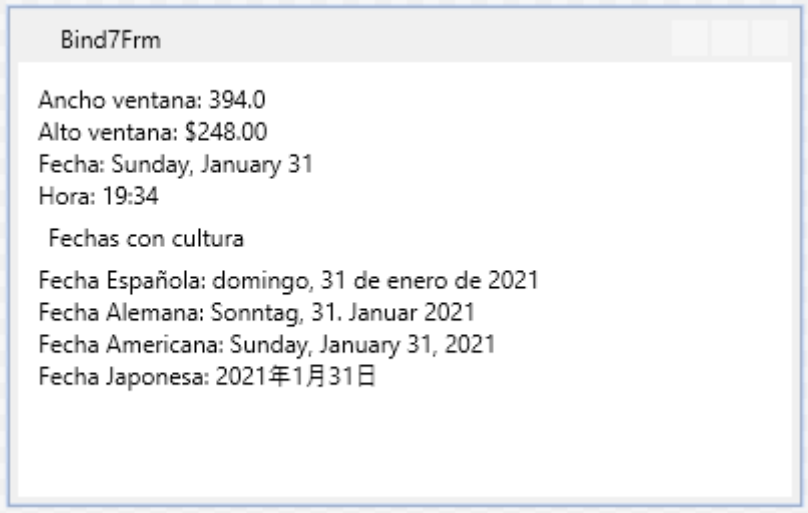
Al igual que con el método ToString() podemos asignar una máscara a los bindings, para ello indicaremos en la propiedad **StringFormat** del Binding la máscara deseada. Otra propiedad del binding es el **ConverterCulture** para asignar la cultura.

En el siguiente ejemplo aparte del StringFormat aparecen otras cosas interesantes, fíjate que le hemos asignado nombre a la ventana para poder referenciarla y además hemos añadido una referencia al espacio de nombres de System.

NOTA: este ejemplo es especialmente relevante ya que los controles WPF para trabajar con fechas ya no soportan las horas. Un textblock con una máscara es una forma fácil de introducir horas.

XAML

```
      ...
      Name="ventana"
      xmlns:system="clr-namespace:System;assembly=mscorlib">
<StackPanel Margin="10">
      <!-- La , es el separador de miles-->
      <TextBlock Text="{Binding ElementName=ventana, Path=ActualWidth,
StringFormat='Ancho ventana: {0:#,#.0}'}" />
      <TextBlock Text="{Binding ElementName=ventana,
Path=ActualHeight, StringFormat=Alto ventana: {0:C}}" />
      <TextBlock Text="{Binding Source={x:Static system:DateTime.Now},
StringFormat='Fecha: {0:dddd, MMMM dd}'}" />
      <TextBlock Text="{Binding Source={x:Static system:DateTime.Now},
StringFormat=Hora: {0:HH:mm}}" />
      <Label Content="Fechas con cultura" />
      <TextBlock Text="{Binding Source={x:Static system:DateTime.Now},
ConverterCulture='es-ES', StringFormat=Fecha Española: {0:D}}" />
      <TextBlock Text="{Binding Source={x:Static system:DateTime.Now},
ConverterCulture='de-DE', StringFormat=Fecha Alemana: {0:D}}" />
      <TextBlock Text="{Binding Source={x:Static system:DateTime.Now},
ConverterCulture='en-US', StringFormat=Fecha Americana: {0:D}}" />
      <TextBlock Text="{Binding Source={x:Static system:DateTime.Now},
ConverterCulture='ja-JP', StringFormat=Fecha Japonesa: {0:D}}" />
</StackPanel>
```



Depuración de ligaduras de datos

Cuando se produce un error de enlace a datos puede resultar complicado su depuración. En los casos más sencillos habremos cometido algún error tipográfico y únicamente deberemos corregirlo, pero cuando tenemos errores en la lógica la solución de los errores puede que no resulte tan clara.

En primer lugar, vamos a ver como cargar las herramientas de diagnóstico en la ventana y forzar la aparición de los mensajes con las trazas del enlace a datos.

En el siguiente ejemplo tenemos una ventana con el XAML correcto en la que hemos olvidado establecer la fuente de datos por lo que el enlace con la propiedad “Title” no mostrara la información.

Para cargar las herramientas de diagnóstico debemos añadir una referencia al espacio de nombres de **System.Diagnostics**. **IMPORTANTE, el asistente únicamente nos ofrece la**

referencia del ensamblado "System", deberemos cambiarla manualmente por "WindowsBase". Este espacio de nombres únicamente tiene una propiedad por lo que no será necesario que la memorices.

Esta opción nos mostrará en una ventana flotante todas las operaciones de enlace a datos en la aplicación así como su resultado.

```
      ...
      xmlns:diag="clr-namespace:System.Diagnostics;
assembly=WindowsBase"
      Title="BindDiagnosticoFrm" Height="250" Width="600">
<Grid Margin="10">
      <TextBlock Text="{Binding Title,
diag:PresentationTraceSources.TraceLevel=High}" />
</Grid>
```

Otra opción disponible es codificar un **IValueConverter** que no realice ninguna acción con los datos, de esta manera podremos depurar manualmente la asignación y lectura de los datos entre el XAML y el código.

Al convertidor le hemos añadido la instrucción "**Debugger.Break()**" que lo único que hace es establecer un punto de depuración por código. De esta forma podremos ver que está haciendo la aplicación al enlazar los datos.

```
public class DepuracionConverter : IValueConverter
{
      public object Convert(object value, Type targetType, object
parameter, CultureInfo culture)
      {
            Debugger.Break();
            return value;
      }

      public object ConvertBack(object value, Type targetType, object
parameter, CultureInfo culture)
      {
            Debugger.Break();
            return value;
      }
}
```

En el fichero XAML ajustamos el recurso y el binding. Recuerda el nombre de la ventana para poder referenciarla y la definición de un recurso local.

```
      ...
      Name="ventana">
<Window.Resources>
      <local:DepuracionConverter x:Key="DepuracionConverter" />
</Window.Resources>
<Grid Margin="10">
      <TextBlock Text="{Binding Title, ElementName=ventana,
Converter={StaticResource DepuracionConverter}}"/>
</Grid>
```

Introducción a los comandos WPF

Normalmente en WinForms cuando queremos realizar una acción común desde múltiples orígenes debemos configurar un evento en cada origen y ejecutar una copia del código, en el mejor de los casos este código se puede refactorizar en una única función. Por ejemplo, la opción de abrir suele aparecer repetida en el menú, en el menú contextual, en el atajo de teclado o en el evento doble click.

En WPF aparece un concepto nuevo, los comandos. Con un comando definimos una acción y la podemos asociar a cualquier control.

Para crear un comando debemos implementar la interfaz "**ICommand**" que define un evento y dos métodos:

- **Execute**(), ejecuta la acción.
- **CanExecute**(), indica si la acción está disponible para su ejecución o no.

Una vez definido el comando lo siguiente es enlazarlo a un control, en este caso aparece un nuevo tipo de binding, el "**CommandBinding**".

WPF trae de serie muchos comandos predefinidos agrupados en 5 categorías: ApplicationCommands, NavigationCommands, MediaCommands, EditingCommands y ComponentCommands.

Especialmente interesante es ApplicationCommands que incluye las acciones de nuevo, abrir, guardar, cortar, copiar o pegar entre otras.

XAML

```
<Window.CommandBindings>
      <CommandBinding Command="ApplicationCommands.New"
Executed="CommandBinding_Executed"
CanExecute="CommandBinding_CanExecute"/>
</Window.CommandBindings>
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
      <Button Command="ApplicationCommands.New">Nuevo</Button>
</StackPanel>
```

Código, manejadores

```
private void CommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
{
      MessageBox.Show("Ejecutando comando");
}

private void CommandBinding_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
      e.CanExecute = true;
}
```

El ejemplo anterior es muy sencillo, vamos a crear otro ejemplo en el que el botón únicamente esté disponible cuando tengamos contenido copiado previamente en el portapapeles.

XAML

```
<Window.CommandBindings>
      <CommandBinding x:Name="ComandoCortar"
Command="ApplicationCommands.Cut"
CanExecute="ComandoCortar_CanExecute"
Executed="ComandoCortar_Executed" />
      <CommandBinding x:Name="ComandoPegar"
Command="ApplicationCommands.Paste"
CanExecute="ComandoPegar_CanExecute" Executed="ComandoPegar_Executed"
/>
</Window.CommandBindings>
<DockPanel>
      <WrapPanel DockPanel.Dock="Top" Margin="3">
            <Button Command="ApplicationCommands.Cut"
Width="60">Cortar</Button>
            <Button Command="ApplicationCommands.Paste" Width="60"
Margin="3,0">Pegar</Button>
      </WrapPanel>
      <TextBox AcceptsReturn="True" Name="txtEditor" />
</DockPanel>
```

Código, manejadores.

NOTA: la gracia del canExecute es que deshabilita el botón.

```
private void ComandoCortar_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
      e.CanExecute = (txtEditor != null) && (txtEditor.SelectionLength
> 0);
}

private void ComandoCortar_Executed(object sender,
ExecutedRoutedEventArgs e)
{
      txtEditor.Cut();
}

private void ComandoPegar_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
      e.CanExecute = Clipboard.ContainsText();
}

private void ComandoPegar_Executed(object sender,
ExecutedRoutedEventArgs e)
{
      txtEditor.Paste();
}
```

El ejemplo anterior le podemos simplificar para los comportamientos predefinidos mediante bindings. Quiere decir, algunas acciones como copiar, pegar o cortar al portapapeles ya tienen una codificación por defecto de dicha funcionalidad.

XAML

```
<DockPanel>
      <WrapPanel DockPanel.Dock="Top" Margin="3">
```

```
            <Button Command="ApplicationCommands.Cut"
CommandTarget="{Binding ElementName=txtEditor}"
Width="60">Cortar</Button>
            <Button Command="ApplicationCommands.Paste"
CommandTarget="{Binding ElementName=txtEditor}" Width="60"
Margin="3,0">Pegar</Button>
      </WrapPanel>
      <TextBox AcceptsReturn="True" Name="txtEditor" />
</DockPanel>
```

Comandos personalizados WPF

Hemos visto los principios de los comandos en los ejemplos previos ahora vamos a ver cómo definir comandos completamente nuevos.

En este caso lo mejor es definir una clase estática con el código en el que cada función sea una acción para nuestros comandos. Seguido podremos definir un comando en el fichero xaml con el nombre de la función estática. (OJO, no son funciones sino variables de tipo comando estáticas que haremos readonly para evitar su modificación).

En el siguiente ejemplo vamos a implementar la funcionalidad de salir de la aplicación, y la vamos a asociar a la pulsación “Alt+f4”, a una opción de menú y a in botón.

Clase estática con la definición de los comandos, por ejemplo, cuando se pulsa alt+f4.

```
public static class ComandosPersonalizados
{
      public static readonly RoutedUICommand Salir = new
RoutedUICommand("Salir", "Salir", typeof(ComandosPersonalizados), new
InputGestureCollection()
      {
            new KeyGesture(Key.F4, ModifierKeys.Alt)
      });

      //Aquí definiría más comandos.
}
```

XAML, los comandos tienen el nombre de clase estática y del campo estático.

```
<Window.CommandBindings>
      <CommandBinding Command="local:ComandosPersonalizados.Salir"
CanExecute="CommandBinding_CanExecute"
Executed="CommandBinding_Executed"/>
</Window.CommandBindings>
<Grid>
      <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
      </Grid.RowDefinitions>
      <Menu>
            <MenuItem Header="Archivo">
                  <MenuItem
Command="local:ComandosPersonalizados.Salir" />
            </MenuItem>
      </Menu>
      <StackPanel Grid.Row="1" HorizontalAlignment="Center"
VerticalAlignment="Center">
            <Button
Command="local:ComandosPersonalizados.Salir">Exit</Button>
      </StackPanel>
```

```
</Grid>
```

Ahora solo queda definir el código manejador cuando para cuando se dispare el evento. Esto ya va en cada ventana. Es decir, defino un comando global a la aplicación y le codifico en cada ventana.

```
public partial class Comando4Frm : Window
{
	public Comando4Frm()
	{
		InitializeComponent();
	}

	private void CommandBinding_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
	{
		e.CanExecute = true;
	}

	private void CommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
	{
		Application.Current.Shutdown();
	}
}
```

Diálogos

El número de cuadros de diálogo en WPF es mucho más reducido que en WinForms y tendremos que emplear referencias. No nos extendemos en ellos porque su funcionalidad es idéntica a la de WinForms o son referencias a WinForms (System.Windows.Forms).

- MessageBox, muestra un popup al usuario con el mensaje y botones indicados.
- OpenFileDialog -> permite seleccionar un nombre de fichero de entrada.
- SaveFileDialog -> permite seleccionar un nombre de fichero de salida.

Si necesitamos compatibilidad total debemos evitar WinForms, las alternativas que tenemos son emplear algo de un tercero o implementarlos nosotros mismos.

OJO: de importar controles de WinForms y tratar de emplearlos en el código XAML las ventanas no tendrían las propiedades propias de WPF, además, aunque se pueda no se recomienda su uso.

Controles de interfaces comunes – Menús y la barra de progreso

Control menú WPF

Es un menú tradicional similar a los de WinForms aunque tenemos que tener en cuenta las siguientes consideraciones. A la hora de definirlos su construcción se parece a un desarrollo web, tenemos la etiqueta del menú y para los elementos que cuelgan debemos añadir las etiquetas correspondientes a cada uno de los subelementos disponibles.

- **Menu**, es el contenedor principal, puede que no se vea hasta que se le añada un hijo.

- **MenuItem**, son las opciones de menú, tanto en el menú como en los submenús desplegables.
    - **Header** -> atributo, texto del menú ítem.
    - Click -> atributo, manejador del evento.
    - Command -> atributo, nombre del comando a ejecutar.
    - IsCheckable -> añade una caja de check al ítem.
    - IsChecked -> indica si la caja de check del ítem esta activa o no.
    - **MenuItem.Icon** -> permite poner un icono al ítem, el content tendrá una etiqueta "**Image**" con la ruta de la imagen.
- **Separator**, la línea de separación tradicional.

TRUCO: WPF deja poner el menú en cualquier posición de la ventana. Para posicionarlo correctamente en la parte superior de la ventana emplea un "DockPanel" y asígnale al menú el atributo 'DockPanel.Dock="Top"'. El segundo objeto ocupará todo el espacio disponible.

XAML

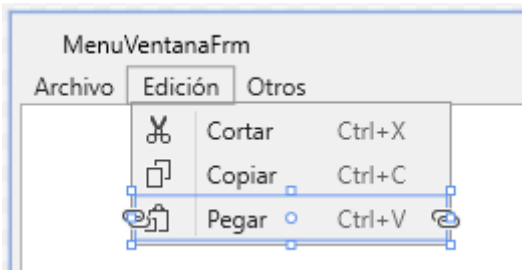
Le pongo nombre al comando para que los manejadores también lo incluyan.

Copiar, pegar y cortar tienen el código genérico de los manejadores. Con nuevo tengo que darle código porque no se conoce la acción por defecto.

NOTA: la opción "nuevo" podría tener un evento "click" sin más.

```
<Window.CommandBindings>
      <CommandBinding x:Name="cmdNuevo"
Command="ApplicationCommands.New" CanExecute="cmdNuevo_CanExecute"
Executed="cmdNuevo_Executed"/>
</Window.CommandBindings>
<DockPanel>
      <Menu DockPanel.Dock="Top">
            <MenuItem Header="Archivo">
                  <MenuItem x:Name="mnArchivoNuevo" Header="Nuevo"
Command="ApplicationCommands.New"/>
                  <MenuItem x:Name="mnArchivoAbrir" Header="Abrir" />
                  <MenuItem x:Name="mnArchivoGuardar" Header="Guardar"
/>
                  <Separator />
                  <MenuItem x:Name="mnArchivoSalir" Header="Salir"
Click="mnArchivoSalir_Click"/>
            </MenuItem>
            <MenuItem Header="Edición">
                  <MenuItem Header="Cortar"
Command="ApplicationCommands.Cut">
                        <MenuItem.Icon>
                              <Image Source="Images/Cut_6523.png" />
                        </MenuItem.Icon>
                  </MenuItem>
                  <MenuItem Header="Copiar"
Command="ApplicationCommands.Copy">
                        <MenuItem.Icon>
                              <Image Source="Images/Copy_6524.png"/>
                        </MenuItem.Icon>
                  </MenuItem>
                  <MenuItem Header="Pegar"
Command="ApplicationCommands.Paste">
```

```
                        <MenuItem.Icon>
                              <Image Source="Images/Paste_6520.png"/>
                        </MenuItem.Icon>
                  </MenuItem>
            </MenuItem>
            <MenuItem Header="Otros">
                  <MenuItem Header="Seleccionable" IsCheckable="True"
IsChecked="True" />
            </MenuItem>
      </Menu>
      <TextBox AcceptsReturn="True" />
</DockPanel>
```



Código

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

      private void mnArchivoSalir_Click(object sender, RoutedEventArgs
e)
      {
            this.Close();
            //Application.Current.Shutdown();
      }

      private void cmdNuevo_CanExecute(object sender,
System.Windows.Input.CanExecuteRoutedEventArgs e)
      {
            e.CanExecute = true;
      }

      private void cmdNuevo_Executed(object sender,
System.Windows.Input.ExecutedRoutedEventArgs e)
      {
            MessageBox.Show("Nuevo");
      }
}
```

Menú contextual WPF

Muy parecidos al control “Menu” empleamos la etiqueta “**ContextMenu**” para el menú contextual y “MenuItem” para las opciones. También se pueden añadir separadores con “Separator”.

Consideraciones.

- **El menú contextual se monta dentro de la etiqueta padre en la propiedad "EtiquetaPadre.ContextMenu", seguido va la etiqueta "ContextMenu".**
- **Algunos eventos del menú contextual se definen en el padre, como es el caso de "ContextMenu_Opening".**

Vamos a ver 3 ejemplo.

1. Menú dentro del padre sin ninguna opción adicional y gestión del evento opening.
2. Menú dentro del padre con iconos y comandos.
3. Creación de un recurso y gestión de apertura mediante código.

Primer ejemplo, estructura básica y evento apertura.

NOTA: Recuerda que el evento de apertura es el sitio ideal para gestionar los permisos de seguridad del usuario sobre el modelo, o las opciones disponibles.

XAML

```
<Button Content="Botón con menú contextual"
ContextMenuOpening="Button_ContextMenuOpening">
      <Button.ContextMenu>
            <ContextMenu>
                  <MenuItem Header="Elemento menú 1" />
                  <MenuItem Header="Elemento menú 2" />
                  <Separator />
                  <MenuItem Header="Elemento menú 3" />
            </ContextMenu>
      </Button.ContextMenu>
</Button>
```

Código función manejadora, **e.Handled** permite gestionar la apertura.

OJO: Fíjate que el evento se produce en el elemento padre.

```
private void Button_ContextMenuOpening(object sender,
ContextMenuEventArgs e)
{
      //Es el elemento padre o contenedor el que gestiona el evento.
NO EL MENÚ.
}
```

Segundo ejemplo, opciones de menú con iconos.

XAML

```
<TextBox Text="Este texto tiene un menú contextual">
      <TextBox.ContextMenu>
            <ContextMenu>
                  <MenuItem Command="ApplicationCommands.Cut">
                        <MenuItem.Icon>
                              <Image Source="/Images/cut_6523.png" />
                        </MenuItem.Icon>
                  </MenuItem>
                  <MenuItem Command="ApplicationCommands.Copy">
```

```
                        <MenuItem.Icon>
                              <Image Source="/Images/copy_6524.png" />
                        </MenuItem.Icon>
                  </MenuItem>
                  <MenuItem Command="ApplicationCommands.Paste">
                        <MenuItem.Icon>
                              <Image Source="/Images/paste_6520.png" />
                        </MenuItem.Icon>
                  </MenuItem>
            </ContextMenu>
      </TextBox.ContextMenu>
</TextBox>
```

Tercer ejemplo, definición de un menú contextual como recurso (reutilizable) y asignación por código.

XAML

```
<Window.Resources>
      <ContextMenu x:Key="ctmContexto">
            <MenuItem Header="Elemento 1"/>
            <MenuItem Header="Elemento 2"/>
      </ContextMenu>
</Window.Resources>

<Label Content="Contextual por código en una etiqueta"
MouseRightButtonUp="Label_MouseRightButtonUp"/>
```

NOTA: Si quisiéramos asignarlo en el XAML <Label ContextMenu="{StaticResource ctmContexto}" />

Código.

```
private void Label_MouseRightButtonUp(object sender,
MouseButtonEventArgs e)
{
      //recuperamos el recurso
      ContextMenu menu = this.FindResource("ctmContexto") as
ContextMenu;
      //posicionamos el menú
      menu.PlacementTarget = sender as Label;
      //abrimos el menú
      menu.IsOpen = true;
}
```

Barra de herramientas WPF

La barra de herramientas normalmente muestra las acciones de uso frecuente, con formato de botón, y agrupadas por funcionalidad.

La forma de definir una barra de herramientas comienza de manera similar al menú. Veamos las peculiaridades propias de la barra de herramientas.

En el caso de la barra de herramienta hay que tener en cuenta que se define en dos pasos, en primer lugar, una “bandeja”, por defecto vacía, sobre la que añadimos barras de herramientas.

Después, en cada bandeja podremos ubicar la mayoría de los controles de manera individual o combinada, por ejemplo, un icono más texto.

Las etiquetas comunes son:

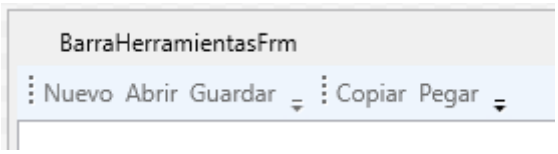
- **ToolBarTray**, bandeja base sobre la que colocar barras de herramientas.
- **ToolBar**, agrupación de herramientas dentro de la bandeja. Puedo incluir varias agrupando los controles habitualmente por funcionalidad.
    - Controles hijos, admite la mayoría, **Botones**, Etiquetas, Combos, etc…
    - **Separator**, permite incluir una línea separadora.
    - Los controles hijo heredan del ToolBar la propiedad "**ToolBar.OverflowMode**" que permite definir como se mostrarán los nodos en función del espacio disponible, es decir, que elementos se ocultan y cuales no a la hora de redimensionar el control. Los valores disponibles son:
        - **Always**, el nodo se muestra siempre
        - **AsNeeded**, el nodo se oculta en el mini icono desplegable si no hay espacio.
        - Never, el nodo siempre se oculta en el mini icono desplegable.

TRUCO: WPF deja poner la barra de herramientas en cualquier posición de la ventana. Para posicionarlo correctamente en la parte superior de la ventana emplea un "DockPanel" y asígnale al menú el atributo 'DockPanel.Dock="Top"'.

Primer ejemplo

XAML

```
<Window.CommandBindings>
      <CommandBinding Command="New"
CanExecute="CommandBinding_CanExecute"
Executed="CommandBinding_Executed"/>
      <CommandBinding Command="Open"
CanExecute="CommandBinding_CanExecute"
Executed="CommandBinding_Executed"/>
      <CommandBinding Command="Save"
CanExecute="CommandBinding_CanExecute"
Executed="CommandBinding_Executed"/>
</Window.CommandBindings>
<DockPanel>
      <ToolBarTray DockPanel.Dock="Top">
            <ToolBar>
                  <Button Command="New" Content="Nuevo" />
                  <Button Command="Open" Content="Abrir" />
                  <Button Command="Save" Content="Guardar" />
            </ToolBar>
            <ToolBar>
                  <Button Command="Cut" Content="Cortar"
ToolBar.OverflowMode="Always"/>
                  <Button Command="Copy" Content="Copiar"
ToolBar.OverflowMode="AsNeeded"/>
                  <Button Command="Paste" Content="Pegar"
ToolBar.OverflowMode="Never"/>
            </ToolBar>
      </ToolBarTray>
      <TextBox AcceptsReturn="True" />
</DockPanel>
```



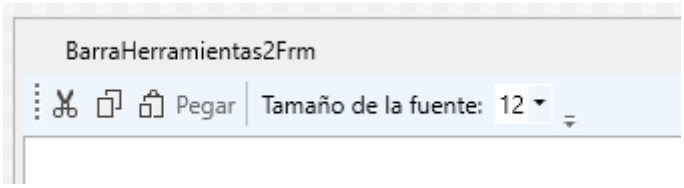
Código

```
private void CommandBinding_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
      e.CanExecute = true;
}

private void CommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
{
      //CUIDADIN, los comandos pueden ser lanzados de cualquier forma
      //este evento no debería ser compartido
      var origen = e.OriginalSource as ContentControl;
      MessageBox.Show(origen.Content.ToString());
}
```

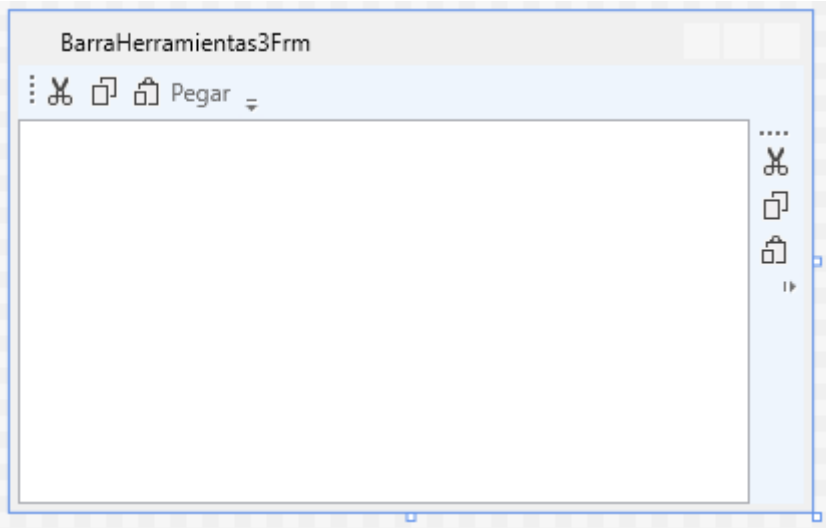
En el segundo ejemplo vamos a ver como decorar los elementos de la barra de herramientas. Tenemos botones que son únicamente imágenes, botones que tienen una imagen y una descripción, separadores y desplegables con su descripción.

```
<ToolBarTray DockPanel.Dock="Top">
      <ToolBar>
            <Button Command="Cut" ToolTip="Cortar selección al
portapapeles.">
                  <Image Source="/Images/cut_6523.png" />
            </Button>
            <Button Command="Copy" ToolTip="Copiar selección al
portapapeles.">
                  <Image Source="/Images/copy_6524.png" />
            </Button>
            <Button Command="Paste" ToolTip="Pegar desde el
portapapeles.">
                  <StackPanel Orientation="Horizontal">
                        <Image Source="/Images/paste_6520.png" />
                        <TextBlock Margin="3,0,0,0">Pegar</TextBlock>
                  </StackPanel>
            </Button>
            <Separator />
            <Label>Tamaño de la fuente:</Label>
            <ComboBox>
                  <ComboBoxItem>10</ComboBoxItem>
                  <ComboBoxItem IsSelected="True">12</ComboBoxItem>
                  <ComboBoxItem>14</ComboBoxItem>
                  <ComboBoxItem>16</ComboBoxItem>
            </ComboBox>
      </ToolBar>
</ToolBarTray>
```



En el tercer ejemplo vamos a ver que es posible montar varias bandejas de barras de herramientas. Una arriba y otra secundaria a la derecha.

```
<DockPanel>
      <ToolBarTray DockPanel.Dock="Top">
            <ToolBar>
                  <Button Command="Cut" ToolTip="Corta la selección al
portapapeles.">
                        <Image Source="/Images/cut_6523.png" />
                  </Button>
                  <Button Command="Copy" ToolTip="Copia la selección al
portapapeles.">
                        <Image Source="/Images/copy_6524.png" />
                  </Button>
                  <Button Command="Paste" ToolTip="Pega desde el
portapapeles.">
                        <StackPanel Orientation="Horizontal">
                              <Image Source="/Images/paste_6520.png" />
                              <TextBlock
Margin="3,0,0,0">Pegar</TextBlock>
                        </StackPanel>
                  </Button>
            </ToolBar>
      </ToolBarTray>
      <ToolBarTray DockPanel.Dock="Right" Orientation="Vertical">
            <ToolBar>
                  <Button Command="Cut" ToolTip="Copia la selección al
portapapeles.">
                        <Image Source="/Images/cut_6523.png" />
                  </Button>
                  <Button Command="Copy" ToolTip="Copia la selección
del portapapeles.">
                        <Image Source="/Images/copy_6524.png" />
                  </Button>
                  <Button Command="Paste" ToolTip="Pega desde el
portapapeles.">
                        <Image Source="/Images/paste_6520.png" />
                  </Button>
            </ToolBar>
      </ToolBarTray>
      <TextBox AcceptsReturn="True" />
</DockPanel>
```



Barra de estado WPF

La barra de estado es una pequeña barra que habitualmente se pone en la parte inferior de las aplicaciones para indicar información relevante y actual de nuestra aplicación. Por ejemplo, la posición del curso, datos sobre opciones activas, barras de progreso sobre un proceso de negocio, etc…

Como es habitual, para posicionar una barra de estado lo suyo es emplear un dock panel, una vez añadida la barra debemos añadir los elementos dentro de contenedores. Podemos añadir tantos contenedores como necesitemos.

Etiquetas.

- **StatusBar**, contenedor de la barra de estado.
    - StatusBar.ItemsPanel, permite definir la plantilla para la barra de estado.
        - ItemsPanelTemplate, permite indicar la plantilla, normalmente un grid con las columnas deseadas.
- **StatusBarItem**, contenedor para distribuir la información en la barra, añadiendo varios puedo distribuir el espacio en la barra.

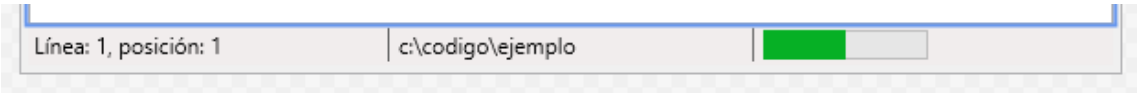
NOTA: recuerda que hasta que no añadas elementos a la barra es posible que no se vea.

```
<DockPanel>
      <StatusBar DockPanel.Dock="Bottom">
            <StatusBarItem>
                  <TextBlock x:Name="tbEstado"
Text="Estado"></TextBlock>
            </StatusBarItem>
      </StatusBar>
      <TextBox AcceptsReturn="True"/>
</DockPanel>
```

Vamos a crear un ejemplo un poco más complejo. El ejemplo tendrá 5 columnas, 3 para datos y 2 separadores. Añadiremos texto dinámico, estático y una barra de progreso.

XAML

```
<DockPanel>
      <StatusBar DockPanel.Dock="Bottom">
            <StatusBar.ItemsPanel>
                  <ItemsPanelTemplate>
                        <Grid>
                              <Grid.ColumnDefinitions>
                                    <ColumnDefinition Width="*" />
                                    <ColumnDefinition Width="Auto" />
                                    <ColumnDefinition Width="*" />
                                    <ColumnDefinition Width="Auto" />
                                    <ColumnDefinition Width="*" />
                              </Grid.ColumnDefinitions>
                        </Grid>
                  </ItemsPanelTemplate>
            </StatusBar.ItemsPanel>
            <StatusBarItem Grid.Column="0">
                  <TextBlock x:Name="tbPosicionCursor" Text="Línea: 1,
posición: 1"/>
            </StatusBarItem>
            <Separator Grid.Column="1" />
            <StatusBarItem Grid.Column="2">
                  <TextBlock Text="c:\codigo\ejemplo" />
            </StatusBarItem>
            <Separator Grid.Column="3" />
            <StatusBarItem Grid.Column="4">
                  <ProgressBar Width="90" Height="16" Value="50"/>
            </StatusBarItem>
      </StatusBar>
      <TextBox x:Name="txtEditor" AcceptsReturn="True"
SelectionChanged="txtEditor_SelectionChanged"/>
</DockPanel>
```



Código, la propiedad “CaretIndex” devuelve la posición del cursor en número de caracteres desde el comienzo del texto. A partir de aquí con las obtener el número de línea y obtener el índice en la línea podemos representar la posición actual del cursor.

```
private void txtEditor_SelectionChanged(object sender, RoutedEventArgs
e)
{
      int fila = this.txtEditor.GetLineIndexFromCharacterIndex(
this.txtEditor.CaretIndex);
      int columna = this.txtEditor.CaretIndex -
this.txtEditor.GetCharacterIndexFromLineIndex(fila);
      this.tbPosicionCursor.Text = "Línea: " + (fila+1) + ", posición:
" + (columna+1);
}
```

Introducción a WPF Rich Text Controls

Paso un poco de este apartado. Se puede trabajar con texto enriquecido, con imágenes, listas y tablas al estilo de Word. La clase con la que trabajan es el “FlowDocument”, la cual está pensada para presentar la información no para su edición.

Contamos con 3 visores.

- FlowDocumentScrollViewer, muestra la información de forma continua.
- FlowDocumentPageViewer, como el anterior, pero paginando la información.
- FlowDocumentReader, como el anterior, pero incluye herramientas como el zoom y modos de lectura.

Si quisiéramos editar un "FlowDocument" deberíamos envolverlo dentro de un control "RichTextBox" y ya podremos modificarlo fácilmente.

En un FlowDocument podemos emplear cualquier elemento de WPF, podríamos incluso crear informes dentro de nuestra aplicación sin motores externos.

NOTA: algunos controles para poder ser incrustados deben envolverse con la etiqueta "BlockUIContainer", por ejemplo, para el caso de un ListView con datos.

Se podría crear un bloc de notas contexto enriquecido, con distintos tipos de letras, párrafos, etc... Los ficheros podrían guardarse y cargarse con formato rft.

Esto escapa a los contenidos del curso.

Controles adicionales

Border

Este control permite definir bordes y el color de fondo, tiene sentido ya que los paneles no incluyen bordes y la única forma de añadirlos es a través de este decorador.

Atributos:

- BorderBrush, color del borde.
- BorderThickness, grosor del borde.
- CornerRadius, esquinas redondeadas, se indica el radio de la curva.
- Background, color de fondo, podríamos desarrollarlo y meterle un gradiente.

NOTA: recuerda que como en el maquetado web le puedo dar 1, 2 o 4 valores a los atributos para indicar cada lado.

```
<Grid Margin="10">
      <Border Background="GhostWhite" BorderBrush="Silver"
BorderThickness="1" CornerRadius="8,8,3,3">
            <StackPanel Margin="10">
                  <Button>Button 1</Button>
                  <Button Margin="0,10">Button 2</Button>
                  <Button>Button 3</Button>
            </StackPanel>
      </Border>
</Grid>
```

Slider

Barra deslizante entre un valor mínimo y máximo.

Propiedades:

- Maximum, valor máximo del selector.

- TickPlacement, lugar en el que van las marcas, por defecto se omiten.
- TickFrequency, cada cuanto se añade una marca.
- IsSnapToTickEnabled, indica que solamente son seleccionables los valores que coincidan con el valor de una marca.

En el siguiente ejemplo además se incluye una caja de texto con el valor bindeado.

```
<DockPanel VerticalAlignment="Center" Margin="10">
      <TextBox Text="{Binding ElementName=slValue, Path=Value,
UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right"
TextAlignment="Right" Width="40" />
      <Slider Maximum="255" TickPlacement="BottomRight"
TickFrequency="5" IsSnapToTickEnabled="True" Name="slValue" />
</DockPanel
```

Barra de progreso

Un indicador gráfico de progreso en barra.

Propiedades:

- Minimum, valor mínimo.
- Maximum, valor máximo.
- Value, valor actual.
- IsIndeterminate, para mostrar una barra moviéndose sin indicar el avance.

```
<Grid Margin="20">
      <ProgressBar Minimum="0" Maximum="100" Value="75" />
</Grid>
```

Navegador

Control con un navegador no muy avanzado. Etiqueta “WebBrowser”.

Métodos.

- .Navigate(string url), ir a la dirección indicada.
- .GoBack(), cargar la página anterior.
- .GoForward(), cargar la página siguiente.

Eventos.

- Navigating, se dispara cuando se abre un enlace dentro del control.

WindowsFormsHost

Podemos añadir los controles de winforms a un proyecto WPF, para ello debemos agregar las siguientes 2 referencias a nuestro proyecto.

- WindowsFormsIntegration
- System.Windows.Forms

NOTA: debemos evitar seriamente incluir WinForms en WPF.

Y añadimos el espacio de nombres.

```
      ...
      xmlns:wf="clr-
namespace:System.Windows.Forms;assembly=System.Windows.Forms">
<Grid>
      <WindowsFormsHost Name="wfhSample">
            <WindowsFormsHost.Child>
                  <wf:WebBrowser
DocumentTitleChanged="wbWinForms_DocumentTitleChanged" />
            </WindowsFormsHost.Child>
      </WindowsFormsHost>
</Grid>
```

Control GroupBox

Permite agrupar controles, el funcionamiento es similar al de WinForms aunque en este caso podemos incluir imágenes en el título del grupo. La etiqueta es “GroupBox”

Propiedades.

- Header, texto a mostrar. Se puede desarrollar.
    - Bloque GroupBox.Header, si queremos darle formato.

```
<GroupBox Header="Título del contenedor">
      ... Contenido, panel.
</GroupBox>
```

Otro ejemplo con una imagen en el título.

```
<GroupBox Margin="10" Padding="10">
      <GroupBox.Header>
            <StackPanel Orientation="Horizontal">
                  <Image Source="Images/group.png" Margin="3,0" />
                  <TextBlock FontWeight="Bold">Título del
contenedor</TextBlock>
            </StackPanel>
      </GroupBox.Header>
      ... Contenido, panel
```

Control Calendar

Muestra un calendario, por defecto con la fecha actual.

NOTA: debido a que es un control grande (ancho, alto) y WPF se parece a una web no redimensiona bien. Si quieres ajustarlo al tamaño del contenedor debes envolverlo con un “ViewBox”.

Propiedades

- DisplayDate, fecha por defecto, sino se muestra el día actual.

- SelectionMode, permite seleccionar múltiples días con la tecla control pulsada. Podemos indicar las opciones:
    - SingleRange, un único rango (días consecutivos en fila o columna).
    - MultipleRange, los días que se quieran.
- SelectedDate, fecha seleccionada, cuando funcionamos con una única fecha.
- SelectedDates, rangos de fecha seleccionados, cuando trabajamos con rangos.
- Calendar.BlackoutDates, permite definir rangos de fechas no seleccionables, los datos van en etiquetas “CalendarDateRange”.
- DisplayMode, permite seleccionar meses o años. Los valores disponibles son:
    - Year, permite navegar entre años y seleccionar un mes.
    - Decade, permite navegar entre décadas y seleccionar un año.

```
<Viewbox>
      <Calendar DisplayDate="01.01.2014" />
</Viewbox>
```

Control DatePicker

Permite seleccionar una fecha de forma más amigable, no deforma la vista al abrirse, que el calendario. La etiqueta es “DatePicker”.

Propiedades.

- DisplayDate, fecha por defecto, sino se muestra el día actual.
- **SelectedDate**, fecha seleccionada.
- **SelectedDateFormat**, formato de la fecha.
    - **Short**, fecha corta
    - **Long**, fecha larga
- Calendar.BlackoutDates, permite definir rangos de fechas no seleccionables, los datos van en etiquetas “CalendarDateRange”.

```
<DatePicker SelectedDate="2020-12-31" SelectedDateFormat="Long">
</DatePicker>
```

NOTA: Si te estas preguntando qué pasa con las horas hay que instalar un paquete de nuget con componentes adicionales llamado “**Extended WPF Toolkit**”. Que incluye muchos controles nuevos, y este caso un “DateTimePicker” con un comportamiento similar al de WinForms.

Otra opción es emplear una caja de texto con una máscara.

```
<TextBlock Text="{Binding TuPropiedadHora, StringFormat={}{0:HH:mm
tt}}" />
```

Control Extender

Es un pequeño circulo que oculta o expande los elementos que contenga. Evidentemente, puede contener cualquier cosa, lo recomendable es emplear un panel.

Propiedades.

- ExpandDirection, por defecto hacia abajo “Down” pero podríamos seleccionar otras como “Right”, “Up” o “Left”.
- IsExpanded, verdadero o falso si esta expandido.
- Header, para añadirle un título o contenido tras el botón.
    - Extender.Header, para cuando queramos emplear una imagen y texto en el título.

XAML

```
<Expander Margin="10">
    <Expander.Header>
    <DockPanel VerticalAlignment="Stretch">
        <Image Source="/Images/icono.png" Height="16"
DockPanel.Dock="Left" />
        <TextBlock FontStyle="Italic" Foreground="Green">Pulsa para
Mostrar/Ocultar...</TextBlock>
    </DockPanel>
    </Expander.Header>
    <TextBlock TextWrapping="Wrap" FontSize="18">
        Información ocultable.
    </TextBlock>
</Expander>
```

TabControl

El control de pestañas tradicional.

Etiquetas

- **TabControl**, elemento contenedor.
    - Propiedad **TabStripPlacement**, permite indicar en qué lado se ponen las pestañas, por defecto arriba, pero puedo Left, Rigth, Bottom.
- **TabItem**, etiqueta para cada pestaña.
    - Propiedad **Header**, texto a mostrar en la pestaña. Podría desarrollar la propiedad para darle el formato deseado.
        - TabItem.Header, y meterle un stackpanel horizontal con una imagen y un textblock.

Código

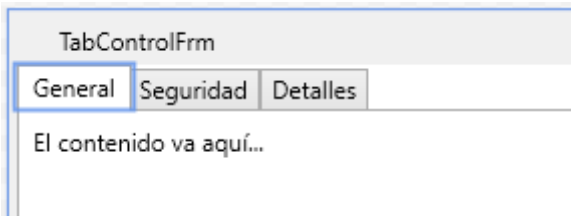
- tabControl.SelectedIndex, índice de la pestaña seleccionada.
- tabControl.SelectedItem, pestaña seleccionada.
- tabControl.Items, colección de pestañas.

Ejemplo sencillo.

XAML

```
<Grid>
      <TabControl>
            <TabItem Header="General">
                  <Label Content="El contenido va aquí..." />
            </TabItem>
            <TabItem Header="Seguridad" />
            <TabItem Header="Detalles" />
      </TabControl>
```

```
</Grid>
```

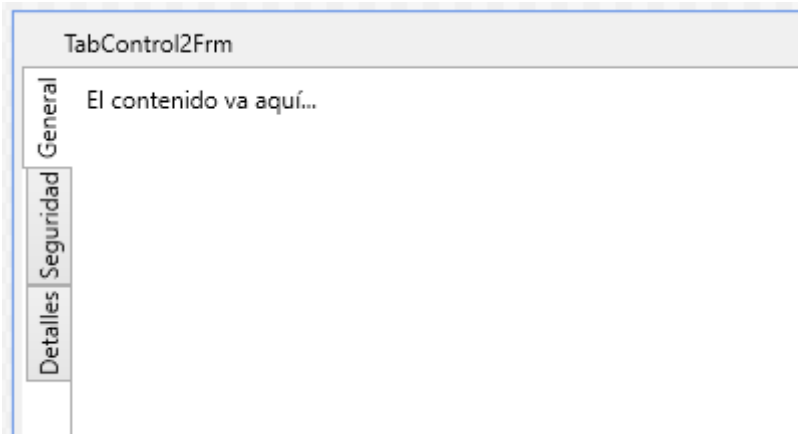


Ejemplo pestañas en lateral rotadas. En este caso modificamos la plantilla para el control.

```
<TabControl TabStripPlacement="Left">
      <TabControl.Resources>
            <Style TargetType="{x:Type TabItem}">
                  <Setter Property="HeaderTemplate">
                        <Setter.Value>
                              <DataTemplate>
                                    <ContentPresenter
Content="{TemplateBinding Content}">

      <ContentPresenter.LayoutTransform>
                                                <RotateTransform
Angle="270" />

      </ContentPresenter.LayoutTransform>
                                    </ContentPresenter>
                              </DataTemplate>
                        </Setter.Value>
                  </Setter>
                  <Setter Property="Padding" Value="3" />
            </Style>
      </TabControl.Resources>
      <TabItem Header="General">
            <Label Content="El contenido va aquí..." />
      </TabItem>
      <TabItem Header="Seguridad" />
      <TabItem Header="Detalles" />
</TabControl>
```



Se podría cambiar la plantilla y los eventos para darle un formato completamente distinto.

Controles de lista

Control ItemsControl

Permite mostrar datos en crudo en la aplicación de forma rápida. Todo lo que pongamos en el contenido se muestra sin más. Podemos definir una plantilla y organizar la salida.

Digamos que por defecto únicamente muestra información, no está pensado para interactuar con él.

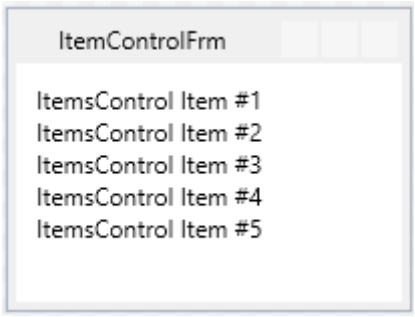
Etiquetas:

- ItemsControl
- ItemsControl.DataTemplate, por si queremos definir una plantilla.
- ItemsControl.ItemsPanel, permite definir un panel para el contenido
    - ItemsPanelTemplate

Ejemplo sencillo, mostrar cadenas

NOTA: incluimos también el control ScrollView para añadir las barras de scroll.

```
<ScrollViewer VerticalScrollBarVisibility="Auto"
HorizontalScrollBarVisibility="Auto">
	<ItemsControl>
		<system:String>ItemsControl Item #1</system:String>
		<system:String>ItemsControl Item #2</system:String>
		<system:String>ItemsControl Item #3</system:String>
		<system:String>ItemsControl Item #4</system:String>
		<system:String>ItemsControl Item #5</system:String>
	</ItemsControl>
</ScrollViewer>
```



Ejemplo con plantilla y enlace a datos

XAML, fíjate que en este caso va con bindings

```
<ItemsControl Name="icTareas">
	<ItemsControl.ItemTemplate>
		<DataTemplate>
			<Grid Margin="0,0,0,5">
				<Grid.ColumnDefinitions>
					<ColumnDefinition Width="*" />
					<ColumnDefinition Width="100" />
				</Grid.ColumnDefinitions>
				<TextBlock Text="{Binding Titulo}" />
				<ProgressBar Grid.Column="1" Minimum="0"
Maximum="100" Value="{Binding Porcentaje}" />
			</Grid>
		</DataTemplate>
	</ItemsControl.ItemTemplate>
```

```
</ItemsControl>
```

Código

```
List<Tarea> items = new List<Tarea>();
items.Add(new Tarea() { Titulo = "Curso de WPF", Porcentaje = 45 });
items.Add(new Tarea() { Titulo = "Aprender C#", Porcentaje = 80 });
items.Add(new Tarea() { Titulo = "Tareas", Porcentaje = 0 });

icTareas.ItemsSource = items;
```

La clase “Tarea” que emplearemos en los siguientes ejemplos.

```
public class Tarea
{
      public string Titulo { get; set; }
      public int Porcentaje { get; set; }
}
```

Control ListBox

Permite mostrar información de forma sencilla y seleccionarla para realizar una interacción sencilla con ella.

En este control si hace falta aparecen automáticamente las barras de scroll.

Propiedades

- **SelectionMode**, indica la forma en la que se seleccionan los elementos. Valores:
  - Single, un único elemento seleccionado.
  - Multiple, múltiples elementos seleccionables de forma libre.
  - Extended, como el anterior, pero hay que pulsar “Ctrl” para poder seleccionar.
- SelectedIndex, índice del elemento seleccionado, -1 para ninguno seleccionado.
- **SelectedItem**, elemento seleccionado.
- **SelectedItems**, colección de elementos seleccionados.
- SelectedValue, valor seleccionado, no elemento. Toma el valor de la propiedad indicada en SelectedValuePath.
- **Items**, elementos mostrados. Para añadir los elementos por código.

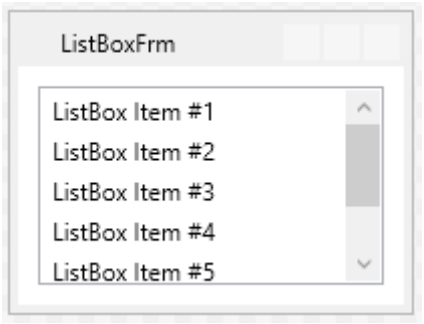
Eventos

- **SelectionChanged**, cuando cambia la selección.

Ejemplo sencillo.

XAML

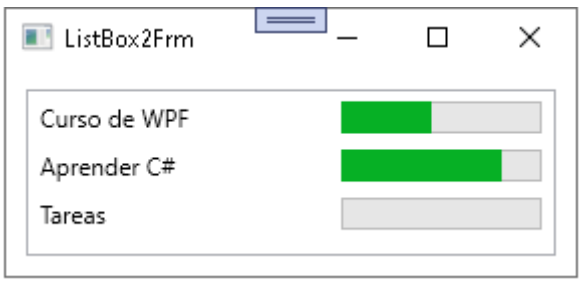
```
<ListBox>
      <ListBoxItem>ListBox Item #1</ListBoxItem>
      <ListBoxItem>ListBox Item #2</ListBoxItem>
      <ListBoxItem>ListBox Item #3</ListBoxItem>
      <ListBoxItem>ListBox Item #4</ListBoxItem>
      <ListBoxItem>ListBox Item #5</ListBoxItem>
      <ListBoxItem>ListBox Item #6</ListBoxItem>
</ListBox>
```



Ejemplo con objetos

XAML

```
<ListBox Name="lbTareas" HorizontalContentAlignment="Stretch">
      <ListBox.ItemTemplate>
            <DataTemplate>
                  <Grid Margin="0,2">
                        <Grid.ColumnDefinitions>
                              <ColumnDefinition Width="*" />
                              <ColumnDefinition Width="100" />
                        </Grid.ColumnDefinitions>
                        <TextBlock Text="{Binding Titulo}" />
                        <ProgressBar Grid.Column="1" Minimum="0"
Maximum="100" Value="{Binding Porcentaje}" />
                  </Grid>
            </DataTemplate>
      </ListBox.ItemTemplate>
</ListBox>
```



Código, mostramos el binding, el tema de los eventos es como siempre.

```
List<Tarea> items = new List<Tarea>();
items.Add(new Tarea() { Titulo = "Curso de WPF", Porcentaje = 45 });
items.Add(new Tarea() { Titulo = "Aprender C#", Porcentaje = 80 });
items.Add(new Tarea() { Titulo = "Tareas", Porcentaje = 0 });

lbTareas.ItemsSource = items;
```

Control ComboBox

Permite seleccionar un valor entre los disponibles en una lista. Trabaja con las etiquetas "ComboBox" para el control y "ComboBoxItem" para cada dato.

Etiqueta ComboBox

Propiedades

- **SelectedIndex**, índice del elemento seleccionado.
- **SelectedItem,** elemento seleccionado.
- **IsEditable**, permite al usuario escribir en el desplegable.
- IsTextSearchEnabled, permite escribir mostrando sugerencias entre los valores disponibles.
- IsTextSearchCaseSensitive, permite diferenciar entre mayúsculas y minúsculas. Usar junto con la propiedad "IsTextSearchEnabled".
- Items, colección de elementos, por si tenemos que editarla por código.

Eventos

- **SelectionChanged**, cuando la selección cambia.

Etiqueta ComboBoxItem

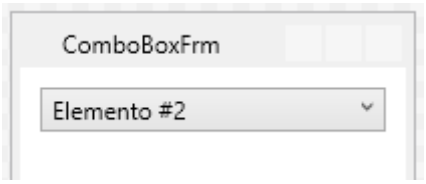
Propiedades

- **IsSelected**, indica si el elemento esta seleccionado.

Ejemplo sencillo.

XAML

```
<ComboBox>
      <ComboBoxItem>Elemento #1</ComboBoxItem>
      <ComboBoxItem IsSelected="True">Elemento #2</ComboBoxItem>
      <ComboBoxItem>Elemento #3</ComboBoxItem>
</ComboBox>
```

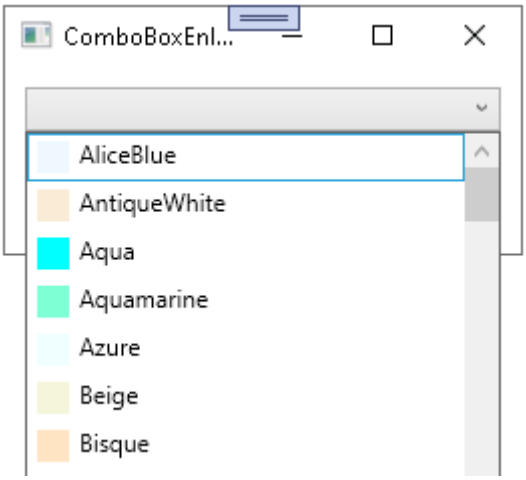


Ejemplo con binding y plantilla.

XAML

```
<StackPanel Margin="10">
      <ComboBox Name="cmbColores"
SelectionChanged="cmbColores_SelectionChanged">
            <ComboBox.ItemTemplate>
                  <DataTemplate>
                        <StackPanel Orientation="Horizontal">
```

```
                                <Rectangle Fill="{Binding Name}"
Width="16" Height="16" Margin="0,2,5,2" />
                                <TextBlock Text="{Binding Name}" />
                            </StackPanel>
                        </DataTemplate>
                    </ComboBox.ItemTemplate>
            </ComboBox>
</StackPanel>
```



Código

```
cmbColores.ItemsSource = typeof(Colors).GetProperties();


private void cmbColores_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
      Color selectedColor = (Color)(cmbColores.SelectedItem as
PropertyInfo).GetValue(null, null);
      this.Background = new SolidColorBrush(selectedColor);
}
```

Control ListView

Introducción

Al igual que en WinForms, este control es bastante potente a la hora de presentar información. Aunque debes de tener en cuenta que en WPF su forma de trabajar con la información no es del todo igual que en la versión de WinForms.

Etiquetas

- ListView, contenedor principal.
- **ListView.View**, **permite establecer el formato de tabla en el control**. Únicamente hay un tipo predefinido, el GridView.
    - **GridView**, contenedor para la vista de tabla.
        - **GridViewColumn**, permite definir una columna en la vista. Si se desea se le puede definir una plantilla para la celda.
            - **Header**, texto de la columna de cabecera.
            - **DisplayMemberBinding**, nombre de la propiedad a la que enlazar.

- ListView.ItemTemplate, permite definir una plantilla para los ListViewItem
    - DataTemplate, contenedor de la plantilla.
- **ListViewItem**, fila en el contenedor, envuelve un dato.
    - IsSelected, indica si la fila esta seleccionada.

Propiedades

- **ItemsSource**, datos en el control, para cuando los gestionamos de manera automática.
- **Items**, datos en el control, para cuando los gestionamos de manera manual.
- **SelectionMode,** indica el modo de selección de los elementos, valores:
    - **Single**, un único elemento.
    - Multiple, múltiples elementos
    - Extended, múltiples elementos pulsando la tecla de mayúsculas.

NOTA: no podemos usar “ItemsSource” e “Items” a la vez.

Ejemplo sencillo.

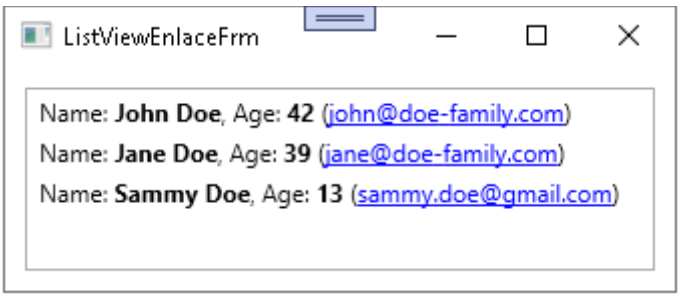
```
<ListView Margin="10">
      <ListViewItem>Elemento #1</ListViewItem>
      <ListViewItem IsSelected="True">Elemento #2</ListViewItem>
      <ListViewItem>Elemento #3</ListViewItem>
</ListView>
```

Ejemplo con plantilla de datos.

XAML

```
<ListView Margin="10" Name="lvUsuarios">
      <ListView.ItemTemplate>
            <DataTemplate>
                  <WrapPanel>
                        <TextBlock Text="Name: " />
                        <TextBlock Text="{Binding Nombre}"
FontWeight="Bold" />
                        <TextBlock Text=", " />
                        <TextBlock Text="Age: " />
```

```
                        <TextBlock Text="{Binding Edad}"
FontWeight="Bold" />
                        <TextBlock Text=" (" />
                        <TextBlock Text="{Binding Correo}"
TextDecorations="Underline" Foreground="Blue" Cursor="Hand" />
                        <TextBlock Text=")" />
                    </WrapPanel>
                </DataTemplate>
            </ListView.ItemTemplate>
</ListView>
```



Código

```
List<Usuario> items = new List<Usuario>();
items.Add(new Usuario() { Nombre = "John Doe", Edad = 42, Correo =
"john@doe-family.com" });
items.Add(new Usuario() { Nombre = "Jane Doe", Edad = 39, Correo =
"jane@doe-family.com" });
items.Add(new Usuario() { Nombre = "Sammy Doe", Edad = 13, Correo =
"sammy.doe@gmail.com" });
lvUsuarios.ItemsSource = items;
```

Clase Usuario, común para los siguientes ejemplos

```
public class Usuario
{
      public string Nombre { get; set; }
      public int Edad { get; set; }
      public string Correo { get; set; }
}
```

Ejemplo con un grid con columnas

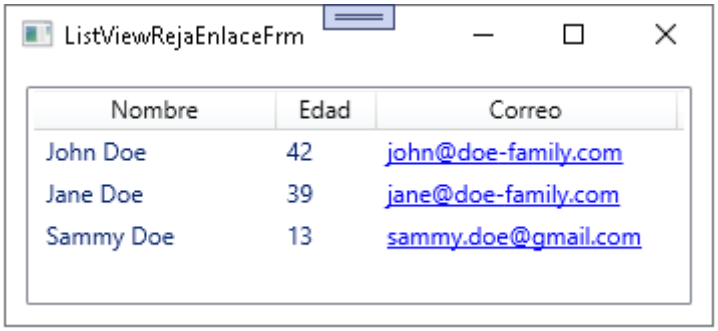
El control ListView por defecto en la propiedad "**View**" define un tipo de vista "**GridView**" (si queremos otros tipos hay que definirlos, único tipo disponible), el cual está pensado para mostrar la información con formato de tabla con una fila de cabecera.

En este caso deberemos definir las columnas que forman la tabla con la etiqueta "**GridViewColumn**". Si quisiéramos podríamos definir una plantilla adicional para la celda.

XAML

```
<ListView Margin="10" Name="lvUsuarios">
      <ListView.View>
            <GridView>
                  <GridViewColumn Header="Nombre" Width="120"
DisplayMemberBinding="{Binding Nombre}" />
```

```
                  <GridViewColumn Header="Edad" Width="50"
DisplayMemberBinding="{Binding Edad}" />
                  <GridViewColumn Header="Correo" Width="150">
                        <GridViewColumn.CellTemplate>
                              <DataTemplate>
                                    <TextBlock Text="{Binding Correo}"
TextDecorations="Underline" Foreground="Blue" Cursor="Hand" />
                              </DataTemplate>
                        </GridViewColumn.CellTemplate>
                  </GridViewColumn>
            </GridView>
      </ListView.View>
</ListView>
```



El código es el análogo al del anterior ejemplo.

Estilos columnas de cabecera

Un poco off topic pero bueno, recuerda que podemos establecer estilos para:

- Un elemento concreto.
- Para una etiqueta en el ámbito de una ventana.
- Para una etiqueta en el ámbito de la aplicación.

En el primer caso sería lo siguiente.

```
<ListView Margin="10" Name="lvUsuarios">
      <ListView.Resources>
            <Style TargetType="{x:Type GridViewColumnHeader}">
                  <Setter Property="HorizontalContentAlignment"
Value="Left" />
            </Style>
      </ListView.Resources>
      ...
```

A nivel de ventana

```
<Window.Resources>
      <Style TargetType="{x:Type GridViewColumnHeader}">
            <Setter Property="HorizontalContentAlignment" Value="Left"
/>
      </Style>
</Window.Resources>
```

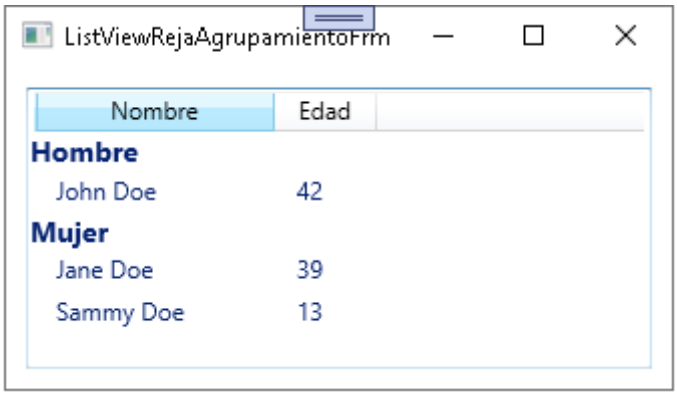
Para la aplicación sería similar y debemos definir el nuevo estilo en el fichero “App.xaml”

Agrupamiento ListView

Se pueden realizar agrupaciones mediante la definición de grupos. Por un lado, hay que definir un grupo, fíjate que el **Binding a la propidad "Name"**, es una propiedad del objeto que agrupa "PropertyGroupDescription".

```
<ListView Name="lvUsuarios">
      <ListView.View>
            <GridView>
                  <GridViewColumn Header="Nombre" Width="120"
DisplayMemberBinding="{Binding Nombre}" />
                  <GridViewColumn Header="Edad" Width="50"
DisplayMemberBinding="{Binding Edad}" />
            </GridView>
      </ListView.View>

      <ListView.GroupStyle>
            <GroupStyle>
                  <GroupStyle.HeaderTemplate>
                        <DataTemplate>
                              <TextBlock FontWeight="Bold"
FontSize="14" Text="{Binding Name}"/>
                        </DataTemplate>
                  </GroupStyle.HeaderTemplate>
            </GroupStyle>
      </ListView.GroupStyle>
</ListView>
```



Código, en este caso localizamos la vista y añadimos una entrada en **GroupDescription** en la que se indica el campo de agrupación.

```
List<Usuario> items = new List<Usuario>();
items.Add(new Usuario() { Nombre = "John Doe", Edad = 42, Correo =
"john@doe-family.com", Sexo=SexoEnum.Hombre });
items.Add(new Usuario() { Nombre = "Jane Doe", Edad = 39, Correo =
"jane@doe-family.com", Sexo=SexoEnum.Mujer });
items.Add(new Usuario() { Nombre = "Sammy Doe", Edad = 13, Correo =
"sammy.doe@gmail.com", Sexo=SexoEnum.Mujer });
lvUsuarios.ItemsSource = items;

CollectionView vista =
(CollectionView)CollectionViewSource.GetDefaultView(lvUsuarios.ItemsSo
urce);
PropertyGroupDescription descricionGrupo = new
PropertyGroupDescription("Sexo");
vista.GroupDescriptions.Add(descricionGrupo);
```

NOTA: se podría añadir un “Expander” al grupo, para ello habría que definir un nuevo estilo en “ListView.GroupStyle”.

Ordenando el ListView

En WPF la manera de ordenar un ListView cambia bastante respecto a Winforms, aunque los principios básicos son iguales. Primero debemos identificar la columna sobre la que se desea ordenar para posteriormente establecer un sentido de ordenación. Con estos dos datos definidos ahora deberemos crear una nueva “SortDescription” y añadírsela a la propiedad “.Items.SortDescriptions” del ListView.

OJO, te habrás dado cuenta de que es una colección por lo que deberemos limpiarla si no queremos concatenar varios criterios a la vez.

Comentar que WPF permite poner en la celda de cabecera una flecha indicando el sentido de ordenación. En realidad, WPF permite dibujar formas sobre cualquier elemento de la venta. Esto escapa al ámbito de este curso, pero si quisieras desarrollarlo la clase que debes extender es “Adorner”, y la clase estática “AdonerLayer” como asistente para gestionar estos “adornos”.

XAML, definimos el evento click y en la propiedad Tag indicamos un descriptor que nos permita conocer porque columna vamos a ordenar.

```
<ListView Name="lvUsuarios">
      <ListView.View>
            <GridView>
                  <GridViewColumn Width="120"
DisplayMemberBinding="{Binding Nombre}">
                        <GridViewColumn.Header>
                              <GridViewColumnHeader Tag="Nombre"
Click="GridViewColumnHeader_Click">Nombre</GridViewColumnHeader>
                        </GridViewColumn.Header>
                  </GridViewColumn>
                  <GridViewColumn Width="80"
DisplayMemberBinding="{Binding Edad}">
                        <GridViewColumn.Header>
                              <GridViewColumnHeader Tag="Edad"
Click="GridViewColumnHeader_Click">Edad</GridViewColumnHeader>
                        </GridViewColumn.Header>
                  </GridViewColumn>
                  <GridViewColumn Width="80"
DisplayMemberBinding="{Binding Sexo}">
                        <GridViewColumn.Header>
                              <GridViewColumnHeader Tag="Sexo"
Click="GridViewColumnHeader_Click">Sexo</GridViewColumnHeader>
                        </GridViewColumn.Header>
                  </GridViewColumn>
            </GridView>
      </ListView.View>
</ListView>
```

Código

```
private GridViewColumnHeader columnaOrdenada = null;
private ListSortDirection sentidoOrden;

public ListViewOrdenFrm()
{
      InitializeComponent();

      List<Usuario> items = new List<Usuario>();
      items.Add(new Usuario() { Nombre = "John Doe", Edad = 42, Correo
= "john@doe-family.com", Sexo = SexoEnum.Hombre });
      items.Add(new Usuario() { Nombre = "Jane Doe", Edad = 39, Correo
= "jane@doe-family.com", Sexo = SexoEnum.Mujer });
      items.Add(new Usuario() { Nombre = "Sammy Doe", Edad = 13,
Correo = "sammy.doe@gmail.com", Sexo = SexoEnum.Mujer });
      lvUsuarios.ItemsSource = items;
}

private void GridViewColumnHeader_Click(object sender, RoutedEventArgs
e)
{
      GridViewColumnHeader columnaClick = (sender as
GridViewColumnHeader);
      string ordenadoPor = columnaClick.Tag.ToString();

      if (columnaOrdenada == null)
      {
            columnaOrdenada = columnaClick;
            sentidoOrden = ListSortDirection.Descending;
      }
      //limpiamos los criterios actuales
      this.lvUsuarios.Items.SortDescriptions.Clear();
      //cambiamos el sentido de la columna actual
      if (columnaOrdenada == columnaClick)
      {
            if (sentidoOrden == ListSortDirection.Ascending)
            {
                  sentidoOrden = ListSortDirection.Descending;
            }
            else
            {
                  sentidoOrden = ListSortDirection.Ascending;
            }
            lvUsuarios.Items.SortDescriptions.Add(new
SortDescription(ordenadoPor, sentidoOrden));
      }
      else
      {
            //al pulsar en otra columna desaparecen los criterios de
ordenación existentes
            columnaOrdenada = null;
      }
}
```

Filtrado en ListView

Podemos crear filtros para un listview fácilmente, en cuanto al marcado no necesitaremos ninguna etiqueta especial ya que todo el trabajo de filtrado se realiza mediante código.

En primer lugar, debemos localizar la parte de la vista que gestiona los datos con el comando CollectionViewSource.GetDefaultView(fuente de datos) y seguido asignarle a la propiedad “Filter” de dicha vista una función que por cada objeto diga si debe o no mostrarse.

En el siguiente ejemplo se filtra una vista a partir de lo indicado en una caja de texto.

NOTA: en función de si trabajamos con ItemsSource o con la colección Items el código cambia ligeramente. En el primer caso podremos invocar a la función Refresh y en el segundo caso deberemos reasignar la propiedad Filter.

XAML, comentado los cambios para ItemsSource.

```
<DockPanel Margin="10">
      <TextBox DockPanel.Dock="Top" Margin="0,0,0,10" Name="txtFiltro"
TextChanged="txtFiltro_TextChanged" />
      <ListView Name="lvUsuarios">
            <ListView.View>
                  <GridView>
                        <GridViewColumn Header="Nombre" Width="120"
DisplayMemberBinding="{Binding Nombre}" />
                        <GridViewColumn Header="Edad" Width="50"
DisplayMemberBinding="{Binding Edad}" />
                  </GridView>
            </ListView.View>
      </ListView>
</DockPanel>
```

Código

```
public ListViewFiltroFrm()
{
      InitializeComponent();

      List<Usuario> items = new List<Usuario>();
      items.Add(new Usuario() { Nombre = "John Doe", Edad = 42, Correo
= "john@doe-family.com", Sexo = SexoEnum.Hombre });
      items.Add(new Usuario() { Nombre = "Jane Doe", Edad = 39, Correo
= "jane@doe-family.com", Sexo = SexoEnum.Mujer });
      items.Add(new Usuario() { Nombre = "Sammy Doe", Edad = 13,
Correo = "sammy.doe@gmail.com", Sexo = SexoEnum.Mujer });
      //lvUsuarios.ItemsSource = items;
      items.ForEach(x => lvUsuarios.Items.Add(x));

      //CollectionView view =
(CollectionView)CollectionViewSource.GetDefaultView(lvUsuarios.ItemsSo
urce);
      CollectionView vista =
(CollectionView)CollectionViewSource.GetDefaultView(lvUsuarios.Items);
      vista.Filter = FiltroVista;
}

private bool FiltroVista(object item)
{
      if (string.IsNullOrEmpty(this.txtFiltro.Text))
      {
            return true;
      }
      Usuario usuario = item as Usuario;
      return usuario.Nombre.Contains(this.txtFiltro.Text);
}
```

```
private void txtFiltro_TextChanged(object sender,
System.Windows.Controls.TextChangedEventArgs e)
{
      //CollectionViewSource.GetDefaultView(lvUsuarios.ItemsSource).Re
fresh();
      CollectionView vista =
(CollectionView)CollectionViewSource.GetDefaultView(lvUsuarios.Items);
      //El método Refresh no funciona con Items por lo que lo forzamos
      //volviendo a aplicar el filtro
      vista.Filter = FiltroVista;
}
```

Control TreeView

Introducción

Al igual que en los controles previos WPF va a permitir personalizar la presentación más allá de lo que nos permitía WinForms, un árbol básico será muy fácil de implementar. Si queremos algo más refinado en WPF podremos definir una plantilla para los datos y enlazarle una fuente de datos.

Etiquetas y atributos XAML

- **TreeView**, contenedor principal, los nodos hijos son del tipo “TreeViewItem”
- **TreeViewItem**, cada nodo visible
    - Header, texto del nodo.
    - **TreeViewItem.Header**, podemos desarrollarlo para darle formato, por ejemplo, imagen más texto.
    - **IsExpanded**, indica si el nodo está expandido o no.
    - IsSelected, indica si el nodo está seleccionado o no.

Eventos TreeViewItem

- **Expanded**, cuando se ha expandido el nodo.
- **Collapsed**, cuando se ha colapsado el nodo.
- **Selected**, cuando se ha seleccionado el nodo.

IMPORTANTE: puedes desde la etiqueta padre asignarles un evento a los hijos con “TreeViewItem.EVENTO=fun_manejadora“, por ejemplo “TreeViewItem.Expanded”.

NOTA: no hay eventos BeforeExpand o AfterSelect como en WinForms.

Con el control árbol podemos trabajar tanto con la fuente de datos “**ItemsSource**” como de forma manual mediante la propiedad “**Items**”.

De cualquier manera, si trabajamos con un conjunto de nodos que cambie podemos considerar el uso de “ObservableCollection<T>” junto con “ItemsSource” para que los nodos se gestionen automáticamente.

NOTA: TreeView soporta plantillas jerárquicas que en el caso de grupos o familias de datos puede ser una opción interesante para la presentación de la información. Cuando la jerarquía de la información es dinámica (más de 2 niveles) añade complejidad.

Ejemplo

```
<TreeView>
      <TreeViewItem Header="Nivel 1" IsExpanded="True">
            <TreeViewItem Header="Nivel 2.1" />
            <TreeViewItem Header="Nivel 2.2" IsExpanded="True">
                  <TreeViewItem Header="Nivel 3.1" />
                  <TreeViewItem Header="Nivel 3.2" />
            </TreeViewItem>
            <TreeViewItem Header="Nivel 2.3" />
      </TreeViewItem>
</TreeView>
```

En el siguiente ejemplo vamos a ver como trabajar con iconos y cargar dinámicamente nodos. Vamos a construir un explorador de carpetas sencillo.

XAML

```
<TreeView x:Name="tvExplorador"
              TreeViewItem.Expanded="tvExplorador_Expanded"
              TreeViewItem.Collapsed="tvExplorador_Collapsed">
</TreeView>
```

Código

Guardamos en 2 variables las imágenes para evitar instanciar un objeto por cada nodo.

El texto del nodo es un panel con una imagen y un texto, mediante los eventos “expand” y “collapse” ajustamos la fuente de la imagen.

```
private BitmapImage carpetaCerrada = new BitmapImage(
      new Uri("/Images/folder_Closed_16xLG.png",
UriKind.RelativeOrAbsolute));
private BitmapImage carpetaAbierta = new BitmapImage(
      new Uri("/Images/folder_Open_16xLG.png",
UriKind.RelativeOrAbsolute));

public TreeViewIconosFrm()
{
      InitializeComponent();
      CargarRaiz();
```

```
}

private void CargarRaiz()
{
      DriveInfo[] drives = DriveInfo.GetDrives();
      foreach (DriveInfo driveInfo in drives)
      {
            this.tvExplorador.Items.Add(CrearNodo(driveInfo));
      }
}

private TreeViewItem CrearNodo(object objeto)
{
      TreeViewItem item = new TreeViewItem();

      StackPanel stackPanel = new StackPanel() { Orientation =
Orientation.Horizontal };
      Image imagenCerrado = new Image()
      {
            Source = this.carpetaCerrada,
            Margin = new Thickness(0d, 0d, 3d, 0d)
      };
      TextBlock texto = new TextBlock() { Text = objeto.ToString() };
      stackPanel.Children.Add(imagenCerrado);
      stackPanel.Children.Add(texto);

      item.Header = stackPanel;
      item.Tag = objeto;
      item.Items.Add("Cargando..."); //nodo dummy
      return item;
}

private void tvExplorador_Expanded(object sender, RoutedEventArgs e)
{
      TreeViewItem item = e.Source as TreeViewItem;
      StackPanel cabecera = (StackPanel)item.Header;
      Image imagen = cabecera.Children[0] as Image;
      imagen.Source = this.carpetaAbierta;

      if ((item.Items.Count == 1) && (item.Items[0] is string))
      {
            item.Items.Clear();

            DirectoryInfo directorioActual = null;
            if (item.Tag is DriveInfo)
            {
                  directorioActual = (item.Tag as
DriveInfo).RootDirectory;
            }
            if (item.Tag is DirectoryInfo)
            {
                  directorioActual = (item.Tag as DirectoryInfo);
            }
            try
            {
                  foreach (DirectoryInfo subDir in
directorioActual.GetDirectories())
                  {
                        item.Items.Add(CrearNodo(subDir));
                  }
            }
```

```
            catch { }
        }
}

private void tvExplorador_Collapsed(object sender, RoutedEventArgs e)
{
        TreeViewItem item = e.Source as TreeViewItem;
        StackPanel cabecera = (StackPanel)item.Header;
        Image imagen = cabecera.Children[0] as Image;
        imagen.Source = this.carpetaCerrada;
}
```

En teoría se puede gestionar el intercambio de iconos mediante plantillas pero no queda una solución sencilla.

https://social.msdn.microsoft.com/Forums/en-US/ff1b6889-a766-4a23-a5b1-c59a64f36629/wpf-c-how-to-add-text-with-icons-in-treeview?forum=wpf

DataGrid

Es tan sencillo como añadir la etiqueta “DataGrid” y asignarle un “ItemsSource”, para que automáticamente se generan las columnas. Se puede personalizar para que muestre las columnas que se deseen. El problema de este control es que trabaja directamente sobre la fuente de datos por lo que en muchos casos no es la opción más aconsejable.

Estilos

A estas alturas ya te habrás dado cuenta que XAML funciona de una manera muy parecida a las tecnologías de desarrollo web, en concreto como al HTML.

Los estilos se aplican en orden.

1. General en la aplicación “Application.Resources”, fichero App.xaml
2. Locales en cada ventana “Window.Resources”.
3. Locales en cada control “CONTROL.Resources”.
4. Dentro de cada etiqueta a través de la definición de propiedades.

Dentro de una propiedad “Resources” tenemos la siguiente estructura.

- Style
    - TargetType, nombre de la etiqueta a la que aplica el estilo.
- Setter, para definir el valor por defecto de una propiedad.
    - Property, nombre de la propiedad a modificar.
    - Value, valor por defecto para la propiedad.

NOTA: los estilos se pueden aplicar tanto explícitamente como implícitamente. La diferencia entre un tipo de estilo y otro es que en la definición explicita el estilo tiene una “llave” o clave, asignada mediante la propiedad “x:Key”.

Ejemplo de estilos locales dentro del control.

```
<StackPanel Margin="10">
```

```
      <StackPanel.Resources>
            <Style TargetType="TextBlock">
                  <Setter Property="Foreground" Value="Gray" />
                  <Setter Property="FontSize" Value="24" />
            </Style>
      </StackPanel.Resources>
      <TextBlock>Header 1</TextBlock>
      <TextBlock>Header 2</TextBlock>
      <TextBlock Foreground="Blue">Header 3</TextBlock>
</StackPanel>
```

Ejemplo estilo global para todos los “TextBlock” de la aplicación.

Todos los TextBox de la aplicación tendrán el estilo definido ya que se define el estilo para un tipo de control especifico.

NOTA: el entorno va a dar un warning indicando la conveniencia de envolver los recursos de la aplicación dentro de una etiqueta “ResourceDictionay”. Al usar este diccionario los datos contenidos se cachean y únicamente se leen una vez, si no lo indicamos cada vez que se solicite el recurso deberá procesarse. Es por lo tanto una opción que mejora mucho el rendimiento general de la aplicación. OJO, en las ventanas no es tan necesario.

```
<Application ...>
    <Application.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Foreground" Value="Gray" />
            <Setter Property="FontSize" Value="24" />
        </Style>
    </Application.Resources>
</Application>
```

Ejemplo asignación de estilo explícito.

Es este caso definimos un estilo para un tipo de control especifico y además le damos una clave. En el control que queramos aplicar dicho estilo únicamente tenemos que pasarle a la propiedad la clave que identifique el estilo.

```
<Window ...>
    <Window.Resources>
        <Style x:Key="EstiloCabecera" TargetType="TextBlock">
            <Setter Property="Foreground" Value="Gray" />
            <Setter Property="FontSize" Value="24" />
        </Style>
    </Window.Resources>
    <StackPanel Margin="10">
        <TextBlock>Cabecera 1</TextBlock>
        <TextBlock Style="{StaticResource EstiloCabecera}">Cabecera
2</TextBlock>
        <TextBlock>Cabecera 3</TextBlock>
    </StackPanel>
</Window>
```

Trigger, DataTrigger y EventTrigger.

Los disparadores se definen dentro de la etiqueta “Style” dentro del apartado “Style.Triggers”. Tenemos 3 tipos de disparadores.

- Triggers
- DataTriggers
- EventTriggers

Trigger de propiedad

Se fija en una propiedad y cuando su valor es igual al deseado se dispara. Por ejemplo, cuando el ratón pasa por un texto este se subraya y cambia de color.

CONSEJO: Usar este trigger cuando el evento se produce en el mismo control que se modifica.

```
<TextBlock Text="Hola mundo, estilos" FontSize="28"
HorizontalAlignment="Center" VerticalAlignment="Center">
      <TextBlock.Style>
            <Style TargetType="TextBlock">
                  <Setter Property="Foreground" Value="Blue"></Setter>
                  <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value="True">
                              <Setter Property="Foreground" Value="Red"
/>
                              <Setter Property="TextDecorations"
Value="Underline" />
                        </Trigger>
                  </Style.Triggers>
            </Style>
      </TextBlock.Style>
</TextBlock>
```

DataTrigger

Se establece un binding con una propiedad de un control, cuando el valor es igual al deseado se dispara.

CONSEJO: Usar este trigger cuando el evento se produce en un control distinto del que se modifica.

```
<CheckBox Name="cbSample" Content="Hola mundo?" />
<TextBlock HorizontalAlignment="Center" Margin="0,20,0,0"
FontSize="48">
      <TextBlock.Style>
            <Style TargetType="TextBlock">
                  <Setter Property="Text" Value="No" />
                  <Setter Property="Foreground" Value="Red" />
                  <Style.Triggers>
                        <DataTrigger Binding="{Binding
ElementName=cbSample, Path=IsChecked}" Value="True">
                              <Setter Property="Text" Value="Sí!" />
                              <Setter Property="Foreground"
Value="Green" />
                        </DataTrigger>
                  </Style.Triggers>
            </Style>
      </TextBlock.Style>
```

```
</TextBlock>
```

EventTrigger

Aconsejable para cuando queramos realizar una animación.

```
<TextBlock Name="lblStyled" Text="Hola mundo estilos!" FontSize="18"
HorizontalAlignment="Center" VerticalAlignment="Center">
      <TextBlock.Style>
            <Style TargetType="TextBlock">
                  <Style.Triggers>
                        <EventTrigger RoutedEvent="MouseEnter">
                              <EventTrigger.Actions>
                                    <BeginStoryboard>
                                          <Storyboard>
                                                <DoubleAnimation
Duration="0:0:0.300" Storyboard.TargetProperty="FontSize" To="28" />
                                          </Storyboard>
                                    </BeginStoryboard>
                              </EventTrigger.Actions>
                        </EventTrigger>
                        <EventTrigger RoutedEvent="MouseLeave">
                              <EventTrigger.Actions>
                                    <BeginStoryboard>
                                          <Storyboard>
                                                <DoubleAnimation
Duration="0:0:0.800" Storyboard.TargetProperty="FontSize" To="18" />
                                          </Storyboard>
                                    </BeginStoryboard>
                              </EventTrigger.Actions>
                        </EventTrigger>
                  </Style.Triggers>
            </Style>
      </TextBlock.Style>
</TextBlock>
```

MultiTrigger y MultiDataTrigger

MultiTrigger

Podemos establecer un disparador que se active cuando varias condiciones se cumplan a la vez.

CONSEJO: Usar este trigger cuando el evento se produce en el mismo control que se modifica.

```
<TextBox VerticalAlignment="Center" HorizontalAlignment="Center"
Text="Cursor y ratón en este texto" Width="160">
      <TextBox.Style>
            <Style TargetType="TextBox">
                  <Style.Triggers>
                        <MultiTrigger>
                              <MultiTrigger.Conditions>
                                    <Condition
Property="IsKeyboardFocused" Value="True" />
                                    <Condition Property="IsMouseOver"
Value="True" />
                              </MultiTrigger.Conditions>
                              <MultiTrigger.Setters>
```

```
                                    <Setter Property="Background"
Value="LightGreen" />
                              </MultiTrigger.Setters>
                        </MultiTrigger>
                  </Style.Triggers>
            </Style>
      </TextBox.Style>
</TextBox>
```

MultiDataTrigger

Podemos establecer un disparador que se active cuando varias condiciones enlazadas a datos de distintos controles se cumplan a la vez.

```
<CheckBox Name="cbSi" Content="Sí" />
<CheckBox Name="cbConfirma" Content="Confirma" />
<TextBlock HorizontalAlignment="Center" Margin="0,20,0,0"
FontSize="28">
      <TextBlock.Style>
            <Style TargetType="TextBlock">
                  <Setter Property="Text" Value="Sin verificar" />
                  <Setter Property="Foreground" Value="Red" />
                  <Style.Triggers>
                        <MultiDataTrigger>
                              <MultiDataTrigger.Conditions>
                                    <Condition Binding="{Binding
ElementName=cbSi, Path=IsChecked}" Value="True" />
                                    <Condition Binding="{Binding
ElementName=cbConfirma, Path=IsChecked}" Value="True" />
                              </MultiDataTrigger.Conditions>
                              <Setter Property="Text"
Value="Verificado" />
                              <Setter Property="Foreground"
Value="Green" />
                        </MultiDataTrigger>
                  </Style.Triggers>
            </Style>
      </TextBlock.Style>
</TextBlock>
```

Animaciones

Todos los tipos de eventos, salvo EventTrigger, pueden implementar animaciones mediante “EnterActions” y “ExitActions”.

```
<Border Background="LightGreen" Width="100" Height="100"
BorderBrush="Green">
      <Border.Style>
            <Style TargetType="Border">
                  <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value="True">
                              <Trigger.EnterActions>
                                    <BeginStoryboard>
                                          <Storyboard>
                                                <ThicknessAnimation
Duration="0:0:0.400" To="3"
Storyboard.TargetProperty="BorderThickness" />
                                                <DoubleAnimation
Duration="0:0:0.300" To="125" Storyboard.TargetProperty="Height" />
```

```
                                                <DoubleAnimation
Duration="0:0:0.300" To="125" Storyboard.TargetProperty="Width" />
                                            </Storyboard>
                                        </BeginStoryboard>
                                    </Trigger.EnterActions>
                                    <Trigger.ExitActions>
                                        <BeginStoryboard>
                                            <Storyboard>
                                                <ThicknessAnimation
Duration="0:0:0.250" To="0"
Storyboard.TargetProperty="BorderThickness" />
                                                <DoubleAnimation
Duration="0:0:0.150" To="100" Storyboard.TargetProperty="Height" />
                                                <DoubleAnimation
Duration="0:0:0.150" To="100" Storyboard.TargetProperty="Width" />
                                            </Storyboard>
                                        </BeginStoryboard>
                                    </Trigger.ExitActions>
                                </Trigger>
                            </Style.Triggers>
                        </Style>
                    </Border.Style>
                </Border>
```

Bibliografía

https://www.wpf-tutorial.com/es/398/comenzando/visual-studio-community/

https://riptutorial.com/es/wpf

https://docs.microsoft.com/es-es/visualstudio/designers/getting-started-with-wpf?view=vs-2019