Eventos en XAML

1. ¿Qué es un evento?

Un evento es una notificación que indica que algo ha sucedido en un elemento de la interfaz de usuario. Pueden ser desencadenados por acciones del usuario (como hacer clic en un botón) o por cambios en el estado del sistema (como la finalización de una operación asíncrona).

Pasos para asignar un evento a un componente en XAML

1. Asociar eventos a elementos de la interfaz de usuario utilizando atributos como "**Click**" para un botón, "**TextChanged**" para un cuadro de texto, etc.
   Por ejemplo, si tenemos un botón en XAML:

```
<Button Content="Haz clic" Click="MiBoton_Click" />
```

2. Crear un método en C# con el mismo nombre que hemos declarado en XAML, es decir, que coincida con la firma del evento "MiBoton_Click":

```
private void MiBoton_Click(object sender, RoutedEventArgs e)
{
    // Código para manejar el evento del botón aquí
}
```

A partir de aquí, habría que rellenar el método para aplicarle la lógica deseada al componente que invoca el método.

Respecto a los parámetros que reciben los eventos:

- *Sender:* es el objeto que desencadenó el evento. Representa al elemento que generó el evento y lo pasó a través de la cadena de eventos.

  Es importante tener en cuenta que **sender** es de tipo **object**, por lo que necesitarás convertirlo al tipo de objeto correcto si deseas acceder a sus propiedades o métodos específicos.

  Usualmente, **sender** se utiliza para identificar qué elemento generó el evento, lo que puede ser útil si estás manejando múltiples elementos con un solo manejador de eventos.

  Ejemplo:

```
private void MiBoton_Click(object sender, RoutedEventArgs e)
{
    Button boton = (Button)sender; // Convierte el sender a tipo Button
    string contenido = boton.Content.ToString(); // Accede al contenido del botón
}
```

- *e:* es un objeto que contiene información adicional sobre el evento que está ocurriendo. El tipo específico de EventArgs depende del tipo de evento que se está manejando.
  Por ejemplo, si estás manejando un evento de click, e podría ser de tipo RoutedEventArgs, que proporciona información sobre el evento de clic, como la tecla que se presionó para activar el evento.

e no siempre se utiliza en todos los manejadores de eventos, pero puede ser útil cuando necesitas información adicional sobre el evento.

Ejemplo:

```
private void MiBoton_Click(object sender, RoutedEventArgs e)
{
    if (e is RoutedEventArgs routedEventArgs)
    {
        // Accede a propiedades específicas de RoutedEventArgs
        // por ejemplo: routedEventArgs.Source, routedEventArgs.Handled, etc.
    }
}
```

Nota: sender y **e** son simplemente convenciones de nomenclatura comunes en la programación de eventos en .NET. Puedes nombrar estos parámetros de la forma que prefieras, pero usar **sender** y **e** es una práctica estándar que hace que tu código sea más legible para otros desarrolladores.

2. Eventos enrutados (*routed events*)

Según la documentación del SDK de Windows, un evento enrutado es una instancia de un evento que se propaga por un árbol de elementos relacionados en lugar de apuntar a un solo elemento. Según esa misma documentación, hay tres tipos de enrutamiento: *bubbling* (de burbuja), *tunneling* (de túnel) y *directo*.

Con el enrutamiento *bubbling*, el evento se propaga ("sube") desde el elemento que lo produce hasta la parte superior del árbol de elementos.

Por otra parte, en el caso del enrutamiento *tunneling* esa propagación "baja" desde la parte superior del árbol hasta el elemento que produce el evento.

Por último, el enrutamiento *directo* no propaga el evento en ninguna dirección y se comporta como los eventos "normales" a los que estamos acostumbrados.

Para comprender mejor estas nuevas definiciones, debemos tener en cuenta cómo funcionan o se comportan los elementos de una aplicación WPF, que como sabemos, en este tipo de aplicaciones están definidos en XAML. Por regla general, los elementos de una aplicación XAML suelen estar contenidos en otros elementos; de hecho, al igual que ocurre con las aplicaciones normales de Windows, siempre hay algún contenedor que "contiene" los controles, aunque sea el propio formulario, en el caso de las aplicaciones de Windows, o cualquiera de los contenedores de las aplicaciones de WPF.

2.1. Los tipos de enrutamiento de los eventos XAML

En WPF cada evento se puede detectar de dos formas diferentes, según lo queramos interceptar antes de llegar al elemento que realmente lo produce (*tunneling*). En cuyo caso el nombre del evento va precedido de **Preview**, lo que nos da una idea de cuál es la intención de dicho evento: tener la posibilidad de interceptarlo antes de que realmente se produzca. Por otro lado, tenemos los eventos *bubbling*, a los que no se añade ningún prefijo. Como regla general, todos los eventos de los elementos de WPF van en pareja, y por cada evento suele haber uno previo y uno normal.

Por ejemplo, la pulsación de las teclas suele detectarse con el evento ***KeyDown***, evento de tipo *bubbling* (lo detectaríamos en el control que lo produce) y el correspondiente de tipo *tunneling* es ***PreviewKeyDown*** (para detectarlo antes de que llegue al control).

Hay ciertos eventos que no van en parejas, aunque sí que suelen estar relacionados con otros eventos; por ejemplo, el evento ***Click*** de un botón sería del tipo *bubbling*, aunque no tiene emparejado el equivalente al enrutamiento *tunneling* (no existe un evento ***PreviewClick***). En principio, podría parecer que ese evento es de tipo directo, ya que solo se produce en el control que lo define y, aparentemente, no tiene equivalente previo. Aunque en el caso de los eventos relacionados con el ratón, siempre hay formas de buscar los equivalentes previos; por ejemplo, el evento ***Click*** va precedido de varios eventos, entre ellos los que detectan la pulsación del botón izquierdo del ratón: ***MouseLeft-ButtonDown*** y ***PreviewMouseLeftButton-Down***, aunque en este caso en particular del evento **Click**, el propio control que detecta la pulsación marca el evento como manipulado (***Handled***), impidiendo que se propague por el árbol de contenedores. Pero tal como XAML nos permite definir los eventos enrutados, también podemos indicar que se intercepte el evento ***Click*** de un botón en el contenedor (o padre) de ese botón; además, de forma independiente al evento interceptado por el propio botón.

2.1.1. Enrutamiento *bubbling*(burbuja)

La mejor forma de entender estos conceptos es viéndolos con un ejemplo. Para mantener las cosas simples, el código XAML de ejemplo está muy simplificado (pero 100% operativo) y consiste en una ventana (***Window***) que contiene un ***StackPanel*** que a su vez contiene dos botones y dos etiquetas; en el ***StackPanel*** definimos un evento que interceptará la pulsación en cualquiera de los botones que contenga, además, de forma independiente, cada botón define su propio evento ***Click***. Para saber cuál es el orden en el que se producen los eventos tengo definida una función que simplemente incrementa una variable y devuelve el valor de esta.

```
<StackPanel Name="stackPanel1" Button.Click="stackPanel_ButtonClick">
    <Button Name="btnUno" Content="Uno" Click="btnUno_Click" />
    <Button Name="btnDos" Content="Dos" Click="btnDos_Click" />
    <Label Name="labelInfo" Content="Info" />
    <Label Name="labelInfo2" Content="Info2" Background="LightBlue" />
</StackPanel>
```

Ilustración 1. Fuente 1. Código XAML primer ejemplo

```
1 reference
private void stackPanel_ButtonClick(object sender, RoutedEventArgs e)
{
    Button btn = (Button)e.OriginalSource;
    labelInfo.Content = "Has pulsado en " +btn.Name + " " + total();
}
1 reference
private void btnUno_Click(object sender, RoutedEventArgs e)
{
    labelInfo2.Content = "Botón Uno " + total();
}
1 reference
private void btnDos_Click(object sender, RoutedEventArgs e)
{
    labelInfo2.Content = "Botón Dos " + total();
}
```

Ilustración 2. Fuente 2. Código de C# del primer ejemplo

En las Ilustraciones 1 y 2 tenemos tanto la definición del código XAML como el correspondiente al uso desde C# para interceptar esos eventos.

El evento ***Click*** es de tipo *bubbling*, por tanto, primero se produce en el elemento que provoca el evento y después se propaga al resto de elementos que están en el mismo árbol, es decir, al contenedor del control y al contenedor del contenedor, etc.; en nuestro caso al ***StackPanel*** y, si así lo hubiéramos previsto, al objeto ***Window***. Por tanto, tal como está el código de la ilustración 2, primero se mostrará el mensaje en la etiqueta ***labelInfo2*** y después en ***labelInfo***.

En el elemento ***StackPanel*** indicamos que queremos interceptar el evento ***Click*** de los botones que contenga este control y en el código del método que intercepta ese evento mostramos el nombre del control que lo produce. En ese código usamos la propiedad ***OriginalSource*** del objeto recibido en el segundo parámetro, aunque en este caso particular también nos hubiera valido usar el valor devuelto por la propiedad ***Source***. Como hemos comentado, el evento interceptado por ese método se producirá después de que se haya producido el evento en cada uno de los botones. En cualquier caso, si dentro de los métodos que interceptan los eventos particulares de cada botón quisiéramos dejar de "reproducir" el evento, es decir, evitar que se propague a los contenedores del botón, podemos asignar un valor verdadero a la propiedad ***Handled*** del segundo parámetro.

```
<Window x:Class="Ejemplo2.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:Ejemplo2"
        mc:Ignorable="d"
        Title="Bubbling02" Height="450" Width="800"
        Button.Click="Window1_ButtonClick">
```

Ilustración 3. Modificación Fuente 1 para interceptar en el objeto Window el evento Click de los botones

```
private void Window1_ButtonClick(object sender, RoutedEventArgs e)
{
    Button btn = (Button)e.OriginalSource;
    labelInfo.Content += "\n(Window)Has pulsado en " +btn.Name + " " + total();
    labelInfo.Content += '\n' +((FrameworkElement)e.Source).ToString();
}
1 reference
private void stackPanel_ButtonClick(object sender, RoutedEventArgs e)
{
    Button btn = (Button)e.OriginalSource;
    labelInfo.Content = "(StackPanel)Has pulsado en " +btn.Name + " " + total();
    labelInfo.Content += '\n' +((FrameworkElement)e.Source).ToString();
}
// Al indicar e.Handled = true el evento no se propagará.
1 reference
private void btnUno_Click(object sender, RoutedEventArgs e)
{
    labelInfo2.Content = "Botón Uno " + total();
    labelInfo.Content = "";
    e.Handled = true;
}
1 reference
private void btnDos_Click(object sender, RoutedEventArgs e)
{
    labelInfo2.Content = "Botón Dos " + total();
}
```

Ilustración 4. Cambios a Fuente 2 para utilizarlo junto con el código de la ilustración 3

En el código de las ilustraciones 3 y 4 tenemos los cambios a realizar en el ejemplo anterior para propagar el evento ***Click*** de los botones al objeto ***Window***, y para marcar como manejado dicho evento, pudiendo comprobar todo lo aquí comentado.

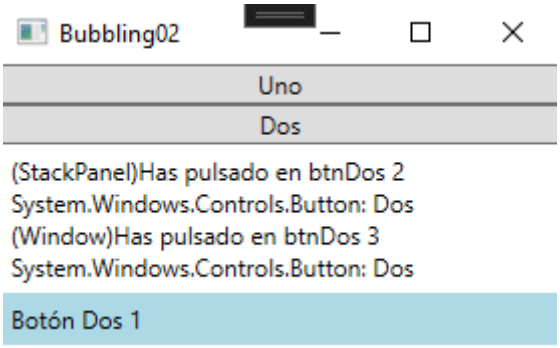

Ilustración 5. Segundo ejemplo en ejecución tras pulsar en el botón Dos

En la ilustración 5 vemos la aplicación en ejecución después de haber pulsado en el segundo botón, en la que podemos apreciar el orden en el que se ejecuta el código.

2.1.2. Enrutamiento tunneling

La otra forma de "enrutar" los eventos en XAML es la técnica conocida como *tunneling*, que consiste en la propagación de los eventos desde el contenedor de nivel más superior hasta el elemento que en realidad produce el evento. Los eventos clasificados en el tipo *tunneling* contienen en el nombre el prefijo ***Preview***. Por ejemplo, si queremos detectar el movimiento del ratón en una serie de controles de una aplicación de tipo WPF, podríamos escribir un código como el mostrado en la ilustración 6.

```xaml
<StackPanel Name="stackPanel1" PreviewMouseMove="stackPanel_MouseMove">
    <Button Name="btnUno" Content="Uno" MouseMove="btnUno_MouseMove" />
    <Button Name="btnDos" Content="Dos" MouseMove="btnDos_MouseMove" />
    <Label Name="labelInfo" Content="Info" />
    <Label Name="labelInfo2" Content="Info2" Background="LightBlue" />
</StackPanel>
```

Ilustración 6. Código XAML del ejemplo de tunneling

En el control ***StackPanel*** definimos el evento ***PreviewMouseMove***, además en cada uno de los dos botones que contiene ese panel hay definido también un evento ***MouseMove***. El primero se producirá antes que cualquiera de los otros dos, al menos cuando el ratón se mueva por cualquiera de esos dos botones. Pero al estar definido a nivel del contenedor, ese evento será capaz de detectar el movimiento del ratón en cualquiera de los controles que contiene además de en sí mismo.

```csharp
private void stackPanel_MouseMove(object sender, RoutedEventArgs e)
{
    FrameworkElement elem = (FrameworkElement)e.Source;
    labelInfo.Content = "El ratón está en " +elem.Name + " " + total();
    // OriginalSource y Source no tienen porqué ser lo mismo
    labelInfo.Content += "\nSource = " +e.Source.ToString();
    labelInfo.Content += "\nOriginalSource = " + e.OriginalSource.ToString();
}
1 reference
private void btnUno_MouseMove(object sender, MouseEventArgs e)
{
    labelInfo2.Content = "El ratón está en el botón Uno " + total();
}
1 reference
private void btnDos_MouseMove(object sender, MouseEventArgs e)
{
    labelInfo2.Content = "El ratón está en el botón Dos " + total();
}
```

Ilustración 7. Código en C# del ejemplo de tunneling

Con el código de la ilustración 7 podemos saber en qué control está el ratón en cada momento y gracias al valor devuelto por la función ***total()*** también sabremos en qué orden se ejecutan los eventos, al menos si ese movimiento de ratón lo hacemos sobre cualquiera de los dos botones, ya que si movemos el ratón encima de cualquiera de las dos etiquetas, también se producirá (e interceptará) el evento a nivel del panel.

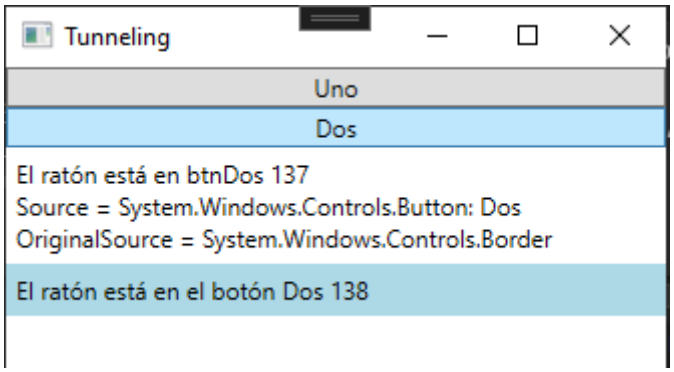

Ilustración 8. Resultado de ejecutar el código de la ilustración 7

Como ya sabemos, cada control de las aplicaciones XAML en realidad está formado por otros controles. Por ejemplo, las etiquetas (o los botones) para mostrar el texto en realidad utilizan un control del tipo ***TextBlock***, y si vemos el resultado mostrado en la ilustración 8, nos daremos cuenta de que el control que se recibe en la propiedad ***OriginalSource*** del segundo parámetro del método ***stackPanel_MouseMove*** en realidad es un control ***Border***, mientras que el control indicado por la propiedad **Source** hace referencia al propio botón en el que se está efectuando el movimiento del ratón.