ACTIVIDAD GUIADA 2
Elementos de una aplicación WPF

Contenidos

XAML básico

Crear un componente XAML es tan sencillo como escribir su nombre, entre los símbolos < y >:

```
<Button>
```

Al ser XML, las etiquetas XAML deben cerrarse, bien mediante una etiqueta de cierre o en la propia etiqueta de apertura:

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

```
<Button />
```

La mayor parte de los controles permiten añadir contenido entre las etiquetas de comienzo y fin, que representa el contenido del control. Por ejemplo, el control **`Button`** permite especificar el texto que se muestra en las etiquetas de comienzo y cierre:

```
<Button> Un botón </Button>
```

XAML es sensible a mayúsculas, pues el nombre del control debe corresponderse directamente con un tipo en el framework .NET. Lo mismo se aplica para los nombres de atributos, que se corresponden con las propiedades del control. En el siguiente botón definimos un par de propiedades añadiendo atributos a la etiqueta:

```
<Button FontWeight="Bold" Content="Un botón" />
```

Establecemos la propiedad **`FontWeight`**, que coloca el texto en negrita, y posteriormente establecemos la propiedad **`Content`**, que es lo mismo que escribir texto entre la etiqueta de comienzo y la de fin.

Otra manera de definir los atributos de un control es la siguiente, donde aparecen como etiquetas hijas del control principal, usando la notación **`Control.Propiedad`**:

```
<Button>
 <Button.FontWeight>Bold</Button.FontWeight>
 <Button.Content>Un botón</Button.Content>
</Button>
```

El resultado en ambos ejemplos es exactamente el mismo, se trata de una cuestión de sintaxis, dado que muchos controles permiten más contenido para las propiedades que únicamente texto.

Por ejemplo, el siguiente botón muestra varios colores distintos en el mismo botón añadiendo varios bloques de texto al botón.

Como la propiedad **`Content`** sólo permite un elemento hijo, usamos un contenedor **`WrapPanel`** para añadir los bloques coloreados de texto:

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

Los paneles, como **`WrapPanel`**, juegan un rol importante en WPF y se verán posteriormente en detalle. Por ahora simplemente los consideramos como contenedores de otros controles.

NOTA: El botón anterior podría haberse conseguido mediante el siguiente código, que es una manera completamente equivalente de escribir lo mismo:

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

Como ya se vio en la actividad anterior, el uso de un lenguaje de marcado facilita mucho la escritura y comprensión del código frente a un lenguaje de propósito general. El código necesario para escribir lo mismo en C# es el siguiente:

```
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 = "Button";
pnl.Children.Add(txt);

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

Eventos en XAML

WPF, al igual que la mayor parte de los frameworks para interfaz de usuario, está **dirigido por eventos.** Todos los controles, igual que **`Window`** (que también hereda de la clase **`Control`**) proporciona un rango de eventos al cual nos podemos **suscribir**. Al suscribirnos a un evento, nuestra aplicación será notificada cada vez que ocurre el evento y podemos reaccionar como queramos.

Hay muchos tipos de eventos, si bien lo más común es responder a la interacción del usuario con nuestra aplicación a través del ratón o el teclado. En la mayoría de controles encontraremos eventos como **`KeyDown`**, **`KeyUp`**, **`MouseDown`**, **`MouseEnter`**, **`MouseLeave`**, **`MouseUp`** y otros.

Vamos a ver como enlazamos un evento en XAML a una pieza de código en el archivo *code-behind.* Pongamos por ejemplo el siguiente archivo:

```
<Window x:Class="HolaWPF.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Ejemplo de evento" Height="100" Width="525">
    <Grid Name="pnlGridPrincipal" MouseUp="pnlGridPrincipal_MouseUp"
          Background="LightBlue">
    </Grid>
</Window>
```

Como se puede ver, usamos el atributo **`Name`** para asignar un nombre al control. Hemos seguido, al igual que hacíamos en Java, la notación húngara para referirnos a los nombres de los componentes, anteponiendo un prefijo a cada variable que indica el tipo de componente que representa la variable.

En este caso hemos suscrito el evento **`MouseUp`**[1] al **`Grid`** escribiendo el nombre del método que se va a ejecutar. Este método se llama **manejador del evento**. El nombre del manejador podemos establecerlo arbitrariamente o seguir la convención **`Nombrecomponente_NombreEvento`** (si pulsamos Control+Espacio para autocompletar al escribir el valor del atributo se escribirá de manera automática).

El manejador recibe 2 parámetros:

- El objeto que desencadenó el evento (tipo **`object`**)
- Un objeto con información relevante acerca del evento que ha ocurrido (en este caso de tipo **`MouseButtonEventArgs.`**

Si vamos al archivo de *Code-Behind* (**`MainWindow.xaml.cs`**) comprobaremos que el manejador se ha generado automáticamente:

```
private void pnlGridPrincipal_MouseUp(object sender, MouseButtonEventArgs e)
        {

        }
```

[1] A diferencia de lo que ocurre en otros lenguajes de programación, la especificación del evento no va antecedida del prefijo **On**. Es decir, el evento es **`MouseUp`** en lugar de **`OnMouseUp`**

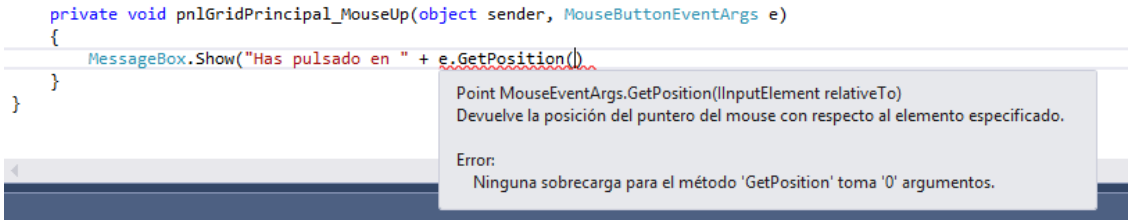
Vamos a añadir algo de código. En este caso, simplemente mostraremos un mensaje indicativo de las coordenadas en las que hemos pulsado:

```
private void pnlGridPrincipal_MouseUp(object sender, MouseButtonEventArgs e)
{
        MessageBox.Show("Has pulsado en " + e.GetPosition(this).ToString());
}
```

Como pautas generales:

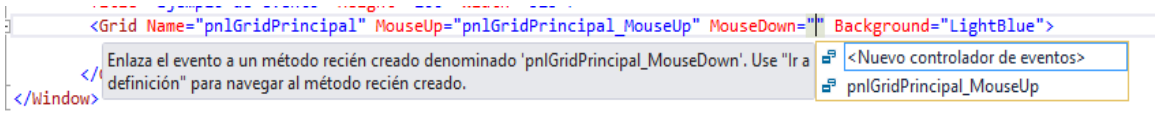
- Los métodos en .NET llevan todas sus palabras en mayúsculas. Es decir, lo que en Java era **`toString()`** en .NET se corresponde con **`ToString()`**. El uso de la autocomplección (Control+Espacio) ayuda a evitar errores al escribir.
- El método **`GetPosition()`** del objeto permite obtener las coordenadas donde hemos pulsado. El método recibe un objeto a partir del cual se calculan las coordenadas (en este caso le pasamos la ventana con `this`).

Visual Studio cuenta con una ayuda contextual muy completa. Simplemente tenemos que colocar el ratón sobre el elemento y se mostrará información sobre el método en que nos encontramos.



Varios eventos pueden tener el mismo tipo de delegado. Por ejemplo, tanto **`MouseUp`** como **`MouseDown`** usan el delegado **`MouseButtonEventHandler`**. Cuando definimos el método manejador de evento, necesitamos saber qué delegado usa.

Visual Studio nos ayuda a generar el método adecuado para un evento. El método más simple es escribir el nombre del evento en el XAML y comprobar las opciones que nos da la ayuda:



Si pulsamos **<Nuevo controlador de eventos>** Visual Studio genera el manejador de eventos correspondiente en el archivo *Code-Behind* con la notación descrita anteriormente (de hecho fue lo que hicimos cuando autocompletamos el nombre). También podemos escoger que un manejador ya escrito maneje el evento. **Sólo se nos mostrarán los manejadores de tipos compatibles**.

Suscripción a un evento desde el *Code-Behind*

El método más adecuado para suscribirnos a un evento es el que acabamos de ver. Sin embargo, en otros casos podemos necesitar suscribirnos a un manejador de eventos directamente desde el código *Code-Behind*. Esto se hace a través de la **sintaxis += de C#**, donde se añade un manejador de eventos al evento directamente en el objeto:

```
namespace HolaWPF
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            pnlGridPrincipal.MouseUp +=
               new MouseButtonEventHandler(pnlGridPrincipal_MouseUp);
        }

        private void pnlGridPrincipal_MouseUp(object sender, MouseButtonEventArgs e)
        {
            MessageBox.Show("Has pulsado en " + e.GetPosition(this).ToString());
        }
    }
}
```

El objeto Window

Cuando creamos una aplicación WPF, el primer objeto que nos encontramos es de clase **`Window`**. Permite visualizar una ventana y proporciona el borde estándar, la barra de título y los botones de maximizar, minimizar y cerrar. El código XAML asociado a la misma es el siguiente:

```
<Window x:Class="HolaWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Ejemplo de evento" Height="300" Width="300">
        <Grid>
       </Grid>
</Window>
```

El atributo **`x:Class`** indica al archivo XAML que archivo de clase vamos a usar. En este caso se trata del archivo **`MainWindow`**, cuyo namespace es **`HolaWPF`** (el nombre que dimos al crear el proyecto). Como hemos visto, el archivo *code-behind* generado es el siguiente:

```
namespace HolaWPF
{
    /// <summary>
    /// Lógica de interacción para MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

La clase **`MainWindow`** es definida como parcial, pues combinada con el archivo XAML en tiempo de ejecución da como resultado la ventana completa. Es lo que hace realmente la llamada a **`InitializeComponent()`**, por tanto la llamada a dicho método es vital para tener la ventana funcional.

Volviendo al archivo XAML, vemos los atributos correspondientes al título de la ventana (que se muestra en la barra de título) así como anchura y altura. También hay un par de definiciones de espacios de nombres, que veremos más adelante.

En cuanto al **Grid**, es uno de los paneles de WPF. Si bien podría tratarse de cualquier panel, el objeto **Window sólo puede tener un control hijo**. Dado que los paneles pueden tener varios hijos, son una buena opción como contenedor de primer nivel. Más adelante también veremos los distintos tipos de paneles.

Algunas propiedades de Window

A continuación se muestran algunos de los atributos principales de `Window`.

- **Icon**. Permite definir el icono de la ventana, mostrado en la esquina superior izquierda.
- **ResizeMode**. Controla como el usuario puede redimensionar la ventana:
  - **CanResize**. Valor por defecto. El usuario puede redimensionar la ventana a través de los botones de maximizar/minimizar o mediante los bordes.
  - **CanMinimize**. Permite minimizar la ventana pero no maximizarla ni redimensionarla con los bordes.
  - **NoResize**. Es el más estricto, donde los botones de maximizar y minimizar son eliminados y la ventana no puede redimensionarse.
- **ShowInTaskBar.** Por defecto es **true**. Si la establecemos a **false**, la ventana no se representará en la barra de tareas de Windows. Esto es útil para aplicaciones de Windows no primarias o para aplicaciones que al minimizarse vayan al área de notificación.
- **SizeToContent.** Decide si la ventana debería redimensionarse automáticamente para albergar contenido. Por defecto es **Manual**, lo que significa que la ventana no se redimensiona automáticamente. Otras opciones son **Width**, **Height** y **WidthAndHeight**, y cada una de ellas ajustan automáticamente el tamaño verticalmente, horizontalmente o ambos.
- **Topmost.** Por defecto es **false**, pero si lo establecemos a **true**, la ventana estará siempre por encima del resto de ventanas (salvo que esté minimizada).
- **WindowStartupLocation.** Controla la posición inicial de la ventana. Por defecto es **Manual**, lo que significa que la ventana estará inicialmente posicionada de acuerdo a las propiedades **Top** y **Left**. Las otras opciones son **CenterOwner**, que posiciona la ventana en el centro de la ventana propietaria, y **CenterScreen**, que posiciona la ventana en el centro de la pantalla.
- **WindowState.** Controla el estado inicial de la ventana. Puede ser **Normal**, **Maximized** o **Minimized**. Por defecto es **Normal**, que es lo que deberíamos usar salvo que queramos que la ventana este de comienzo maximizada o minimizada.

Hay otras propiedades que se pueden explorar desde el inspector de propiedades del componente.

Trabajando con App.xaml

App.xaml es el punto de comienzo de la aplicación. Visual Studio lo crea automáticamente cuando comenzamos una aplicación WPF, así como un archivo *Code-Behind* llamado **App.xaml.cs**. Trabajan de manera colaborativa al igual que en una ventana. **App.xaml** extiende la clase **Application**, que es una clase central en una aplicación WPF. .NET busca en esta clase las instrucciones de comienzo y luego lanza la ventana (**Window**) correspondiente.

Una de las características más usadas del archivo **App.xaml** es definir recursos globales que pueden ser usados desde toda la aplicación, como por ejemplo estilos globales.

Estructura de App.xaml

A crear una nueva aplicación, el **App.xaml** generado es similar al siguiente:

```
<Application x:Class="HolaWPF.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

El principal atributo es **StartupUri**, que determina la ventana o página a lanzar. En este caso se apunta a **MainWindow.xaml**, pero si queremos usar otra ventana como punto de comienzo, cambiamos esta propiedad.

En ocasiones queremos más control sobre cómo y cuando se muestra la primera ventana. En este caso, eliminamos la propiedad y el valor de **StartupUri** y realizamos la asignación desde el archivo *Code Behind* como se muestra a continuación.

Estructura de App.xaml.cs

En un nuevo proyecto, el archivo **App.xaml.cs** tendrá el siguiente aspecto:

```
namespace HolaWPF
{
    public partial class App : Application
    {
    }
}
```

Como vemos, la clase extiende la clase base **Application**, que permite realizar ciertas tareas a nivel de aplicación. Por ejemplo, podemos suscribirnos al evento **Startup**, donde podemos crear de manera manual la ventana. Veámoslo con un ejemplo:

```
<Application x:Class="HolaWPF.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Startup="Application_Startup">
    <Application.Resources>
    </Application.Resources>
</Application>
```

Como se puede ver, **StartupUri** ha sido reemplazado con una suscripción al evento **Startup**. En el *Code-Behind*, implementamos el evento como queramos:

```
namespace HolaWPF
{
    public partial class App : Application
    {
    private void Application_Startup(object sender, StartupEventArgs e)
        {
            MainWindow ventana = new MainWindow();
            ventana.Title = "Nuevo título";
            ventana.Show();
        }
    }
}
```

La ventaja de este enfoque frente a la utilización de la propiedad **StartupUri** es que podemos manipular la ventana antes de mostrarla. En este caso simplemente cambiamos el título, lo cual no es demasiado útil, pero también podemos realizar otras acciones como mostrar una pantalla de presentación (*Splash Screeen*) o suscripción a nuevos eventos.

Recursos

WPF introduce un concepto muy útil: La posibilidad de guardar datos como recursos, ya sea de manera local para un control, de manera local para una ventana o de manera global para toda la aplicación. Los datos pueden ser lo que queramos, desde información real a una jerarquía de controles WPF. Esto nos permite guardar los datos en un lugar y acceder a ellos desde uno o varios lugares, lo que es muy útil.

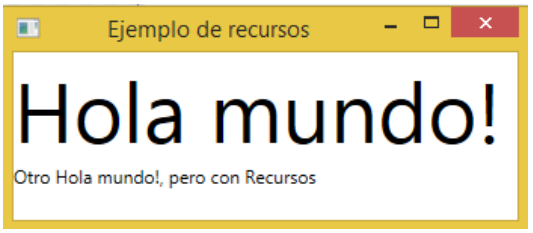
Este concepto es muy usado para estilos y plantillas, lo que se verá más adelante, pero pueden usarse para otro tipo de elementos, como se muestra en el siguiente ejemplo:

```
<Window x:Class="HolaWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"

        Title="Ejemplo de recursos" Height="150" Width="350">
    <Window.Resources>
        <sys:String x:Key="strHolaMundo">Hola mundo!</sys:String>
    </Window.Resources>

        <StackPanel>
            <TextBlock Text="{StaticResource strHolaMundo}" FontSize="56" />
        <TextBlock>Otro <TextBlock Text="{StaticResource strHolaMundo}" />, pero con
Recursos</TextBlock>
    </StackPanel>
</Window>
```

Los recursos se identifican mediante una clave, usando el atributo **x:Key** que permite referenciarlos desde otra parte de la aplicación usando dicha clave, en combinación con la extensión de marcado **StaticResource**. En este ejemplo simplemente guardamos una cadena, que posteriormente usamos dentro de dos bloques de texto. A continuación se muestra el resultado:



Recursos estáticos vs Recursos dinámicos

En el ejemplo anterior hemos utilizado la extensión de marcado **`StaticResource`**. Otra alternativa hubiera sido usar la extensión **`DinamicResource`**.

La principal diferencia es que un recurso estático (**`StaticResource`**) se evalúa únicamente en un punto (cuando el archivo XAML se carga). Si el recurso se cambia posteriormente, el cambio no se reflejará en el punto donde los usamos.

Por otra parte un recurso dinámico (**`DinamicResource`**) se resuelve cada vez que se necesita, y luego de nuevo cada vez que el recurso cambia.

Excepciones

El concepto de excepción no es nuevo si estamos familiarizados con los lenguajes orientados a objetos: Cuando tenemos una pieza de código susceptible de generar un error deberíamos agruparlo en un bloque **`try-catch`** para manejar la excepción correctamente. Por ejemplo si llamamos al método **`Trim()`** de una variable nula se producirá un error. Si no tratamos la excepción, la aplicación se colgará y Windows deberá manejar el programa, lo que habitualmente no es muy amigable para el usuario. Por eso podemos evitar este tipo de comportamientos usando un bloque **`try-catch`** como el siguiente:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
          try {
                string s = null;
                s.Trim();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Se produjo la excepción " + ex.Message);
            }
}
```

Sin embargo, en ocasiones puede ser engorroso englobar todo el código susceptible de generar excepciones en bloques **`try-catch`**, dado que el código que genera excepciones puede encontrarse en muy diversas partes de la aplicación. Por tanto, WPF permite manejar todas las excepciones no tratadas de manera global. Esto puede lograrse a través del evento **`DispatcherUnhandledException`** de la clase **`Application`**. Si nos suscribimos al mismo, WPF llamará al mismo cada vez que se lanza una excepción que no recogemos en el código.

Veamos un ejemplo que prueba dicho comportamiento:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
          string s = null;
          try {
              s.Trim();
            }
            catch (Exception ex)
            {
               MessageBox.Show("Se produjo la excepción " + ex.Message,
                               "Ejemplo de excepción", MessageBoxButton.OK,
                                MessageBoxImage.Exclamation);
            }
          s.Trim();
}
```

Llamamos al **`Trim()`** otra vez fuera del bloque **`try-catch`**, de modo que la primera llamada al método es atrapada pero la segunda no. Para atrapar dicha llamada vamos a usar **`App.xaml`**:

```
<Application x:Class="HolaWPF.App"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       DispatcherUnhandledException="Application_DispatcherUnhandledException"
       StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Recogemos ambas excepciones mostrándolas con un cuadro de diálogo **`MesageBox`**, cambiando el icono que la acompaña. Nótese también que establecemos la propiedad **`e.Handled`** a **`true`** para indicar que se ha tratado la excepción.

```
namespace HolaWPF
{
    public partial class App : Application
    {
private void Application_DispatcherUnhandledException(object sender,
System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e)
      {
      MessageBox.Show("Se produjo la excepción (no tratada) " + e.Exception.Message,
         "Ejemplo de excepción", MessageBoxButton.OK, MessageBoxImage.Information);
      e.Handled = true;
      }
    }
}
```