Jesús García | http://jgarcia.dev
Versión 21.1.16

Unidad 2
Interfaces de usuario con Xamarin.Forms

Índice

Introducción a Xamarin.Forms

Xamarin es una plataforma que permite desarrollar en C# mediante una serie de librerías (*Xamarin.Android*, *Xamarin.iOS*), aplicaciones que funcionan tanto en *iOS* como en *android*. Las aplicaciones desarrolladas con *Xamarin* siguen siendo nativas (no son interpretadas).

Xamarin incluye *Xamarin.Forms*, una librería que nos permite desarrollar una única interfaz común para las diferentes plataformas. Podemos desarrollarla utilizando tres herramientas:

1. Lenguaje *XAML*: estudiaremos más adelante en qué consiste.
2. Lenguaje *C#*: mediante la instanciación de los elementos de la interfaz de usuario.
3. Mediante el editor gráfico (WYSIWYG) de Visual Studio.

Anatomía de proyecto de Xamarin

Una vez creemos una solución en Visual Studio nos encontraremos con que está compuesta por tres proyectos:

> *NombreProyecto*: Contienen los archivos XAML necesarios para definir la interfaz de usuario así como el código compartido entre todas las plataformas.
>
> *NombreProyecto*.Android: Contiene el código y los recursos específicos para la plataforma Android.
>
> *NombreProyecto*.iOS: Contiene el código y los recursos específicos para la plataforma iOS.

Introducción a XAML

Se trata de un lenguaje de marcas declarativo basado en XML que permite inicializar objetos de la interfaz de usuario y establecer sus propiedades. Cada archivo XAML va asociado a un archivo con código en C# en el que podemos responder a eventos y manipular los objetos originalmente declarados en XAML. XAML no es exclusivo de Xamarin, se utiliza con otras tecnologías como WPF para el desarrollo de aplicaciones de escritorio Windows.

En un proyecto XAML se trabajan con pares de archivos cuya extensión son:

.xaml: el archivo XAML propiamente dicho.

.xaml.cs: el código C# asociado al archivo XAML.

Con XAML podemos definir la interfaz de usuario de nuestra aplicación, esto también lo podríamos hacer directamente en código en C# sin utilizar XAML. En clase veremos los dos métodos.

Atributos y elementos

En XAML los elementos se corresponden con clases mientras que los atributos se corresponden con sus propiedades. Por ejemplo:

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large"
       TextColor="Aqua" />
```

- *Label*: Es un objeto de la clase *Label* de *Xamarin.Forms* expresado como un elemento xml.
- *Text*, *VerticalOptions*, *FontAttributes*, *FontSize*, *TextColor* son propiedades del objeto *Label* expresado como atributos del elemento.

A veces la propiedad de un objeto puede ser a su vez otro objeto con múltiples propiedades, esto hace que para especificar los valores de sus propiedades no sea suficiente con la sintaxis propiedad-atributo. Por eso XAML dispone de una sintaxis alternativa para definir los valores de las propiedades mediante el uso de elementos (propiedad-elemento):

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large">
    <Label.TextColor>
        Aqua
    </Label.TextColor>
</Label>
```

Otro ejemplo sería la clase *Grid*, que tiene dos propiedades llamadas *RowDefinitions* y *ColumnDefinitions*. Cada una de estas propiedades a su vez son una colección de *RowDefinition* y *ColumnDefinition* respectivamente. En este caso, para definir los elementos de las colecciones es necesario utilizar la sintaxis propiedad-elemento.

```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
```

```
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="100" />
        </Grid.ColumnDefinitions>
</Grid>
```

Contenidos de los elementos

Un elemento puede contener otros elementos a través de su propiedad *Content* o *Children.* No es obligatorio expresar estas propiedades en XAML.

Por ejemplo:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.XamlPlusCodePage"
             Title="XAML + Code Page">

        <StackLayout>
                <Slider VerticalOptions="CenterAndExpand"
                        ValueChanged="OnSliderValueChanged" />
                <Label x:Name="valueLabel"
                       Text="A simple Label"
                       FontSize="Large"
                       HorizontalOptions="Center"
                       VerticalOptions="CenterAndExpand" />
                <Button Text="Click Me!"
                      HorizontalOptions="Center"
                      VerticalOptions="CenterAndExpand"
                      Clicked="OnButtonClicked" />
        </StackLayout>
</ContentPage>
```

En este caso *ContentPage* tiene una propiedad *Content* que le permite tener un contenido (en este caso el *StackLayout*). A su vez, el *StackLayout* tiene una propiedad *Children* que es una colección de todos los objetos que puede contener (en este caso un *Slider*, *Label* y *Button*). No es obligatorio indicar la propiedad *Content* o *Children*, pero si lo quisiéramos hacer lo podríamos expresar de la siguiente forma:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.XamlPlusCodePage"
```

```
            Title="XAML + Code Page">
    <ContentPage.Content>
        <StackLayout>
            <StackLayout.Children>
                <Slider VerticalOptions="CenterAndExpand"
                        ValueChanged="OnSliderValueChanged" />

                <Label x:Name="valueLabel"
                       Text="A simple Label"
                       FontSize="Large"
                       HorizontalOptions="Center"
                       VerticalOptions="CenterAndExpand" />

                <Button Text="Click Me!"
                      HorizontalOptions="Center"
                      VerticalOptions="CenterAndExpand"
                      Clicked="OnButtonClicked" />
            </StackLayout.Children>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Espacio de nombres

Los espacios de nombres se utilizan en XML para organizar conceptos, determinar el significado de cada uno de los elementos y atributos; y evitar ambigüedades.

Hay una serie de espacios de nombres que encontraremos en el elemento raíz de los documentos XAML.

```
xmlns="http://xamarin.com/schemas/2014/forms"
```

Es el espacio de nombres por defecto y hace referencia a las clases de *Xamarin.Forms* como por ejemplo *ContentPage*.

```
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

Espacio de nombres que utiliza el prefijo *x*. Son elementos y atributos intrínsecos de XAML. Los atributos que podemos utilizar dentro de este espacio de nombres son:

x:Class

Permite enlazar un archivo XAML con una clase. Se indica el espacio de nombres y la clase que conectaremos con el XAML siguiendo el formato *namespace.classname*. Sólo puede aparecer en el elemento raíz de un archivo XAML.

x:Arguments

Se utiliza cuando es necesario instanciar objetos con constructores que requieren parámetros. Por ejemplo, el objeto *Color* tiene un constructor que recibe como parámetros los componentes RGB como *Double*.

```
<Color>
      <x:Arguments>
        <x:Double>0.25</x:Double>
        <x:Double>0.5</x:Double>
        <x:Double>0.75</x:Double>
      </x:Arguments>
</Color>
```

Algunos de los tipos que podemos utilizar son: *x:Object*, *x:Boolean, x:Byte, x:Int16, x:Int32, x:Int64, x:Single, x:Double, x:Decimal, x:Char, x:String, x:TimeSpan, x:Array, x:DateTime*. También podríamos pasar instancias de una clase añadiendo el elemento correspondiente en el contenido de la etiqueta *x:Arguments*.

x:Name

Identifica un elemento. Asignar un x:Name es similar a declarar una variable en el código. El valor que asignemos al atributo será el nombre que tendrá la variable que representa al elemento en el código C#.

Eventos

XAML es un lenguaje declarativo de objetos y sus propiedades, pero también permite incluir manejadores de eventos. Para ello se especifica el evento como un atributo y se le asigna un método de la clase C# asociada.

Por ejemplo, el elemento *Button* dispone de un evento *Clicked*, ya sea mediante XAML o C# podemos indicar qué método queremos que se ejecute cuando se produzca dicho evento.

Ejemplo XAML:

En el archivo *.xaml*:

```
<Button Text="Haz clic aquí!"
                Clicked="OnButtonClicked" />
```

Ejemplo C#:

En el archivo *.xaml.cs*:

```
Button b = new Button();
b.Text = "Haz click aquí!";
b.Clicked += OnButtonClicked;
```

En ambos casos:

En el archivo *.xaml.cs*:

```
void OnButtonClicked(object sender, EventArgs args)
 {
        Console.Write("Hola Mundo!");
 }
```

Páginas, diseños o layouts y vistas

Las interfaces de usuario desarrolladas con Xamarin.Forms están compuesta por una serie de controles agrupados en:

- Páginas
- Diseños o layouts
- Views

Una página generalmente ocupa toda la pantalla. Una págna contiene un layout, el cual contiene a su vez vistas y posiblemente otros layouts.

Páginas

Representa una pantalla de la aplicación. Todos los tipos de página derivan de la clase *Page*. Algunos tipos de páginas son: *ContentPage*, *MasterDetailPage*, *NavigationPage*, *TabbedPage* y *CarouselPage*. Por defecto la página principal es de tipo *ContentPage*. En el apartado de navegabilidad estudiaremos los diferentes tipos de páginas.

Diseños o layouts

Son un tipo de vistas (heredan de la clase *View*) que actúan como contenedores de vistas y otros layouts. Algunos tipos de diseños o layouts son: *StackLayout*, *Grid*, *FlexLayout* y *ScrollView*. Por defecto, cuando creamos un proyecto en blanco, la página (de tipo *ContentPage*) contiene un *StackLayout*. En el apartado *Diseños o layouts* estudiaremos cómo funcionan y cómo utilizar los diferentes *layouts*.

Vistas

Son objetos de la interfaz de usuario tales como etiquetas, botones, cuadros de texto, también conocidos como controles. Las vistas soportadas heredan de la clase *View*. Algunas de las vistas disponibles son: *Label*, *Button*, *Image*, *CheckBox*, *Picker*, *ListView*, etc. En el próximo apartado veremos cómo utilizar algunas de estas vistas.

Vistas

En esta parte de la asignatura vamos a estudiar cómo añadir algunas de las vistas disponibles en *Xamarin.Forms*. Aprenderemos a incluirlas en nuestra aplicación mediante el uso de XAML y en C#.

Label

Permiten mostrar un texto compuesto por una o múltiples líneas.

Atributo /Propiedad	**Valores**	**Descripción**
Text	Texto	Texto que se mostrará por pantalla.
TextColor	Tipo enumerado. Color en hexadecimal.	Color del texto.
TextDecorations	*None* *Underline* *Strikethrough*	Indica si el texto debe aparecer subrayado o tachado.
FontSize	Número decimal. *Default*, *Micro*, *Small*, *Medium*, *Large*, *Body*, *Header*, *Title*, *Subtitle*,	Tamaño de la fuente.

	Caption	
BackgroundColor	Tipo enumerado. Color en hexadecimal.	Color de fondo.
FontAttributes	*None* *Bold* *Italic*	Indica si el texto debe aparecer en cursiva o negrita.
MaxLines	Número entero.	Indica el número máximo de líneas que se mostrarán.
HorizontalTextAlignment	Center End Start	Indica la alineación del texto: centrada (Center), a la derecha (End) o a la izquierda (Start).

x:Name

Como hemos mencionado anteriormente *x:Name* sirve para asignar un identificador a un elemento que nos permitirá acceder al objeto desde el código C# utilizando el identificador como si fuera una variable que hubiéramos declarado nosotros. Podremos obtener y modificar las propiedades del objeto como haríamos normalmente.

Button

Crea un botón que al pulsarlo puede desencadenar una determinada tarea.

Atributo / Propiedad	**Valores**	**Descripción**
Text	Texto	Texto que se mostrará en el botón.
TextColor	Tipo enumerado. Color en hexadecimal.	Color del texto.
FontSize	Número decimal. *Default*, *Micro*, *Small*, *Medium*, *Large*, *Body*, *Header*, *Title*, *Subtitle*, *Caption*	Tamaño de la fuente.
BackgroundColor	Tipo enumerado. Color en hexadecimal.	Color de fondo.
FontAttributes	*None* *Bold* *Italic*	Indica si el texto debe aparecer en cursiva o negrita.

BorderColor	Tipo enumerado. Color en hexadecimal.	Color del borde del botón.
BorderWidth	Número entero.	Tamaño del borde en píxeles.

Evento	**Descripción**
Clicked	Ocurre cuando se hace clic sobre el botón
Pressed	Ocurre cuando el botón está pulsado.

Entry

Crea un cuadro de entrada de texto.

Atributo / Propiedad	**Valores**	**Descripción**
TextColor	Tipo enumerado. Color en hexadecimal.	Color del texto.
BackgroundColor	Tipo enumerado. Color en hexadecimal.	Color de fondo.
Placeholder	Texto.	Texto que aparecerá en el cuadro de texto como indicador.
IsPassword	*true*, *false*.	Indica si se trata de una contraseña y por tanto se deben ocultar los caracteres.
MaxLength	Número entero.	Cantidad máxima de caracteres que se pueden introducir.
Keyboard	*Chat*, *Default*, *Email*, *Numeric*, *Plain*, *Telephone*, *Text*, *Url*.	Tipo de teclado que se mostrará.
ReturnType	*Default*, *Done*, *Go*, *Next*, *Search*, *Send*.	Determina la apariencia del botón de retorno.
IsTextPredictionEnabled	*true*, *false*.	Indica si se utilizará la predicción de texto o no.

Evento	Descripción
Completed	Ocurre cuando el usuario pulsa *enter* para indicar que ha finalizado la entrada de texto.
TextChanged	Ocurre cuando el texto es modificado.

Colores

Cuando asignamos un color a un fondo o a un texto desde XAML tenemos dos opciones:

- Escribir el nombre del color:
  ```
  BackgroundColor="LightGray"
  ```
- Escribir el color en hexadecimal:
  ```
  BackgroundColor="#FF3333"
  ```

Desde C# la forma de asignar un color es diferente.

- Podemos utilizar la clase *Color*, indicando el nombre del color:
  ```
  lTexto.TextColor = Color.Red;
  ```
- Utilizar RGB mediante el método *FromRgb* de la clase *Color*:
  ```
  lTexto.TextColor = Color.FromRgb(255,0,0);
  ```
- Utilizar hexadecimal mediante el método *FromHex* de la clase *Color*:
  ```
  lTexto.TextColor = Color.FromHex("FF0000");
  ```

Ejercicio 1

Desarrolla una aplicación que tenga un *entry*, un botón y un *label*. Cuando se pulse el botón el texto que haya en el *entry* se mostrará en el *label*.

Ejercicio 2

Desarrolla una aplicación que tenga un botón y un *label*. Al iniciar la aplicación el *label* mostrará le texto "Hola". Cuando se pulse el botón el *label* pasará a mostrar el texto "Mundo", pero si se vuelve a pulsar volverá a mostrar "Hola". Es decir, al pulsar el botón el texto del *label* irá alternando entre "Hola" y "Mundo".

Ejercicio 3

En este ejercicio se pretende implementar una calculadora. La interfaz de usuario tendrá dos campos de entrada para introducir dos números y uno de salida para mostrar el resultado. Dispondrá de cuatro botones (suma, resta, multiplicación y división), cuando se pulse uno de estos botones se mostrará el resultado de realizar la operación sobre los números introducidos.

Ejercicio 4

Crea una aplicación con un *entry* y un *label*. En todo momento el *label* mostrará el mismo texto que el *entry*. Es decir, a la vez que se escribe en el *entry* el texto aparecerá en el *label*. Si la longitud del texto es inferior a 5 caracteres el color del texto en el *label* será rojo, si la longitud está entre 5 y 10 (inclusive) se mostrará de color negro; y si es superior a 10 se mostrará en rojo.

Ejercicio 5

Crea una aplicación que tenga dos entradas de texto y un botón. Las entradas de texto solo permitirán introducir números. Al pulsar el botón la aplicación mostrará en un *label* los números pares que existen entre los dos números introducidos.

Ejercicio 6

Crea una aplicación que tenga una entrada de texto y un *label* que mostrará el texto *Hola Mundo!*. En la entrada de texto solo se podrá introducir números. Conforme se escriba el número en la entrada de texto el tamaño de la fuente del *label* cambiará al valor indicado.

Añadiendo vistas dinámicamente

Es posible añadir vistas dentro de un un layout dinámicamente, esto es, añadir componentes de tipo *Button*, *Entry* o *Label*, etc, a la interfaz de usuario haciendo uso de código en C#. Para ello podemos asignar un nombre al propio elemento *StackLayout*:

```
<StackLayout x:Name="lMiStackLayout">
</StackLayout>
```

Posteriormente en el código podemos acceder a la propiedad *Children* del layout (que es una colección con todas las vistas que contiene) y añadir mediante el método *Add* una nueva vista.

```
Label l = new Label();
l.Text = "Hola Mundo!";
lMiStackLayout.Children.Add(l);
```

Ejercicio 7

Crea una aplicación que tenga dos entradas de texto y un botón. La primera entrada de texto será numérica, mientras que la segunda será de texto. Al pulsar el botón la aplicación creará dinámicamente múltiples *label* (tantos como el número indicado en la primera entrada de texto). El texto de cada *label* será el mismo que se haya escrito en la segunda entrada de texto.

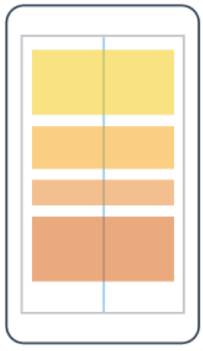
Alineación, expansión y tamaño de las vistas

Todas las vistas disponen de dos atributos *HorizontalOptions* y *VerticalOptions* de tipo *LayoutOptions*. La estructura *LayoutOptions* encapsula dos tipos de preferencias: alineación y expansión.

Alineación

Indica la alineación de la vista, determinando su posición respecto al contenedor padre.

Atributo	**Valores**	**Descripción**
HorizontalOptions	*Start*	Alinea el elemento a la izquierda.
	Center	Centra el elemento horizontalmente.
	End	Alinea el elemento a la derecha.
	Fill	La vista ocupa todo el ancho del *layout*. Esta es la opción por defecto.
VerticalOptions	*Start*	Alinea el elemento arriba.
	Center	Centra el elemento verticalmente.
	End	Alinea el elemento en la parte inferior.
	Fill	La vista ocupa todo el alto del *layout*. Esta es la opción por defecto.



StackLayout

Más adelante estudiaremos los diferentes tipos de *layout* que existen, de momento es importante saber que estamos trabajando con un *layout* de tipo *StackLayout* que por defecto dispone las vistas siguiendo una línea unidimensional ya sea vertical u horizontal.

- Si la orientación del *StackLayout* es vertical (orientación por defecto), el *layout* apila las vistas de arriba hacia abajo, ocupando todo el ancho e "ignorando" la alineación vertical indicada*.
- Si la orientación del *StackLayout* es horizontal, el layout apila las vistas de izquierda a derecha, ocupando todo el alto e "ignorando" la alineación horizontal indicada*. Para que un *StackLayout* sea horizontal hay que indicar que su propiedad *Orientation* es "*Horizontal*"

En ambos casos puede suceder que quede espacio al final del *layout* sin ocupar por las vistas ya que no llegan al final del *layout*.

** La alineación se ignora porque las vistas están apiladas y no tienen margen de movimiento, podríamos decir que ya están alineadas o que no hay ninguna diferencia al cambiar su alineación.*

Expansión

Se utiliza solo dentro de un *layout* de tipo *StackLayout* e indica que la vista debe ocupar el espacio disponible si lo hubiese. Esto se indica añadiendo el sufijo *AndExpand* al valor del atributo *HorizontalOptions* y *VerticalOptions* según corresponda.

- Si la orientación del *StackLayout* es vertical (orientación por defecto) solo podrá quedar espacio sin ocupar verticalmente y por tanto sólo tiene sentido utilizar el sufijo en el atributo *VerticalOptions*.
- Si la orientación del *StackLayout* es horizontal sólo podrá quedar espacio sin ocupar horizontalmente y por tanto sólo tiene sentido utilizar el sufijo en el atributo *HorizontalOptions*.

Se puede utilizar el sufijo en más de una vista, esto hará que se repartan el espacio disponible en partes iguales.

Imaginemos que tenemos un *StackLayout* con una orientación vertical (orientación por defecto) y que existe espacio libre al final del *layout* ya que tenemos pocas vistas. Si utilizamos el sufijo *AndExpand* en el atributo *VerticalOptions* de alguna de las vistas, estas pasarán a repartirse el espacio libre del layout. Puesto que ahora esas vistas sí que tiene margen de movimiento, ya no se ignora la alineación vertical indicada.

Alto y Ancho

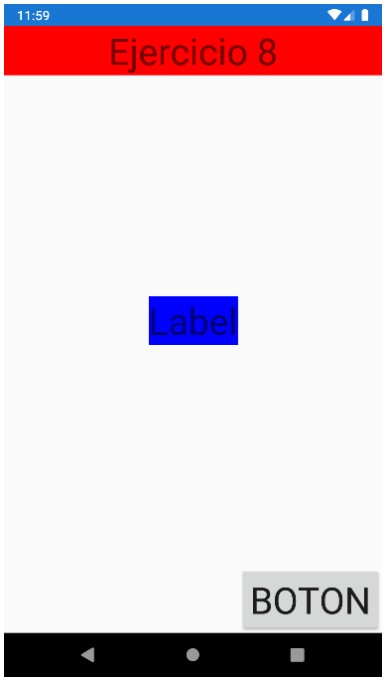
Podemos indicar el alto y ancho de una vista mediante los siguientes atributos:

Atributo	**Valores**	**Descripción**
WidthRequest	Número	Ancho de la vista
HeightRequest	Número	Alto de la vista

La alineación *Fill* por lo general invalida el valor del atributo *WidthRequest* y *HeightRequest*.

Ejercicio 8

Haz una aplicación que replique la siguiente interfaz de usuario haciendo uso exclusivamente de XAML:



Ejercicio 9

Repite el ejercicio anterior, pero haciendo uso únicamente de C#. Es decir, el archivo *MainPage.xaml* solo debe contener el siguiente código y no debe ser modificado:

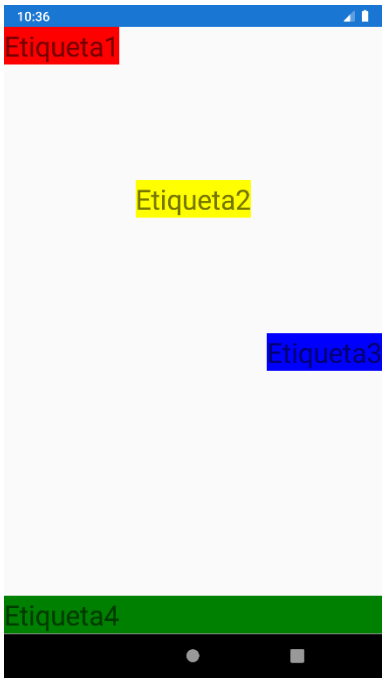
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Ejercicio9.MainPage">

    <StackLayout x:Name="slMiLayout">
    </StackLayout>

</ContentPage>
```

Ejercicio 10

Haz una aplicación que replique la siguiente interfaz de usuario haciendo uso exclusivamente de XAML:

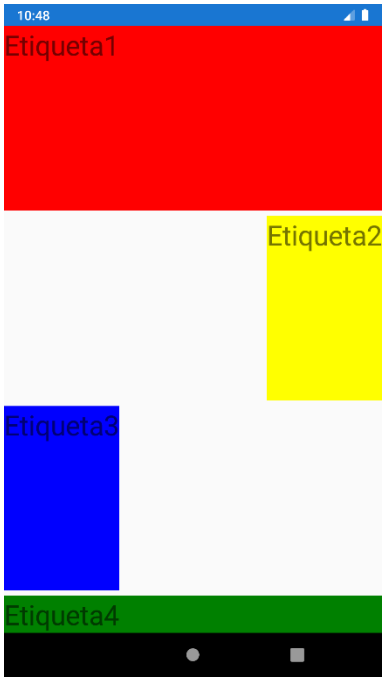
Ejercicio 11

Haz una aplicación que replique la siguiente interfaz de usuario haciendo uso exclusivamente de XAML:



Ejercicio 12

Haz una aplicación que replique la siguiente interfaz de usuario haciendo uso exclusivamente de XAML:

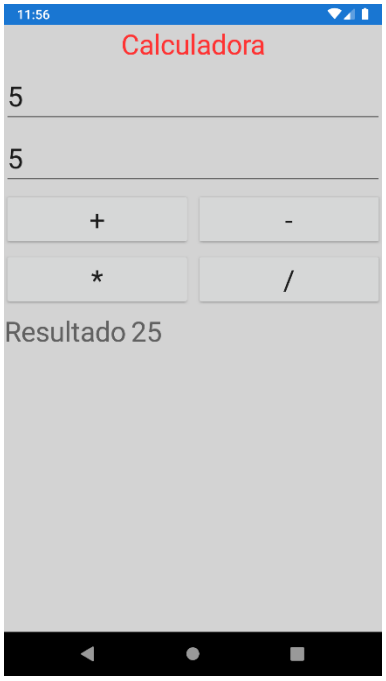


Anidación de *Stacklayout*

Si nos fijamos en cualquiera de los archivos *MainPage.xaml* siempre hay un elemento raíz llamado *ContentPage*. Este elemento solo puede tener un único elemento hijo que, de momento, siempre es un *StackLayout*. Como hemos visto hasta ahora un *StackLayout* sí que puede contener múltiples hijos (*Label*, *Entry*, *Button*, etc.) e incluso puede contener otros *StackLayout*. Recuerda que por defecto un *StackLayout* alinea sus hijos verticalmente, pero podemos hacer que los alinee horizontalmente mediante la propiedad *Orientation="Horizontal".*

Ejercicio 13

Modifica el ejercicio de la calculadora (ejercicio 3) para que muestre la interfaz de la siguiente forma:



Ejercicio 14

Realiza una interfaz de usuario en la que cada vez que se pulse un botón *Añadir* se añadan dos *labels* (uno al lado del otro) ocupando todo el ancho de la pantalla. El color de fondo de cada *label* debe ser asignado aleatoriamente. A continuación se muestra una captura de la aplicación tras haber pulsado cuatro veces el botón añadir.

Image

Inserta una imagen. El archivo lo ubicaremos dentro la carpeta *resources* > *drawable* del proyecto de *Android*.

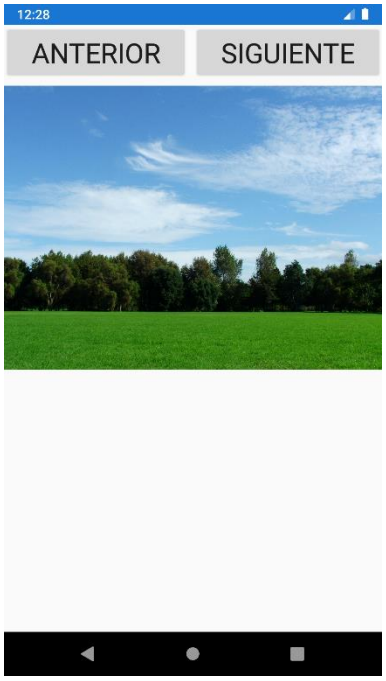
Atributo / Propiedad	Valores	Descripción
Source	URI recurso	Imagen que se quiere mostrar.
Aspect	*AspectFill*	Escala la imagen para ocupar toda la vista, recortándola si es necesario.
	AspectFit	Escala la imagen para que quepa toda la imagen en la vista, añadiendo espacios en blanco si es necesario.
	Fill	Ajusta la imagen para que ocupe toda la vista, distorsionando la imagen si es necesario.

Ejercicio 15

Crea una aplicación con tres botones y una vista *image*. Al pulsar cada uno de los botones se mostrará una imagen diferente.

Ejercicio 16

Se quiere implementar una galería de fotos. Para ello crea un nuevo proyecto con dos botones (anterior y siguiente) y una vista de tipo *image*. Los botones *Anterior* y *Siguiente* cambiarán la imagen. En el caso de que estemos en la última imagen y pulsemos *Siguiente* se mostrará la primera imagen. Si estamos en la primera imagen y pulsamos *Anterior* se mostrará la última. En total se deben mostrar 5 imágenes diferentes.



Checkbox

Tipo de botón que puede estar marcado o desmarcado.

Atributo / Propiedad	Valores	Descripción
IsChecked	*true* *false*	Indica si el *checkbox* está marcado o desmarcado.
IsEnabled	*true* *false*	Indica si el *checkbox* está activo, es decir, si se puede marcar/desmarcar.
Color	Tipo enumerado. Color en hexadecimal.	Color del checkbox.

Evento	Descripción
Checked Changed	Ocurre cuando cambia el estado del *checkbox*.

Ejercicio 17

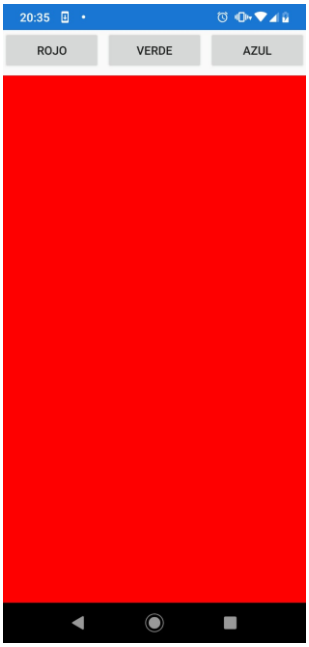
Todas las vistas disponen de una propiedad *IsVisible* que nos permite indicar si queremos que la vista se muestre (*true*) u oculte (*false*). Realiza la siguiente interfaz de usuario de forma que al desactivar los *checkbox* se oculten las etiquetas correspondientes. Por ejemplo, si desactivamos el *checkbox* rojo el *label* rojo dejará de estar visible.

Ejercicio 18

Crea una aplicación con un único botón definido en XAML. Cuando se pulse el botón cambiará su color de fondo a un color aleatorio. No utilices el atributo *x:Name*.

Ejercicio 19

Un *BoxView* permite representar un rectángulo simple al que se le puede cambiar el color de fondo. Puedes encontrar información sobre cómo utilizarlo en la página web de la documentación de *Xamarin*. Crea una aplicación con tres botones (*Rojo*, *Verde* y *Azul*) y un *BoxView* con el siguiente diseño:



Al pulsar uno de los botones el *BoxView* cambiará de color.

Ejercicio 20

Desarrolla una aplicación con un botón *Añadir* que añada *labels* cuyo color de fondo sea rojo, verde o azul (el color se elegirá aleatoriamente).

Mediante tres *checkbox*, uno para cada color, y un botón *Borrar*, podremos elegir los *labels* que queremos eliminar.

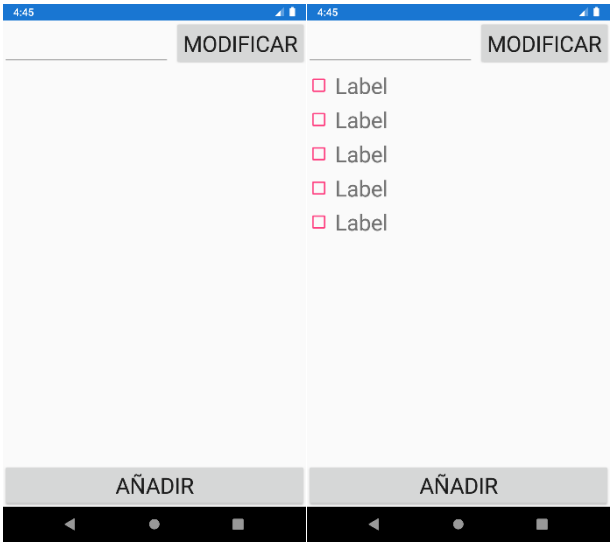
A continuación, se muestran varias capturas de la aplicación. A la izquierda se muestra la aplicación antes de añadir ningún *label* y a la derecha después de pulsar el botón *Añadir* diez veces.

En las siguientes capturas se muestra el resultado de borrar los *label* de color rojo y verde.

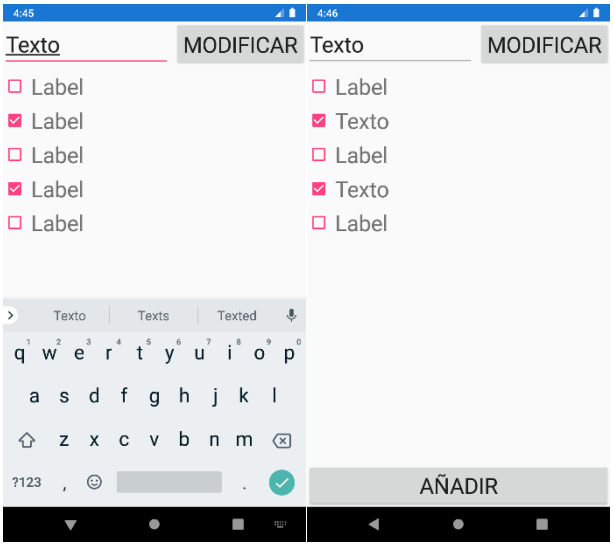
Ejercicio 21

Desarrolla una aplicación con un botón *Añadir* que al pulsarlo añada un *checkbox* junto con un *label*. La aplicación también tendrá un *entry* y un botón *Modificar*. Al pulsar el botón *modificar*, aquellos *labels* cuyo *checkbox* esté seleccionado, modificarán su texto al texto indicado en el *entry*.

En la imagen de la izquierda se puede ver la aplicación tras ejecutarla. En la de la derecha se muestra la aplicación tras pulsar 5 veces el botón *Añadir*.



En las siguientes imágenes se muestra la aplicación tras seleccionar algunos *checkbox* y escribir texto en el *entry* (imagen de la izquierda) y el resultado tras pulsar el botón *modificar* (imagen de la derecha).



ListView

El elemento *ListView* se utiliza para mostrar listas desplazables. Los elementos que se mostrarán son aquellos guardados en su propiedad *ItemsSource*, en la que podremos guardar cualquier colección que implemente la interfaz *IEnumerable*.

Atributo / Propiedad	Valores	Descripción
ItemsSource	IEnumerable	Colección de elementos que se mostrará en la lista.
SelectedItem	Object	Elemento seleccionado.

Evento	Descripción
ItemSelected	Ocurre cuando se selecciona un elemento.

La forma más sencilla de añadir elementos desde el *XAML* es utilizando un *Array*:

```
<ListView>
      <ListView.ItemsSource>
          <x:Array Type="{x:Type x:String}">
            <x:String>monopoli</x:String>
            <x:String>parchis</x:String>
            <x:String>cluedo</x:String>
          </x:Array>
      </ListView.ItemsSource>
</ListView>
```

El equivalente en C# sería:

```
var listView = new ListView();
listView.ItemsSource = new string[]
{
  "monopoli",
  "parchis",
  "cluedo"
};
```

Otro ejemplo con un *List* sería:

```
List<Empleado> arList = new List<Empleado>();
arList.Add(new Empleado { Nombre = "Pepe" }); ;
arList.Add(new Empleado { Nombre = "Jorge" }); ;
arList.Add(new Empleado { Nombre = "Sofía" }); ;
m.ItemsSource = arList;
```

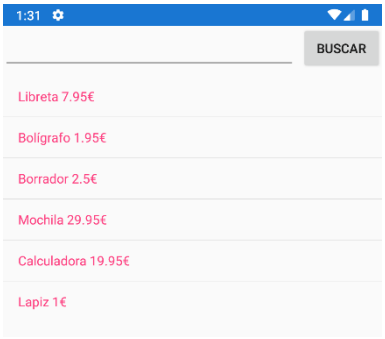
Por defecto el *ListView* llamará al método *ToString* de los elementos guardados en el *ItemsSource* a la hora de mostrarlos.

En este caso la clase *Empleado* sobreescribe el método *ToString* devolviendo su nombre.

Mediante el evento *SelectedItem* podemos detectar el caso en el que se seleccione un elemento de la lista. Entre los parámetros del método recibiremos una variable de tipo *SelectedItemChangedEventArgs* que incluye dos propiedades: *SelectedItem* (el elemento seleccionado) y *SelectedItemIndex* (la posición en la que se encuentra).

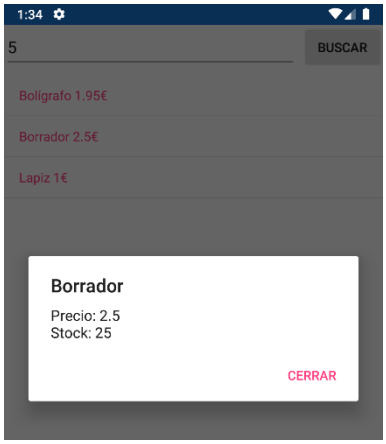
Ejercicio 22

Desarrolla una aplicación que muestre una lista de artículos y su precio.



La aplicación permitirá filtrar los artículos por precio mediante un *entry*, de forma que al pulsar el botón *buscar* solo se mostrarán los artículos cuyo precio sea inferior al indicado. A continuación, se muestra qué ocurre al buscar los artículos cuyo precio es inferior a 5€.

Al seleccionar uno de los elementos de la lista se mostrará en un diálogo (`DisplayAlert`) los datos del artículo, que incluye: su nombre, precio y *stock*. A continuación se muestra qué ocurriría al seleccionar el elemento *Borrador*.



Picker

Se trata de un control que permite elegir un elemento de una lista. Se corresponde con el control clásico llamado desplegable, *dropdown* o *combobox*.

Atributo / Propiedad	**Valores**	**Descripción**
Title	string	Título de la vista.
TitleColor	Tipo enumerado. Color en hexadecimal.	Color utilizado para el título.
ItemsSource	IList	Elementos a mostrar.
SelectedIndex	int	Elemento seleccionado.
SelectedItem	object	Elemento seleccionado.

Evento	**Descripción**
SelectedIndexChanged	Ocurre cuando se cambia el elemento seleccionado.

El funcionamiento es el mismo que el *ListView*.

Ejercicio 23

Realiza un conversor de moneda para realizar conversiones entre euros, dólares y libras. La interfaz tendrá dos *Picker* uno para indicar la moneda de origen y otro para indicar la moneda a la que queremos convertir.

DatePicker

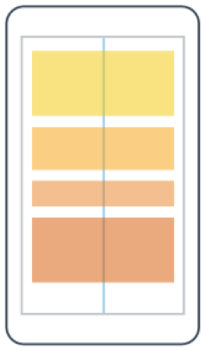
Ejercicio 24

Crea un nuevo proyecto investiga en Google cómo utilizar la vista *DatePicker* ya sea mediante XAML o C#.

Diseños o layouts

Los *layouts* o diseños permiten distribuir las vistas que contienen. Los diferentes *Layout* heredan de *View*, es decir, también son vistas, por lo que un *layout* puede contener también otros *layouts*.

StackLayout



StackLayout

Como hemos visto anteriormente este *layout* apila las vistas que contiene de forma vertical (por defecto) u horizontal.

```
<StackLayout >
<!-- Añadir las vistas aquí -->
</StackLayout>
```

Atributo / Propiedad	Valores	Descripción
Orientation	*Vertical* *Horizontal*	Indica si las vistas se deben apilar vertical (por defecto) u horizontalmente.
Spacing	Número	Deja el espacio especificado entre las vistas.

Padding	Cuatro números separados por comas. left, top, right, bottom	Deja el espacio indicado alrededor del layout.
BackgroundColor	Tipo enumerado. Color en hexadecimal.	Color de fondo.

Grid

Grid

Layout que organiza los elementos dentro de una tabla. En la definición del *Grid* se indica mediante los elementos *<Grid.RowDefinitions>* y *<Grid.ColumnDefinitions>* la cantidad de filas y columnas que la componen, así como su anchura/altura. Por ejemplo, el siguiente Grid tiene 2 filas y 2 columnas con un total de 4 celdas.

```
<Grid>
      <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
      </Grid.ColumnDefinitions>
      <!-- Añadir las vistas aquí -->
</Grid>
```

Para indicar en qué celda se debe ubicar una vista se utilizan los siguientes atributos:

Atributo / Propiedad	**Valores**	**Descripción**
Grid.Row	Número	Número de la fila. Siendo la primera la número 0.
Grid.Column	Número	Número de la columna. Siendo la primera la número 0.

Tamaño de filas y columnas

Podemos especificar el tamaño de las filas y columnas mediante los atributos *Height* y *Width* respectivamente de los elementos *RowDefinition* y *ColumnDefinition*.

En el elemento *RowDefinition* podremos utilizar el atributo *Height*. En el elemento *ColumnDefinition* podremos utilizar el atributo *Width*.

Atributo / Propiedad	**Valores**	**Descripción**
Height Width	*Auto*	Se ajusta al contenido de las celdas de la columna/fila. Si no hay, su tamaño será 0.
	Número	Tamaño absoluto. Indica la dimensión que debe tener.
	Notación asterisco	Se ajusta al espacio disponible de forma proporcional (por defecto).

Notación asterisco

Si indicamos que el valor del atributo Height o Width es "*", la fila/columna ocupará el espacio libre disponible. Si varias filas/columnas tienen asignado el valor "*" se repartirán proporcionalmente el espacio disponible.

Es posible anteponer antes del asterisco un número entero que indicará la proporción del espacio disponible que ocupará la fila o columna.

Por ejemplo:

```
<Grid HeightRequest="300">
            <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
</Grid>
```

En este caso las tres filas tendrán la misma altura, resultado de dividir la altura del Grid en tres.

```
 <Grid HeightRequest="200">
            <Grid.RowDefinitions>
```

```
            <RowDefinition Height="1*" />
            <RowDefinition Height="2*" />
            <RowDefinition Height="3*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
</Grid>
```

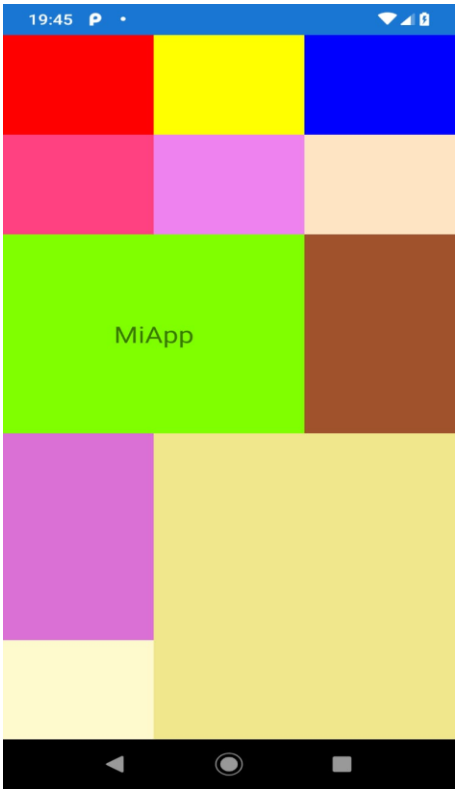
En este caso la primera fila ocupa 1/6 parte de la altura del *Grid*, la segunda ocupa 2/6 y la tercera 3/6 partes.

```
<Grid HeightRequest="200">
        <Grid.RowDefinitions>
            <RowDefinition Height="100" />
            <RowDefinition Height="1*" />
            <RowDefinition Height="3*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
    </Grid>
```

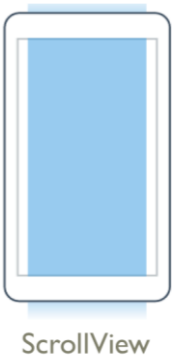
En este caso la primera fila ocupa 100 píxeles, por lo que solo quedarán otros 100 disponibles. La segunda fila ocupará 1/4 parte de esos 100 píxeles, mientras que la última fila ocupará 3/4 partes.

Ejercicio 1

Crea un nuevo proyecto e intenta reproducir la siguiente interfaz de usuario haciendo uso de un *Grid*. Para poder realizarlo necesitarás utilizar la propiedad *Grid.ColumnSpan* y *Grid.RowSpan,* investiga cómo utilizarlas.



Scroll View



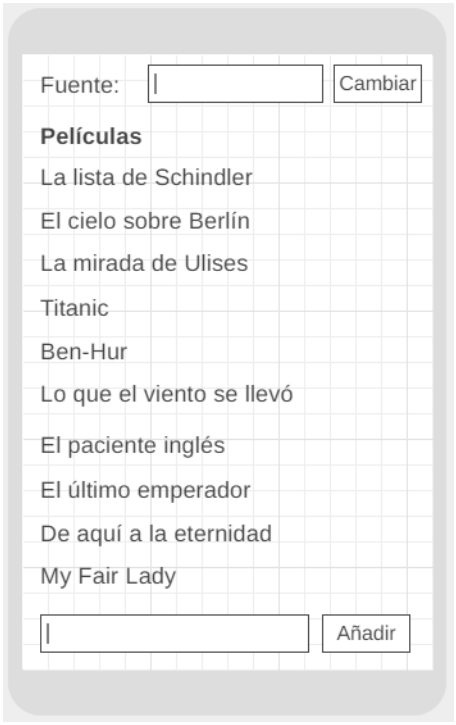
Un *ScrollView* contiene un *layout* que se podrá desplazar en el caso de que su contenido se salga de la pantalla. *ScrollView* tiene una propiedad *Content* a la que se puede asignar una vista o *layout*.

```
<ContentPage.Content>
    <ScrollView>
        <StackLayout>
            <BoxView BackgroundColor="Red"
HeightRequest="600" WidthRequest="150" />
            <Entry />
        </StackLayout>
    </ScrollView>
</ContentPage.Content>
```

Atributo / Propiedad	Valores	Descripción
ScrollX	Número	Devuelve el desplazamiento en el eje X (solo lectura).
ScrollY	Número	Devuelve el desplazamiento en el eje Y (solo lectura).
HorizontalScrollBarVisibility	*true* *false*	Indica si la barra de desplazamiento horizontal es visible o no.
VerticalScrollBarVisibility	*true* *false*	Indica si la barra de desplazamiento vertical es visible o no.

Ejercicio 2

Crea una aplicación con la siguiente interfaz:



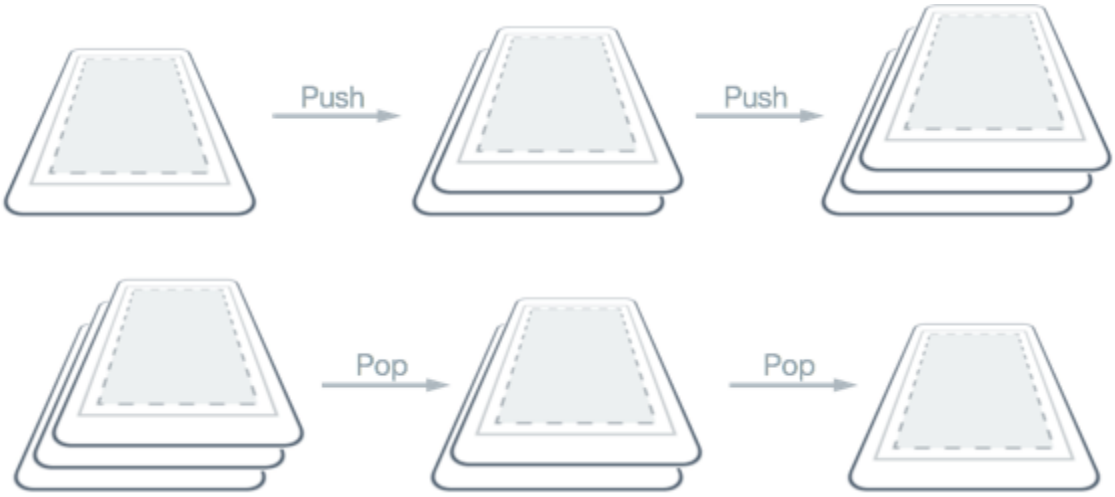
La aplicación muestra una lista de películas mediante *labels.* Las películas se pueden añadir mediante el *entry* y el botón de la parte inferior. Con la lista de películas se debe poder hacer *scroll*, pero los *entry* y los botones deben permanecer fijos. El *entry* y el botón de la parte superior permiten cambiar el tamaño de la fuente de todos los *labels*.

Navegabilidad

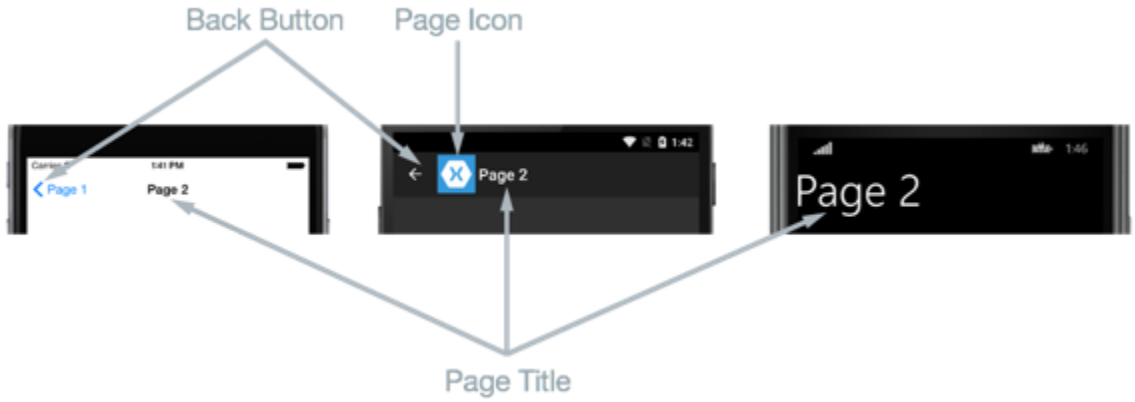
Xamarin nos proporciona diferentes patrones para implementar la navegación entre páginas o pantallas de nuestra aplicación.

Navegación jerárquica

Basada en una estructura de pila LIFO. Cada vez que visitamos una página esta se inserta en la pila. Cuando pulsamos el botón para ir atrás, se desapila la página anterior.



La navegación jerárquica se representa en cada sistema operativo de la siguiente forma:



NavigationPage

Para implementar la navegación jerárquica utilizaremos la clase *NavigationPage*. Su constructor recibe como parámetro un objeto de tipo *Page* aunque se recomienda que sea del tipo *ContentPage* (que hereda de *Page*). La primera página que añadimos a la jerarquía se llama página raíz. Con el siguiente fragmento de código hacemos que *MainPage* pase a ser la página raíz.

```
public App()
{
    InitializeComponent();
    MainPage = new NavigationPage(new MainPage());
}
```

Para navegar a una página utilizaremos la siguiente línea de código:

```
await Navigation.PushAsync (new Pagina());
```

Donde *Pagina* es el nombre de la página a la que queremos ir.

Por ejemplo:

```
async private void Button_Clicked(object sender, EventArgs e)
{
     await Navigation.PushAsync(new Page1());
}
```

Para ir a la página anterior:

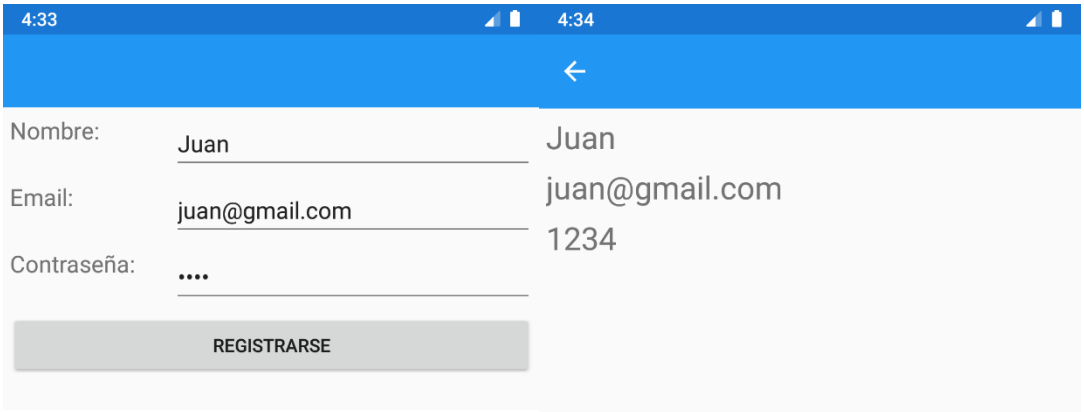
```
await Navigation.PopAsync ();
```

Para ir a la página raíz:

```
await Navigation.PopToRootAsync ();
```

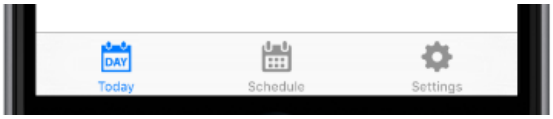
Ejercicio 1

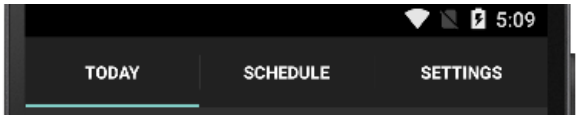
Crea una aplicación con dos páginas: la primera de ellas tendrá una serie de *entry* para que el usuario introduzca su nombre, email y contraseña. Al pulsar el botón *registrarse* se pasará a la segunda página en la que se mostrarán los datos introducidos.



Navegación por pestañas

Permite añadir una serie de pestañas que nos llevarán a diferentes páginas. La navegación por pestañas se representa en cada sistema operativo de la siguiente forma:





TabbedPage

Podemos crear una página con navegación por pestañas directamente desde *Visual Studio*, haciendo clic derecho sobre nuestra solución: *Agregar > Nuevo elemento > Página con pestañas*.

Las páginas que se muestran al seleccionar las diferentes pestañas se pueden añadir de dos maneras:

- En línea, mediante el uso de *ContentPage*. Por defecto la plantilla incluirá tres páginas de este tipo pudiendo añadir o quitar las que queramos. El diseño de cada una de las páginas lo haremos dentro de cada uno de los elementos *ContentPage*.

```
<ContentPage Title="Tab 1" />
<ContentPage Title="Tab 2" />
<ContentPage Title="Tab 3" />
```

- Mediante referencias. Podemos indicar que la página que queremos incluir en nuestro *TabbedPage* es externa, es decir, no vamos a definir su interfaz en el xaml del *TabbedPage*.

Para utilizar referencias seguiremos los siguientes pasos:

1. Crearemos una nueva página.
2. Añadiremos el siguiente atributo al elemento *TabbedPage*.

   ```
   xmlns:pages="clr-namespace:Solucion;assembly=Solucion"
   ```

   Donde *Solucion* es el nombre de nuestra solución.

3. Dentro del elemento *TabbedPage* añadiremos el siguiente elemento:

   ```
   <pages:NombrePagina/>
   ```

   Donde *NombrePagina* es el nombre de la página que hemos creado y que queremos que se muestre.

Tanto si añadimos las páginas mediante un *ContentPage* como mediante una referencia podemos indicar el título e icono de la pestaña mediante los siguientes atributos:

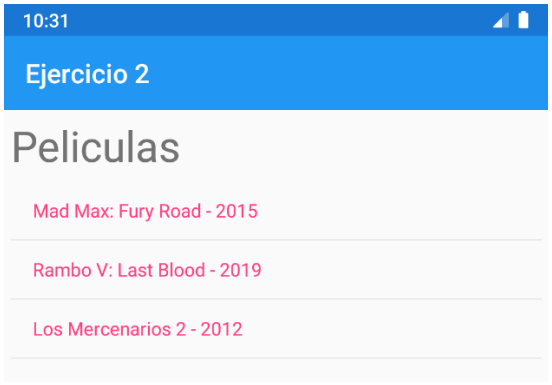
Atributo / Propiedad	**Valores**	**Descripción**
Title	Texto	Texto que se mostrará en la pestaña.
IconImageSource	URI recurso	Icono que se mostrará en la pestaña.

Algunos de los atributos que podemos utilizar en un *TabbedPage* son:

Atributo / Propiedad	**Valores**	**Descripción**
BarBackgroundColor	Tipo enumerado. Color en hexadecimal.	Color de fondo de las pestañas.
BarTextColor	Tipo enumerado. Color en hexadecimal.	Color del texto de las pestañas.
SelectedTabColor	Tipo enumerado. Color en hexadecimal.	Color de la pestaña seleccionada.
UnselectedTabColor	Tipo enumerado. Color en hexadecimal.	Color de las pestañas no seleccionadas.

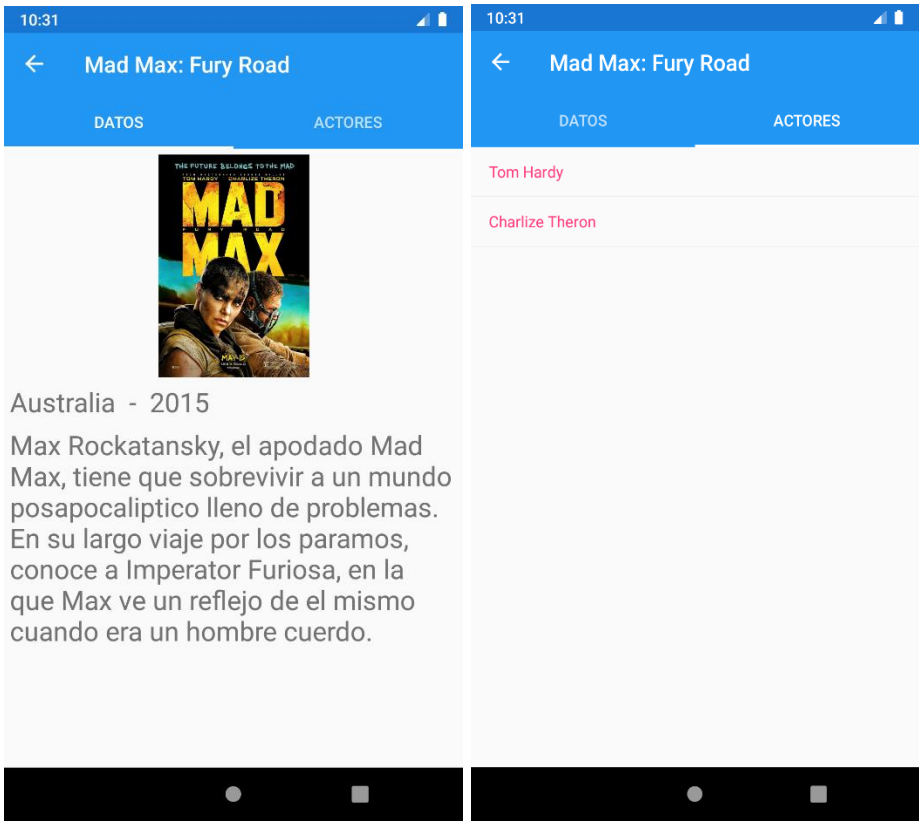
Ejercicio 2

Desarrolla una aplicación que muestre por pantalla una lista de películas con su título y año de la siguiente forma:



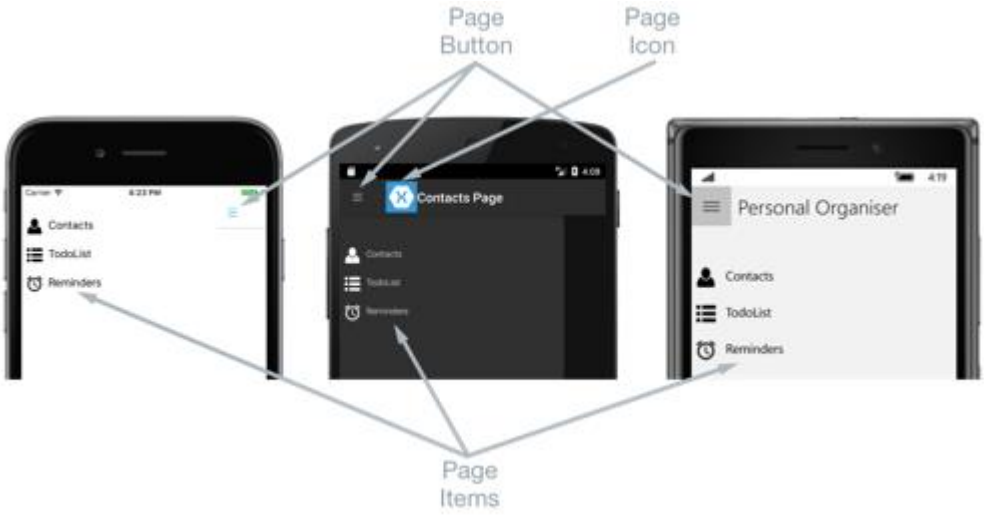
Al seleccionar una de las películas se mostrará en un *Tabbed Page* con dos pestañas:

- Datos: En ella se mostrará los datos de la película en una pestaña (título, año, portada, descripción).
- Actores: Mostrará la lista de actores que aparecen en la película.



Maestro detalle

Muestra una vista máster con una lista de elementos y una vista detalle con información sobre el elemento de la lista elegido. En los diferentes sistemas operativos se representa de la siguiente forma:



MasterDetailPage

Podemos crear una página master detalle directamente desde *Visual Studio*, haciendo clic derecho sobre nuestra solución: *Agregar* > *Nuevo elemento* > *Página de maestro y detalles* o la podemos crear de forma manual de la siguiente forma:

1. Crearemos 3 páginas de tipo *ContentPage*: *MasterDetalle*, *Master* y *Detalle*. La página *Master* es la que contiene el menú lateral, la página *Detalle* la que muestra la página al seleccionar un elemento del menú. *MasterDetalle* es la página que junta *Master* y *Detalle*.
2. En el *xaml* de *MasterDetalle* cambiaremos el nombre de la etiqueta *ContentPage* por *MasterDetailPage* y borraremos todo su contenido, incluido el elemento *ContentPage*. Hay que tener cuidado con el valor del atributo *x:Class* que debe ser el nombre de la clase.

```
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:d="http://xamarin.com/schemas/2014/forms/design"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             mc:Ignorable="d"
             x:Class="App1.MasterDetalle">
```

```
</MasterDetailPage>
```

3. En el código C# de *MasterDetalle* debemos asegurarnos de que el nombre de la clase es el mismo que el especificado en el atributo *x:Class* mencionado en el punto anterior. También hay que asegurarse de que la clase hereda de *MasterDetailPage*.

```
public partial class MasterDetalle : MasterDetailPage
 {
     public MasterDetalle()
     {
         InitializeComponent();
     }
 }
```

4. Dentro del *xaml* de *MasterDetalle* indicaremos la página que será master y la que será detalle. La página *Master* debe ser siempre de tipo ContentPage, mientras que la página *Detail* debe ser un *TabbedPage* o *NavigationPage*.

```
<MasterDetailPage.Master>
    <pages:Master/>
</MasterDetailPage.Master>
<MasterDetailPage.Detail>
    <NavigationPage>
        <x:Arguments>
            <pages:Detalle />
        </x:Arguments>
    </NavigationPage>
</MasterDetailPage.Detail>
```

Recordad que hay que definir el espacio de nombres *pages*.

4. Finalmente hay que añadir obligatoriamente a la página *Master* un título.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       x:Class="Ejercicio3.Master"
       Title="Página Maestra">
```

La clase *MasterDetailPage* tiene una propiedad llamada *IsPresented* que nos permite mostrar u ocultar el menú o master.

Atributo / Propiedad	Valores	Descripción
IsPresented	*true* *false*	Indica si el master se debe mostrar u ocultar.

Para añadir un botón en el menú (en la página Master) que al pulsarlo cargue una página en el detalle, dicho botón deberá ejecutar el siguiente código:

```
private void OnBotonPulsado(object sender, EventArgs e)
{
     ((MasterDetailPage)Parent).Detail = new NavigationPage(new Page1());
     ((MasterDetailPage)Parent).IsPresented = false;

}
```

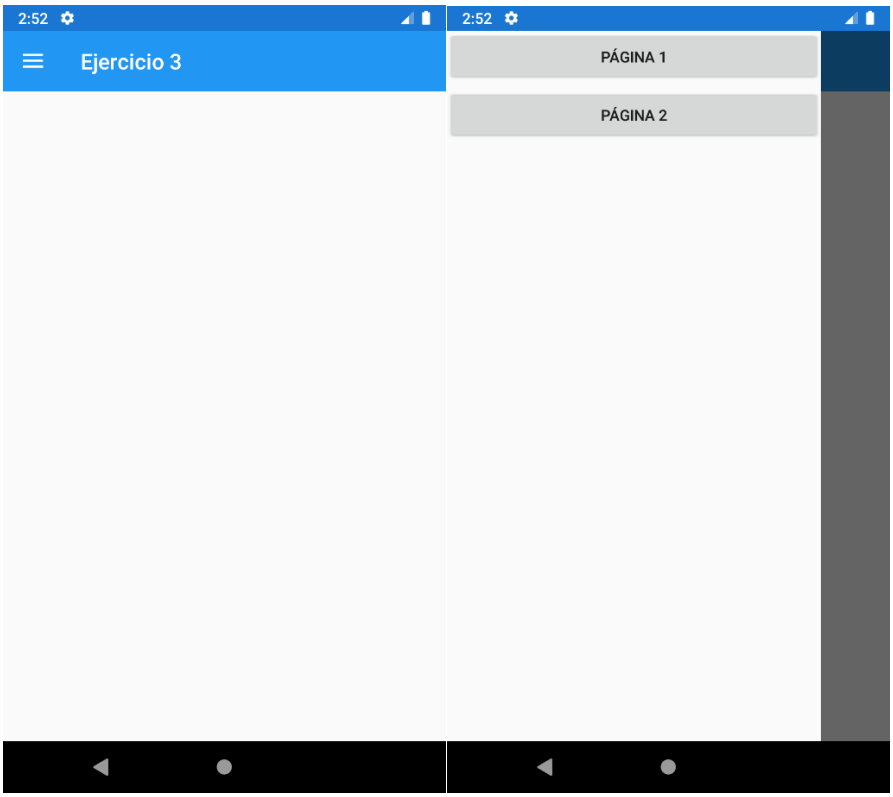
Como vemos se convierte la propiedad *Parent* de la página *Master* en *MasterDetailPage*. Esto podemos hacerlo ya que la página padre de *Master* es *MasterDetalle* que hereda de *MasterDetailPage.*

En primer lugar se asigna la propiedad *Detail* a la página que queramos navegar y en segundo lugar se oculta el menú.

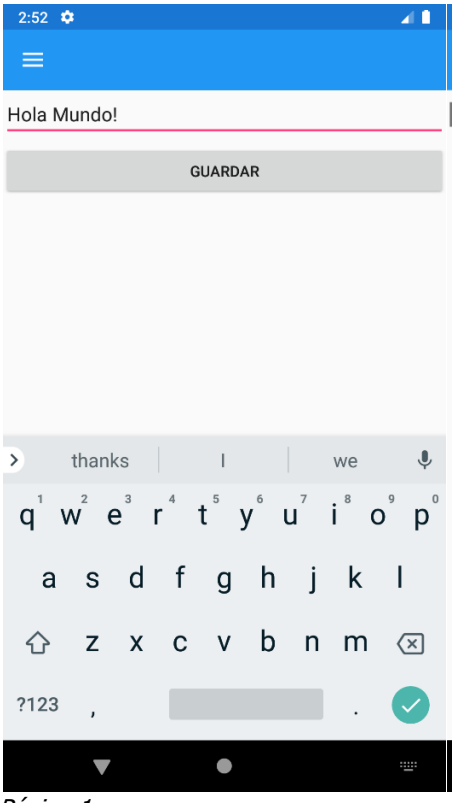
Ejercicio 3

Desarrolla una aplicación que haga uso de la navegación Master-Detalle. En el menú se mostrarán dos botones que al pulsarlos mostrarán páginas diferentes. En la primera habrá un *entry* y un botón *guardar*. En la segunda página se mostrará en un *label* el texto guardado en la primera página.

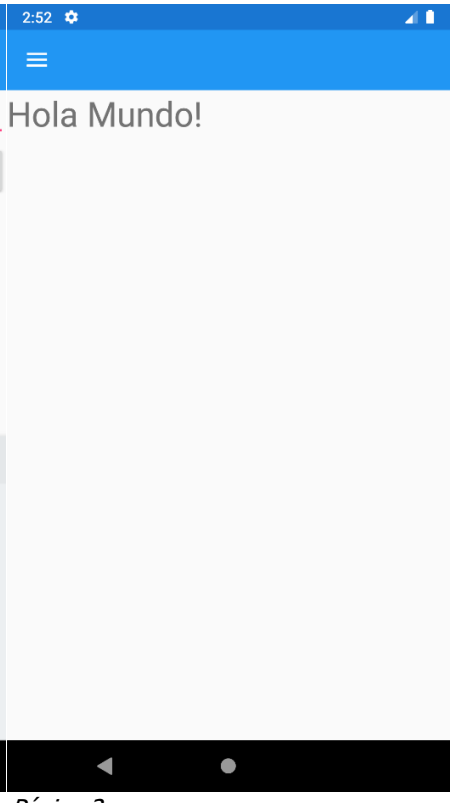
A continuación, se muestran unas capturas de la aplicación:



Página principal con el menú cerrado (izquierda) y abierto (derecha)



Página 1



Página 2

Ejercicio 4

Desarrolla una aplicación cuya página principal sea de tipo *Master-Detalle*. El menú mostrará dos opciones: *Noticias* y *Configuración*. Al seleccionar *Noticia* se mostrará una página que contiene un *label* con texto. Al seleccionar *Configuración* se mostrará una página en la que es posible indicar el tamaño de la fuente, el color y si está en negrita o no el *label* de la página *Noticia*.