

1. Introducción a JavaFX

Programación de Servicios y Process

Rafael Sala

IES Eduardo Primo Marqués



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit

1. Introducción a JavaFX.....	1
1. Primeros pasos con JavaFX	3
1.1. ¿Qué es JavaFX?	3
1.2. Creando nuestra primera aplicación JavaFX	3
1.3. Estructura de una aplicación JavaFX FXML	11
1.4. Entendiendo Scene Builder.....	13
2. Controles básicos y layouts.....	15
2.1. Las clases Stage y Scene	15
2.2. Algunos de los controles más útiles en JavaFX.....	17
2.3. Organizando los controles. El paquete "layout"	25
3. Eventos	30
3.1. Tipos de eventos principales.....	30
3.2. Definiendo manejadores y conectándolos con eventos	30
3.3. Ejemplos	33

1. Primeros pasos con JavaFX

1.1. ¿Qué es JavaFX?

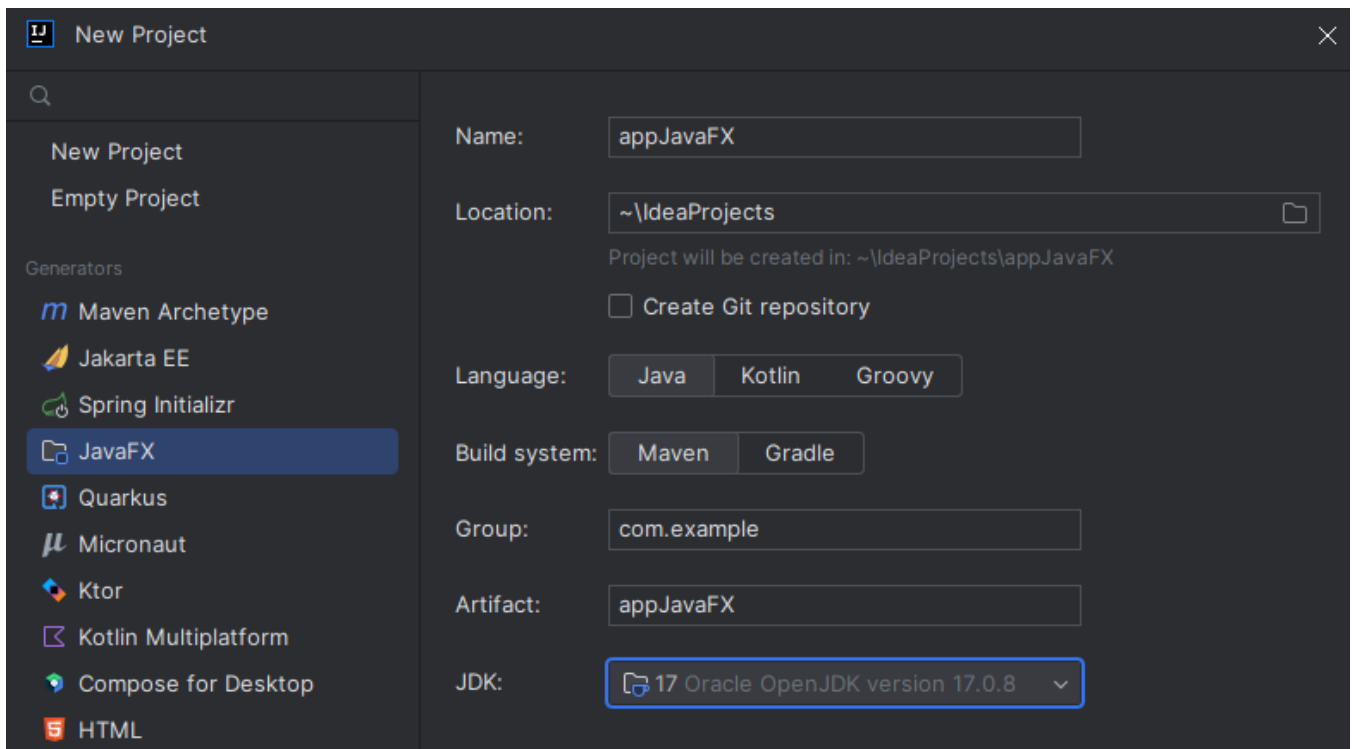
JavaFX es un conjunto de paquetes Java que nos permite crear una amplia variedad de interfaces gráficas de usuario (GUI), desde las clásicas con controles típicos como etiquetas, botones, texto, menús, etc., hasta algunas aplicaciones avanzadas y modernas, con algunas opciones interesantes, como animaciones o perspectiva.

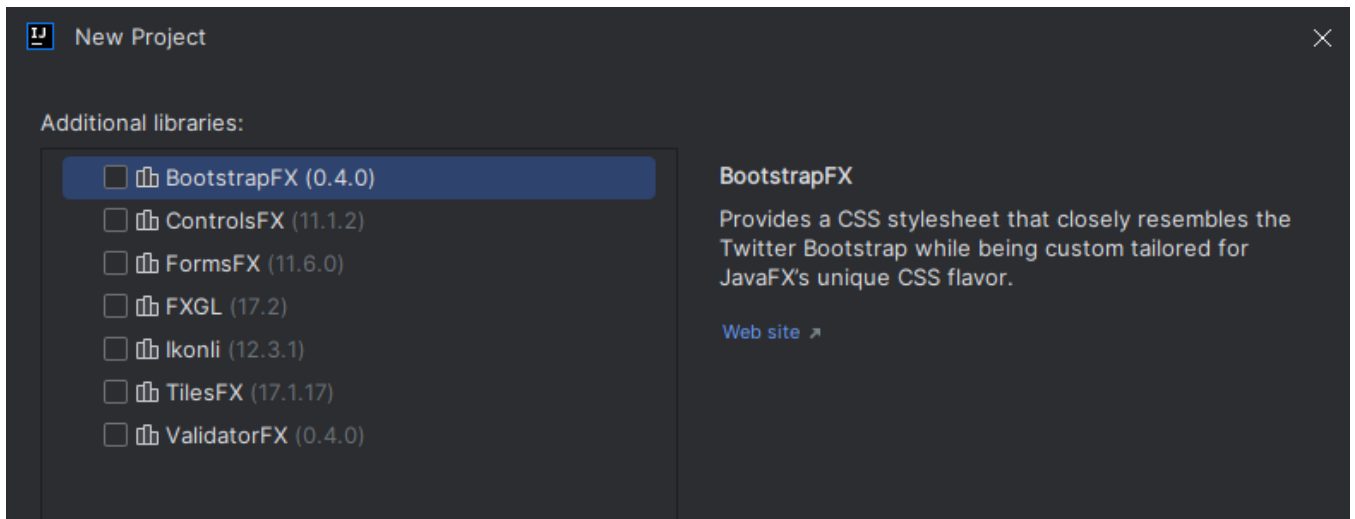
Si miramos hacia atrás, podemos ver JavaFX como una evolución de una librería Java anterior, llamada Swing, que sigue incluida en el JDK oficial, aunque se está quedando bastante obsoleta, y las posibilidades que ofrece son mucho más reducidas. Por eso ahora la mayoría de las aplicaciones Java de escritorio se están desarrollando con JavaFX.

1.2. Creando nuestra primera aplicación JavaFX

Para poder utilizar **Scene Builder** para crear nuestras interfaces (vistas) debemos crear un proyecto **"Java FXML application"** para que el IDE cree los archivos necesarios siguiendo el patrón Modelo-Vista-Controlador.

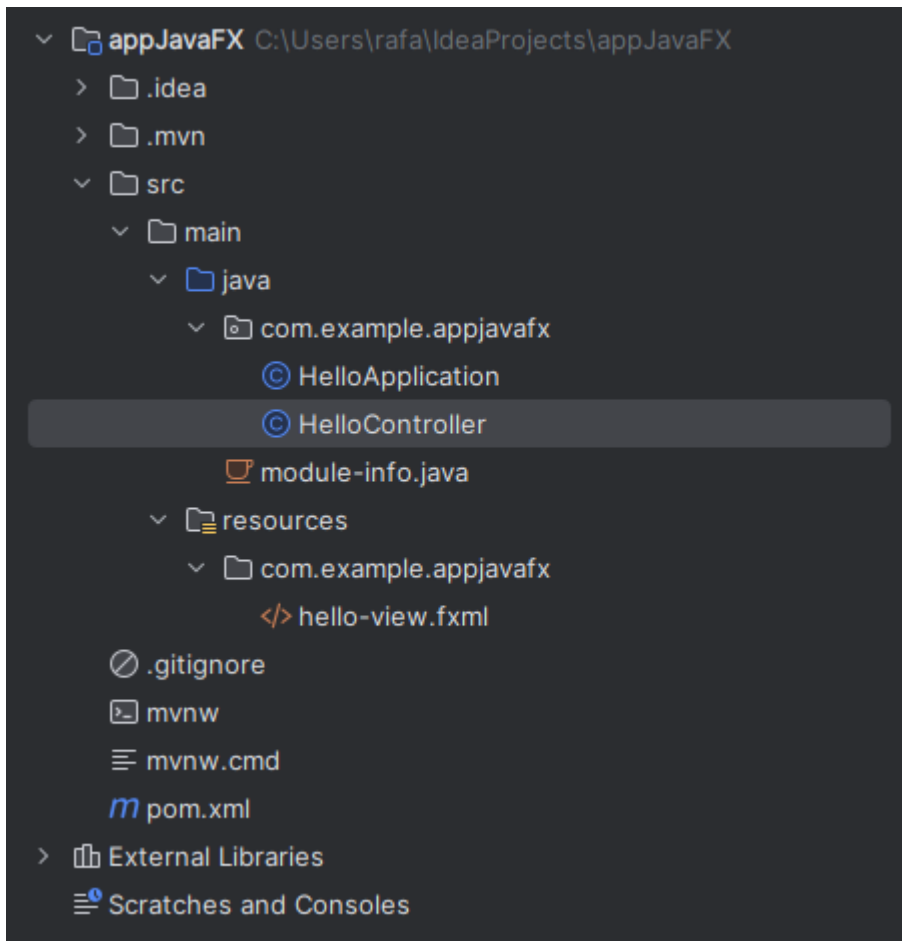
Después de eso, debemos dar un nombre al proyecto y al archivo FXML. Comprueba también que se creará una clase **Application**. Esta clase contendrá el método main que iniciará la aplicación.



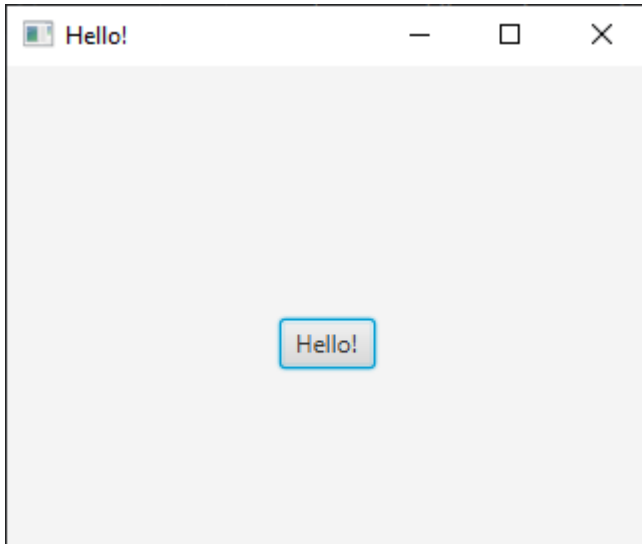


Esta es la estructura básica que tendrá nuestro proyecto. En este ejemplo el archivo **HelloApplication.java** contiene nuestra clase de aplicación (y el método main) que inicializará todo.

hello-view.fxml contiene nuestra única (por el momento) vista FXML que editaremos usando Scene Builder, y **HelloController.java** contiene nuestra clase controladora para esta vista.



De esta forma si compilamos el programa y lo ejecutamos debería aparecer la siguiente ventana:



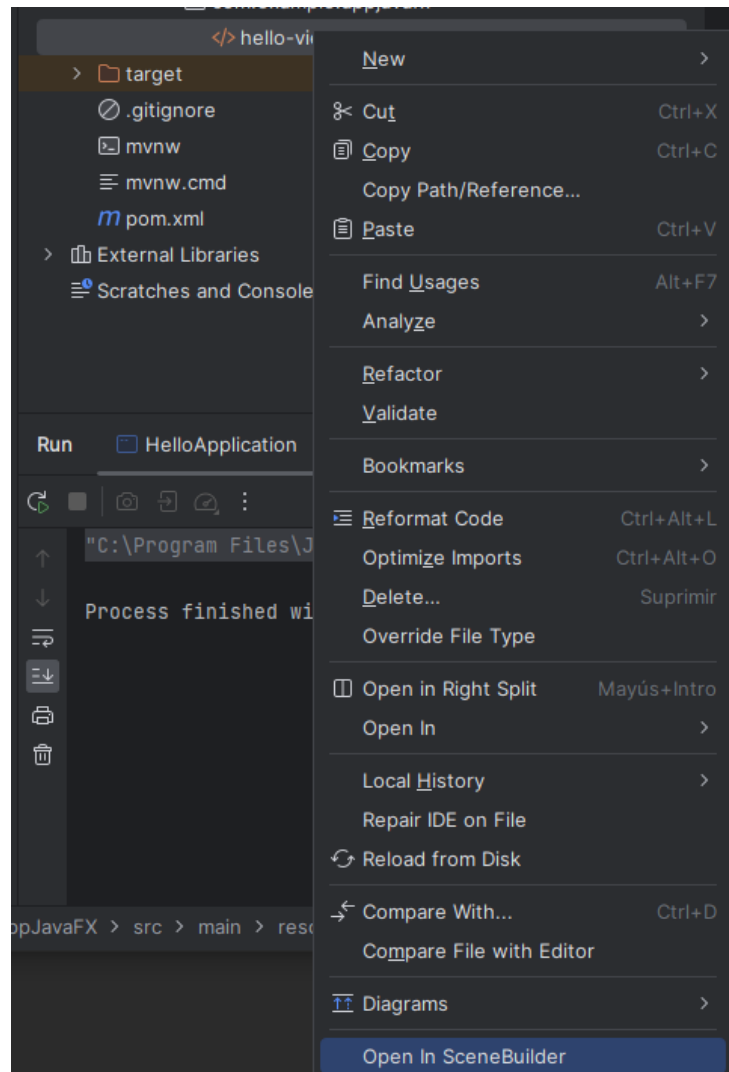
1.2.1. Añadiendo la librería JavaFX

Si tienes problemas para ejecutar la aplicación debes añadir la librería de JavaFX al proyecto tal y como se explicó en el Tema 0.

De todas formas, puedes curarte en salud y añadirla igualmente por si te encuentras con problemas a lo largo del desarrollo.

1.2.2. Editando la vista FXML

Para abrir el archivo FXML con SceneBuilder solo tenemos que hacer click derecho y seleccionar la opción **Open in Scene Builder**.

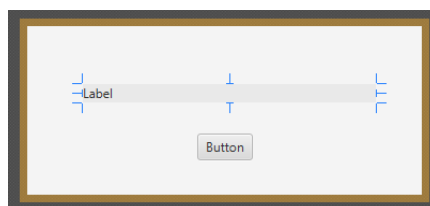


Si configuraste de forma correcta SceneBuilder, se abrirá la aplicación. También podemos editar directamente el código desde el IDE, pero es más sencillo trabajar con Scene Builder que con XML puro.

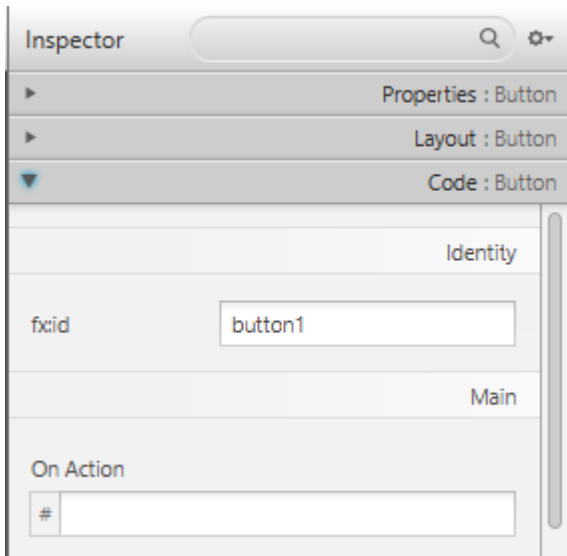
1.2.3. Creando nuestra primera aplicación JavaFX

Sigue los pasos:

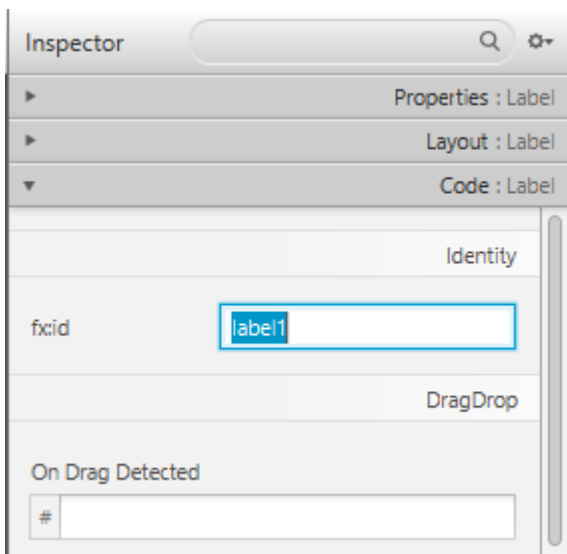
1. Elimina el VBox con sus componentes.
2. Añade un AnchorPane
3. Añade un Botón y un Label tal y como se muestra:



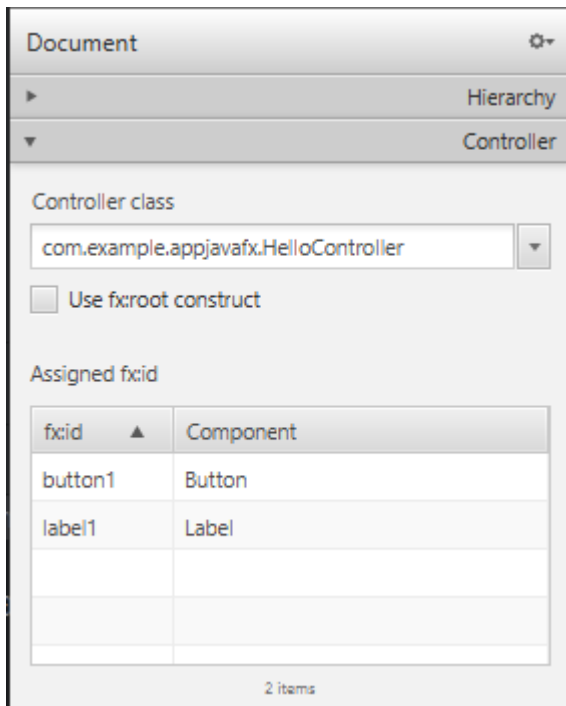
4. Añade un fx:id al botón (button1)



5. Añade un fx:id a la label (label1)



6. Selecciona el AnchorPane y define su clase controladora (en el menú de la izquierda)



7. Guarda los cambios, vuelve a IntelliJ y comprueba el archivo hello-view.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.AnchorPane?>

<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
  <children>
    <Button fx:id="button1" layoutX="159.0" layoutY="100.0" mnemonicParsing="false" text="Button" />
    <Label fx:id="label1" layoutX="52.0" layoutY="54.0" prefHeight="17.0" prefWidth="274.0" />
  </children>
</AnchorPane>
```

Como podemos comprobar ha cambiado.

8. Clicka en los id y pulsa en Create field 'button1' in 'HelloController'. Repite el proceso para cada componente con fx:id.

```
<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
  <children>
    <Button fx:id="button1" layoutX="159.0" layoutY="100.0" mnemonicParsing="false" text="Button" />
    <Label fx:id="label1" layoutX="52.0" layoutY="54.0" prefHeight="17.0" prefWidth="274.0" />
  </children>
</AnchorPane>
```

Unresolved fx:id reference

Create field 'button1' in 'HelloController' Alt+Mayús+Intro More actions... Alt+Intro

Una vez hecho si comprobamos la clase controladora (HelloController) tendremos los atributos creados con visibilidad pública. Cambia la visibilidad de los mismos de pública a privada:

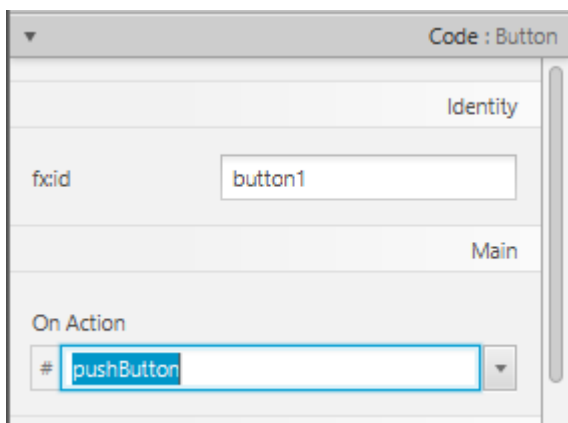
```
public class HelloController {  
    no usages  
    private Button button1;  
    no usages  
    private Label label1;  
}
```

Arregla el error de anotación de los componentes (si no los haces, el programa lanzará un error durante la ejecución):

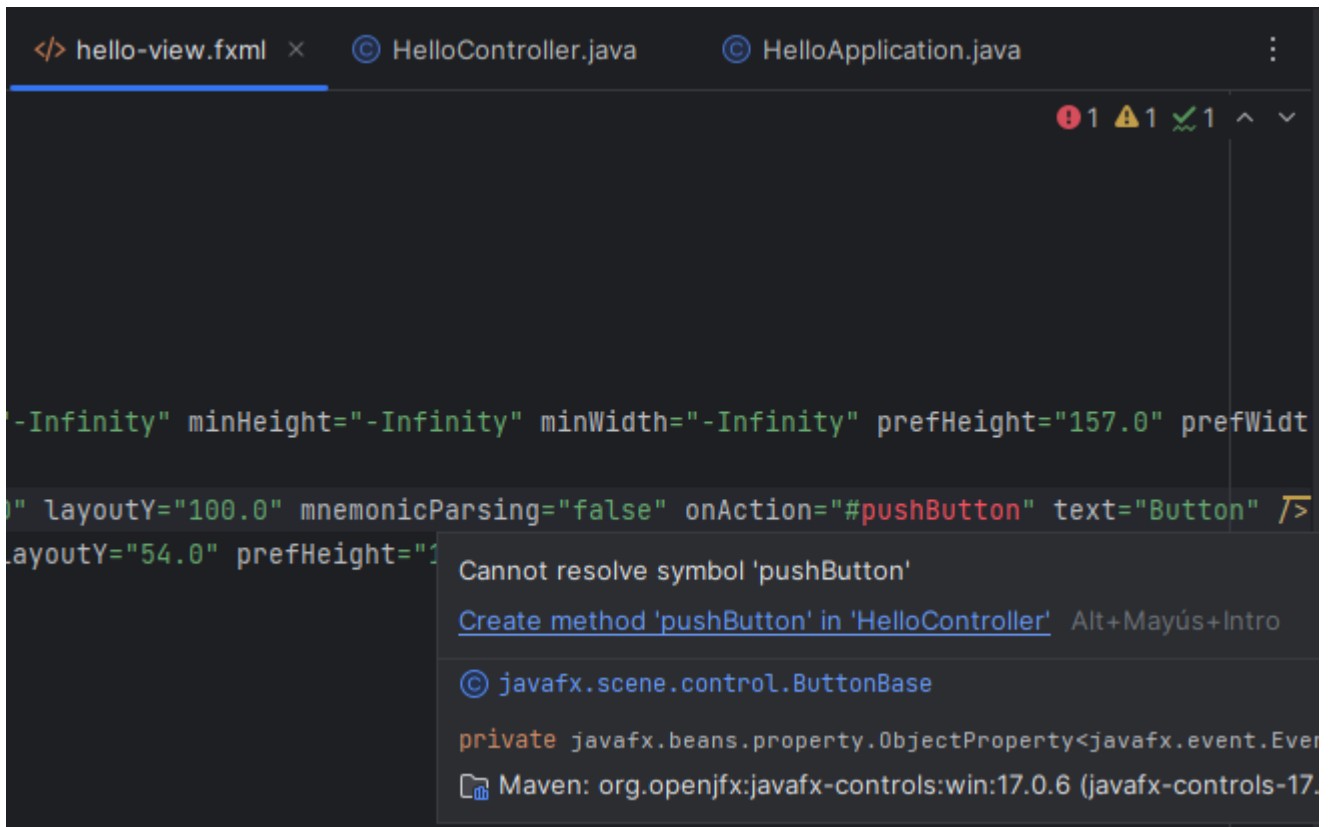
```
<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" style="background-color: #f0f0f0;">  
    <children>  
        <Button fx:id="button1" layoutX="159.0" layoutY="100.0" mnemonicParsing="false" text="Button" style="background-color: #f0f0f0;"></Button>  
        <Label fx:id="label1" layoutX="52.0" layoutY="54.0" prefHeight="17.0" prefWidth="274.0" style="background-color: #f0f0f0;"></Label>  
    </children>  
</AnchorPane>
```

'label1' should be public or annotated with @FXML
[Annotate field 'label1' as '@FXML'](#) Alt+Mayús+Intro More actions... Alt+Intro
© com.example.appjavafx.HelloController
private Label label1
appJavaFX

9. Añade un método (pushButton) desde SceneBuilder para que se ejecute cuando se pulse el botón:



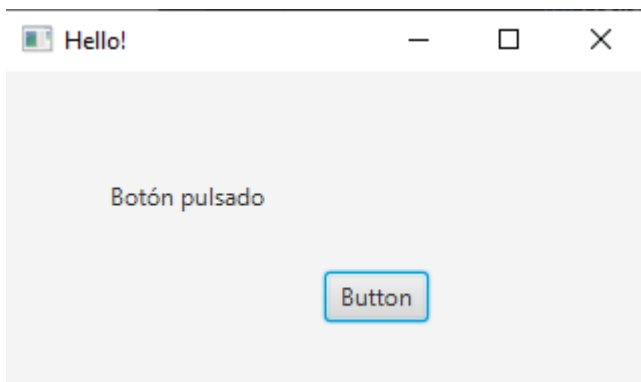
10. Crea el método en la clase Controlador de forma automática de la siguiente forma pulsando en 'Create method':



11. Añade el comportamiento para cuando el botón se pulse:

```
public class HelloController {
    @FXML
    private Button button1;
    @FXML
    private Label label1;
    1 usage
    public void pushButton(ActionEvent actionEvent) {
        label1.setText("Botón pulsado");
    }
}
```

12. Ejecuta la aplicación (Shift+F10)



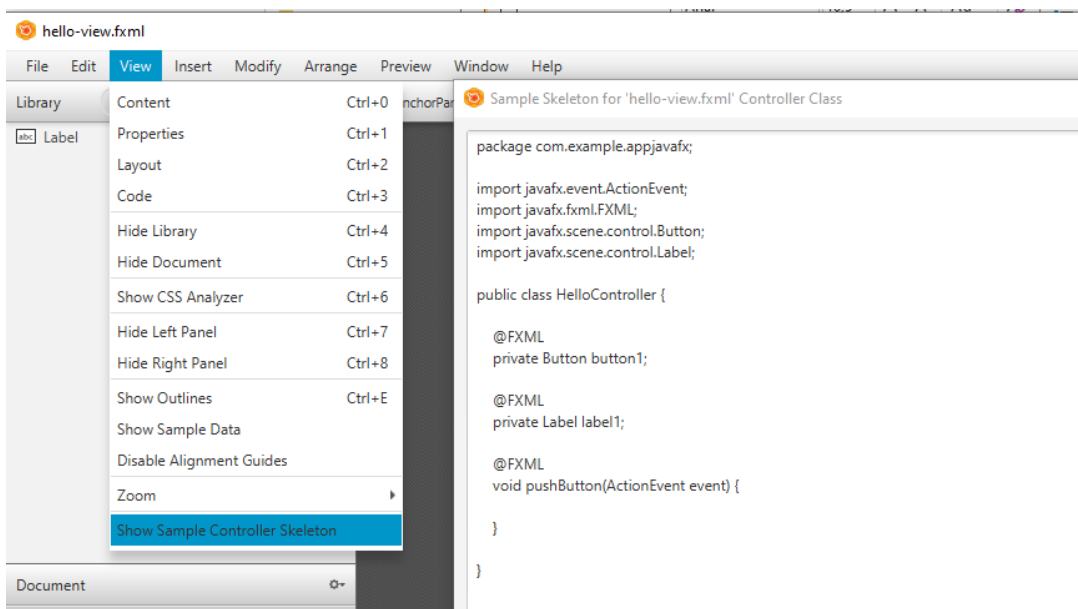
1.2.4. La clase Controlador

Una clase controladora se asocia a una vista (FXML) y vincula los controles y métodos de eventos de dicha vista a su código Java mediante la anotación `@FXML` a su código Java. Puedes ver qué clase de controlador está asociado a una vista mirando el campo **fx:controller** en la primera etiqueta XML.

Puedes vincular un control de vista en el controlador utilizando el mismo nombre de variable que su atributo **fx:id** en el archivo FXML y anteponiendo la anotación `@FXML`. Si quieres vincular un método para un evento declarado en el controlador FXML (como en el ejemplo anterior `→onAction="#pushButton"`), crea un método con el mismo nombre (sin el #) y la `@FXML` antes.

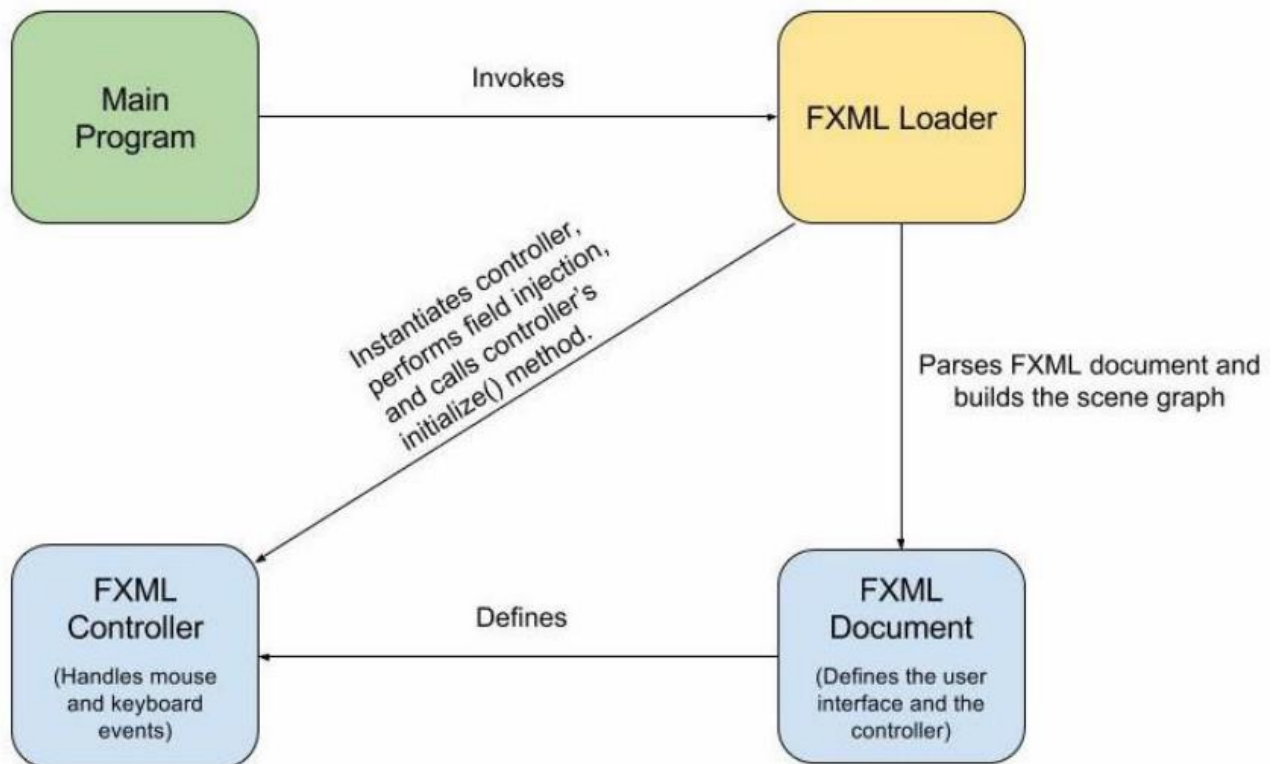
1.2.5. El esqueleto del Controlador

La forma más sencilla de escribir el código del controlador, y no tener que estar corrigiendo errores del archivo fxml en IntelliJ, es utilizar la opción **Ver** → **Mostrar esqueleto de controlador**:



De esta forma puedes copiar el código generado (esqueleto) en el controlador y completarlo con tu propio código.

1.3. Estructura de una aplicación JavaFX FXML



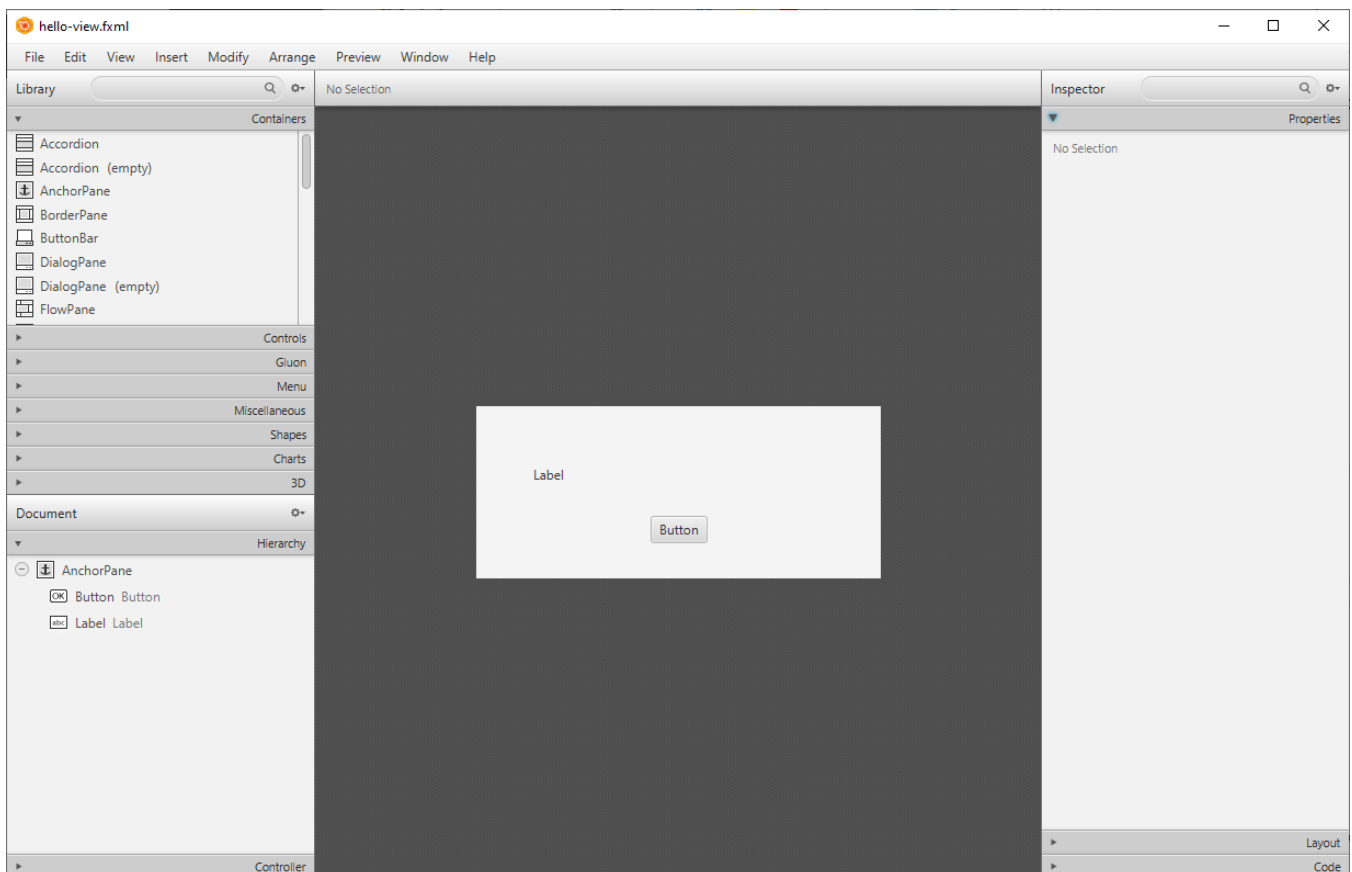
La figura anterior muestra la estructura de una aplicación JavaFX FXML típica.

Como se muestra en la figura, la interfaz de usuario de una aplicación FXML se define dentro de un documento FXML y toda la lógica para manejar los eventos de entrada se escribe dentro de una clase controladora.

La ejecución del programa comienza con la clase Main, que invoca al cargador FXML. El cargador FXML analiza el documento FXML, crea instancias de los nodos especificados en el documento FXML y construye el gráfico de la escena.

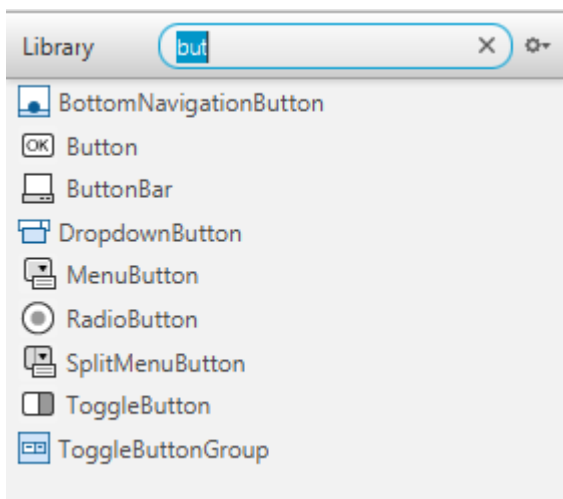
Después de construir el gráfico de la escena, el cargador FXML crea una instancia de la clase de controlador, inyecta los campos definidos en la clase del controlador con objetos instanciados del documento fxml y luego llama al método **initialize()** del controlador (Se verá más adelante).

1.4. Entendiendo Scene Builder

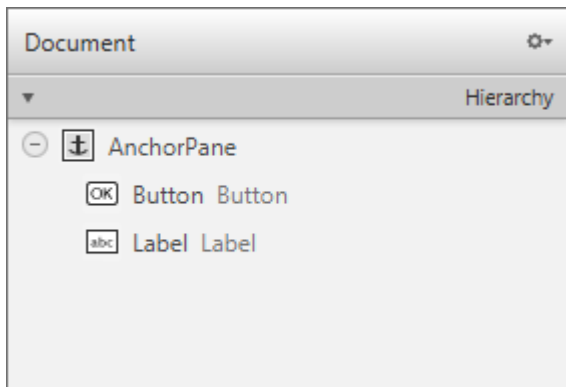


En la parte superior izquierda de la aplicación, tenemos la librería de JavaFX, desde la que podemos seleccionar el widget que necesitamos y arrastrarlo a la vista (escena).

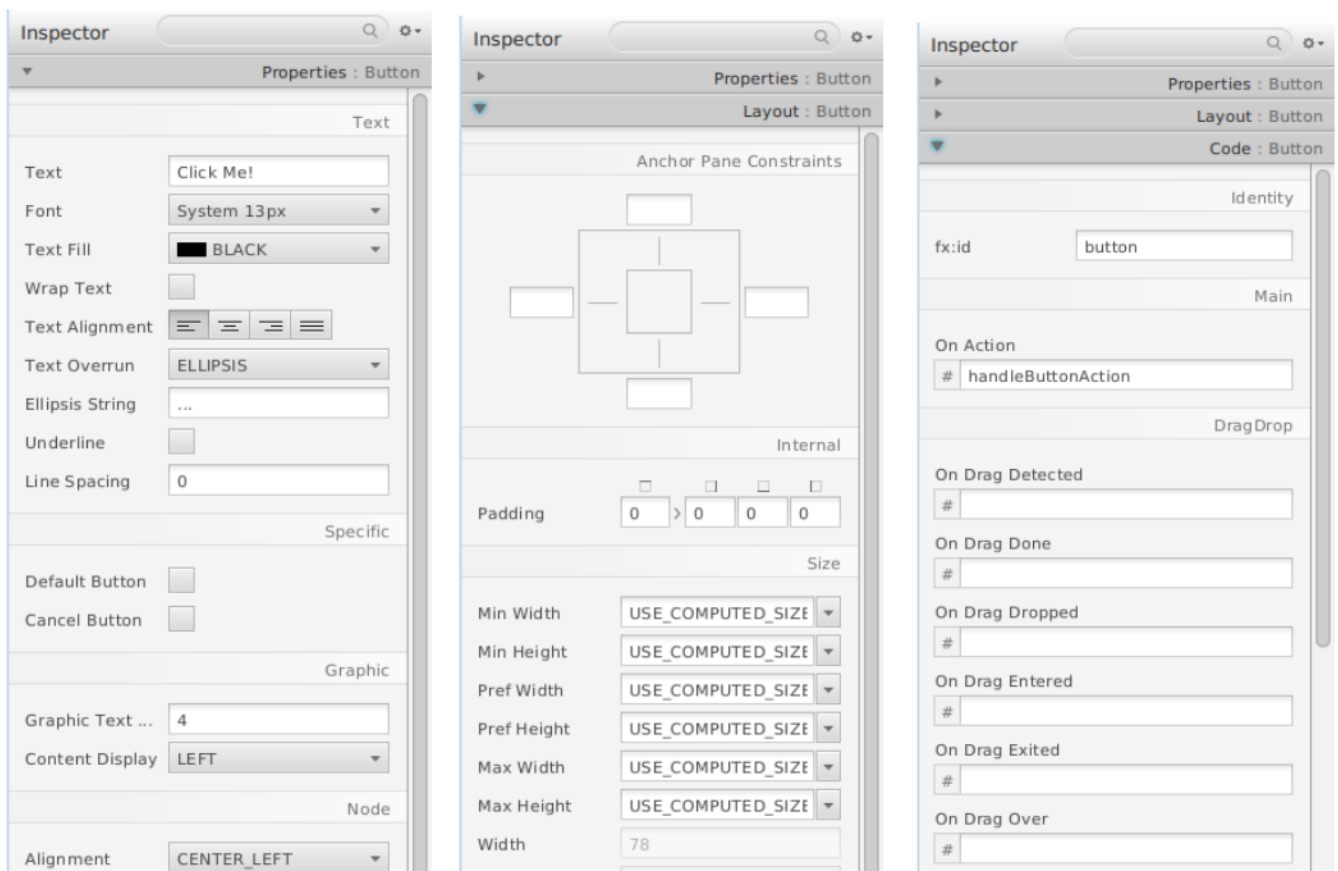
También puedes utilizar el cuadro de búsqueda para encontrar lo que estás buscando de una manera más rápida.



En la parte inferior izquierda, verás la jerarquía de objetos de la escena. Ahí también puedes arrastrar los elementos y controlar qué elementos están dentro de otros elementos.



En la parte derecha de la aplicación encontrarás el inspector del objeto seleccionado en ese momento, desde el que podrás cambiar sus propiedades (visuales y de código). Desde la pestaña de código, puedes especificar el id del objeto (fx:id) a utilizar en el código del controlador, y el método que se llamará cuando se active un evento (ejemplo: acción sobre botón) para ese objeto.



También puede utilizar el cuadro de búsqueda de la parte del inspector para buscar rápidamente cualquier cosa que desees.

2. Controles básicos y layouts

En esta sección aprenderemos algunos de los conceptos básicos más importantes para desarrollar aplicaciones JavaFX como, por ejemplo, algunos controles típicos que podemos utilizar y cómo organizarlos en la ventana.

2.1. Las clases Stage y Scene

2.1.1. La clase Stage

Como ya hemos visto en el anterior (y sencillo) ejemplo, cuando creamos una aplicación JavaFX la clase principal extiende la clase Application, y sobrescribe un método start que tiene como parámetro un objeto Stage. El objeto Stage es una referencia al contenedor principal de nuestra aplicación. Esto será una ventana en sistemas operativos como Linux, Windows o Mac OS X, pero puede ser la pantalla completa si nuestra aplicación se ejecuta en un smartphone o una tablet.

La clase Stage proporciona algunos métodos útiles para cambiar algunas características (tamaño, comportamiento...). Algunos de los métodos más útiles son:

- **setTitle(String)**: sets the application title (it is visible in the upper bar of the window).
- **setScene(Scene)**: sets our application scene (where all the controls will be placed). We will learn later that there can be more than one scene in a stage.
- **show**: makes the application (stage) visible, and keeps on running next instructions
- **showAndWait**: makes the application (stage) visible, and waits until it is closed before going on.
- **setMinWidth(double), setMaxWidth(double)**: set the minimum and maximum width (respectively) of the window, so that we will not be able to resize it beyond these limits.
- **setMinHeight(double), setMaxHeight(double)**: set the minimum and maximum height (respectively) of the window, similar to the width methods seen before.
- **getMinWidth, getMaxWidth, getMinHeight, getMaxHeight**: get the maximum or minimum width or height of the application.
- **setFullScreen(boolean)**: sets if our application will run in full screen mode (so it will not be resizable, and there will not be any upper bar), or not.
- **setMaximized(boolean)**: sets if our application is maximized or not.
- **setIconified(boolean)**: sets if our application is iconified (minimized) or not.
- **setResizable(boolean)**: sets if our application is resizable or not.

<https://openjfx.io/javadoc/12/javafx.graphics/javafx/stage/Stage.html>

Por ejemplo, con estas líneas dentro del método `start` podemos definir el título de la ventana, y el tamaño máximo y mínimo para nuestra ventana (si la redimensionamos):

```
stage.setTitle("Hello World");  
  
stage.setMinimumWidth(200);  
  
stage.setMaximumWidth(500);  
  
stage.setMinimumHeight(100);  
  
stage.setMaximumHeight(400);
```


2.1.2. La clase Scene

Cada programa JavaFX tiene (al menos) un objeto Scene para contener todos los controles de la aplicación. Cuando lo creamos, necesitamos especificar su nodo principal (el que FXMLLoader devuelve cuando parseamos un FXML):

```
Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));

Scene scene = new Scene(root);

stage.setScene(scene);
```

También podemos encontrar algunos métodos útiles de la clase Scene como:

- **getWidth, getHeight:** gets the scene's current width and height
- **getX, getY:** gets the scene's current coordinates in the screen (referring its upper left corner)
- **setRoot (Parent):** sets a new layout manager as the main node for this scene.

Un Stage puede tener múltiples Scenes, y puede alternarlas llamando al método setScene como veremos más tarde.

<https://openjfx.io/javadoc/12/javafx.graphics/javafx/scene/Scene.html>

2.2. Algunos de los controles más útiles en JavaFX

Si echas un vistazo a la API de JavaFX, encontrarás un montón de controles que puedes utilizar en tus aplicaciones. Algunos de ellos, como TreeViews o Accordions apenas se utilizan, pero algunos otros como Buttons, Labels, TextFields... se utilizan en casi todas las aplicaciones. Nos vamos a centrar en estos controles, y a explicar brevemente cómo utilizarlos. La mayoría de estos controles pertenecen al paquete **javafx.scene.control**.

2.2.1. Label

Una etiqueta se utiliza para mostrar algún texto en la escena. Una vez que hemos puesto la etiqueta dentro de un layout, existen algunos métodos útiles en la clase Label, como **getText** o **setText**, para obtener/establecer el texto de la etiqueta.

```
@FXML private Label label;
```

2.2.2. Inicializando los componentes JavaFX

Initialize() es el método que se utiliza cuando se quiere interactuar con @FXML. Durante la construcción esas variables no se llenan por lo que no se puede interactuar con ellas por lo que JavaFX llamará al método initialize después de que todo esté configurado. En ese momento las variables están disponibles y pueden ser manipuladas:

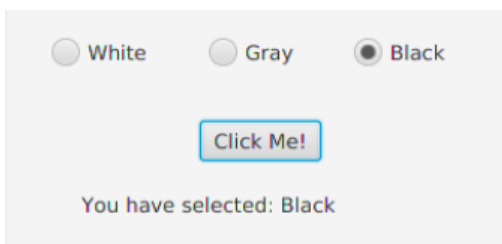
```
public class HelloController {  
  
    @FXML  
    private Label label1;  
  
    public void initialize() {  
        label1.setText("HOLA!");  
    }  
}
```

2.2.3. Button

Si queremos utilizar un botón, lo creamos con su texto (también existen otros constructores, similares a los que podemos encontrar para la clase Label):

```
@FXML private Button button;
```

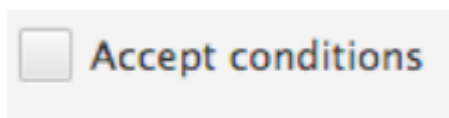
2.2.4. Radio Buttons



Los Radio Buttons son un conjunto de botones en los que sólo uno de ellos puede seleccionarse al mismo tiempo. Necesitamos definir un grupo (clase ToggleGroup), y añadir los botones de radio a la misma.

```
public class Controller {  
  
    @FXML  
    private ToggleGroup colorGroup;  
    @FXML  
    private Label label1;  
    @FXML  
    private Button button1;  
  
    public void pushButton(ActionEvent actionEvent) {  
        RadioButton selected = (RadioButton)colorGroup.getSelectedToggle();  
        label1.setText("You have selected: " + selected.getText());  
    }  
}
```

2.2.5. Checkboxes



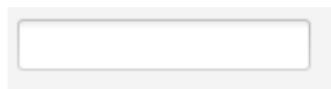
Una checkbox es una casilla de verificación que podemos marcar y desmarcar, alternativamente, cada vez que hacemos clic sobre ella.

```
@FXML private CheckBox acceptCheck;  
...  
acceptCheck.setSelected(true); // selected by default  
...  
if (acceptCheck.isSelected())  
{  
    ...  
}
```

También podemos definir si la casilla de verificación está inicialmente seleccionada o no con el método **setSelected** como se muestra en el código anterior, y utilizar el método **isSelected** para comprobar si está actualmente seleccionada.

2.2.6. Text fields

En cuanto a los campos de texto, los controles más comunes que podemos utilizar en nuestras aplicaciones son TextFields (para entradas de texto cortas, con una sola línea), y TextAreas (para textos más largos, con varias filas y columnas).



```
@FXML private TextField text;
```

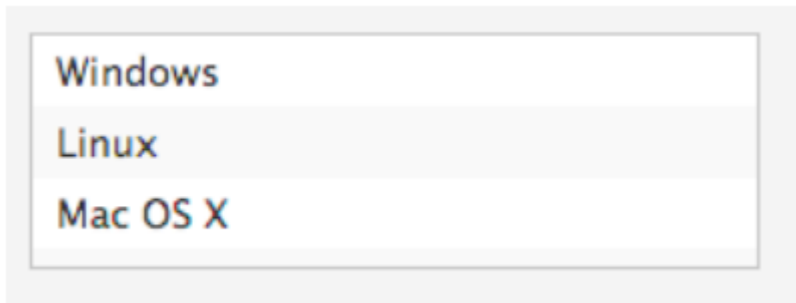
Existen algunos métodos como **getText** o **setText** para obtener y establecer el texto del control, respectivamente. También hay métodos como **setPromptText** (para establecer un texto que se borrará tan pronto como el usuario empiece a escribir algo en el campo de texto), o **setPrefColumnCount** (para establecer el número de caracteres que serán visibles en el campo de texto).

Si queremos utilizar un **TextArea**, lo creamos con un constructor vacío (o con un texto inicial), y luego tenemos dos métodos para establecer el número inicial de filas y columnas:

Cuando se utiliza un **TextArea**, el número de filas y columnas (caracteres en cada fila) se definirá en el archivo FXML. Hay otros métodos que pueden ser útiles, como **setWrapText** (establece si deben añadirse nuevas líneas cuando el texto excede la longitud del área de texto), o **getText/setText**, como en la clase TextField.

2.2.7. Lists

Tenemos dos tipos principales de listas que podemos usar en cualquier aplicación:



Listas con un tamaño fijo, en las que algunos elementos se muestran, y podemos desplazarnos por la lista para buscar el elemento(s) que deseemos. Para trabajar con estas listas en JavaFX, tenemos el control `ListView` que utiliza un `ObservableList` de elementos a mostrar. He aquí un ejemplo de cómo usarlo:

```
@FXML private ListView<String> list;
...
public void initialize()
{
    list.setItems(
        FXCollections.observableArrayList( "Windows", "Linux", "Mac OS X"));
    ...
}
```

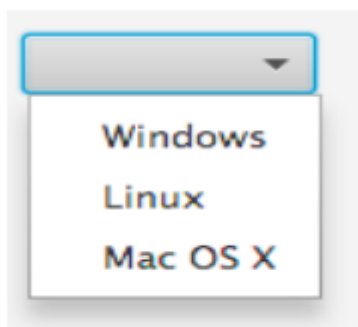
Necesitamos indicar el tipo de elementos que se mostrarán en la lista (en este ejemplo, trabajamos con `Strings`. Si trabajamos con objetos, se llamará al método `toString()` para mostrar los valores en la lista). A continuación, creamos la lista de objetos llamando a el método `FXCollections.observableArrayList`, y finalmente creamos el objeto `ListView` con estos elementos.

Podemos definir el modelo de selección de la lista, es decir, si queremos habilitar la selección de múltiples elementos, o si sólo queremos seleccionar un elemento. Podemos cambiar esta funcionalidad con estos métodos:

```
list.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
list.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);
```

Si queremos obtener el elemento o elementos seleccionados de la lista, tenemos que obtener el modelo de selección de la lista y, a continuación, llamar a **`getSelectedItem`** (para listas de selección única) o a **`getSelectedItems`** (para listas de selección múltiple). También tenemos las funciones **`getSelectedItemIndex`** y **`getSelectedItemIndices`**, si queremos obtener la posición o posiciones del elemento o elementos seleccionados, en lugar de sus valores.

```
String element = myList.getSelectionModel().getSelectedItem();
```



En las **listas desplegadas**, sólo se muestra un elemento, y podemos elegir cualquier otro elemento desplegando la lista. Si queremos utilizar este tipo de listas en nuestras aplicaciones JavaFX, podemos elegir entre las clases **`ChoiceBox`** y **`ComboBox`**. Las diferencias entre ellas son bastante sutiles, aunque `ComboBox` es más apropiada cuando trabajamos con listas grandes.

Tanto si trabajamos con `ChoiceBox` como con `ComboBox`, creamos las listas de forma muy similar manera a la mostrada para `ListView`: necesitamos un `ObservableList` para establecer los elementos y luego creamos la lista con ellos.

```
@FXML private ChoiceBox<String> list;
...
list.setItems(
    FXCollections.observableArrayList( "Windows", "Linux", "Mac OS X"));
```

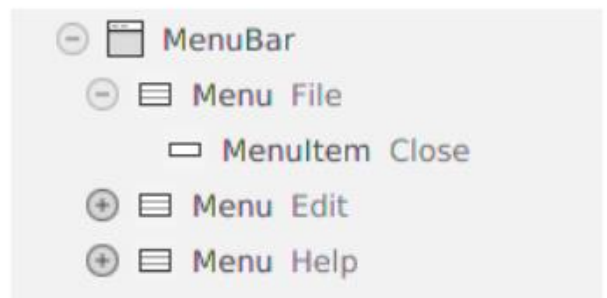
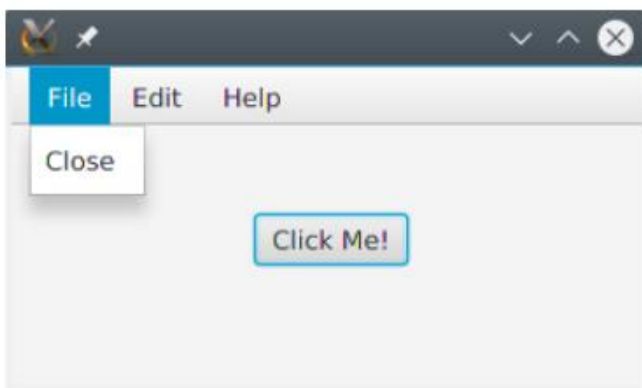
Si queremos obtener el elemento seleccionado de la lista, podemos utilizar el método **getValue** (y el método **setValue** para establecer el elemento seleccionado, si queremos).

Hay otras formas de añadir elementos a las listas, como llamar al método **getItems** y luego llamar al método **addAll** para añadir más elementos:

```
myList.getItems().addAll("Android", "iOS");
```

2.2.8. Menus

Como en muchas aplicaciones de escritorio, podemos añadir un menú a nuestra aplicación JavaFX (esto no es habitual cuando desarrollamos una aplicación móvil). El patrón que solemos seguir es poner una barra de menú (con menús por defecto dentro que podemos editar), definir las categorías (Menú), y añadir elementos de menú a las categorías.



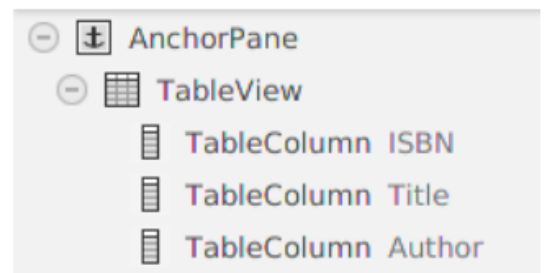
Como puedes observar, necesitamos usar tres elementos distintos:

- **MenuBar** para definir la barra donde todos los menús serán colocados.
- **Menu** para definir las categorías de los menús.
- **MenuItem** para definir cada elemento de nuestros menús. Si pulsamos sobre un elemento, podemos definir algún código asociado a esa acción, como veremos al hablar de los eventos. En el ejemplo anterior hemos definido un elemento de menú llamado "Cerrar" dentro del menú Archivo.
 - También existen algunos subtipos de MenuItem, como **CheckMenuItem** (elementos que pueden marcarse/desmarcarse, como las casillas de verificación), o **RadioMenuItem** (grupos de (grupos de elementos de los que sólo se puede marcar uno al mismo tiempo, como los botones de radio). También podemos utilizar un **SeparatorMenuItem** para crear una línea de separación entre grupos de elementos de menú.

2.2.9. Tables

Existe un control muy útil que podemos utilizar cuando trabajamos con grandes cantidades de datos, o con algunos datos estructurados que necesitamos mantener visibles al mismo tiempo: las tablas. Este control nos permite organizar la información de forma que podemos mostrar diferentes registros en diferentes filas, y diferentes informaciones sobre un mismo registro en diferentes columnas.

Para manejar las tablas, utilizaremos el control **TableView**. Podemos definir las cabeceras de las columnas con la clase **TableColumn**. Por ejemplo, si queremos gestionar una lista de libros con su ISBN, título y autor, crearíamos una tabla como ésta:



Una vez que tengamos nuestra tabla creada, puede que necesitemos llenarla con algunos datos que hayamos almacenado previamente. Es recomendable crear una clase (una clase pública) que almacene cada objeto de la tabla (cada fila), y asociar cada columna a un atributo de esta clase. En nuestro ejemplo anterior, si queremos trabajar con libros, podemos crear una clase Libro con tres atributos (ISBN, título y autor), y sus correspondientes getters y setters:

```
public class Book {
    String isbn, title, author;
    public Book (String isbn, String title, String author)
    {
        this.isbn = isbn;
        this.title = title;
        this.author = author;
    }
    public String getIsbn()
    {
        return isbn;
    }
    public void setIsbn(String isbn)
    {
        this.isbn = isbn;
    }
    //... Rest of getters and setters
}
```

Ahora debemos enlazar en el controlador la vista de la tabla y sus columnas. Deberías notar que definimos el tipo de elementos que va a contener la tabla (Book), y para cada columna, también especificamos el tipo de datos que mostrará (String, Integer,...).

```

@FXML private TableView<Book> table;
@FXML private TableColumn<Book, String> colIsbn;
@FXML private TableColumn<Book, String> colTitle;
@FXML private TableColumn<Book, String> colAuthor;

public void initialize()
{
    // Text to show when the table is empty
    table.setPlaceholder(new Label("No items to show..."));
}

```

También adjuntamos cada propiedad de libro a un objeto *TableColumn* determinado:

```

colIsbn.setCellValueFactory(new PropertyValueFactory("isbn"));
colTitle.setCellValueFactory(new PropertyValueFactory("title"));
colAuthor.setCellValueFactory(new PropertyValueFactory("author"));

```

Es importante que utilicemos el mismo nombre en el parámetro de PropertyValueFactory que un getter correspondiente, pero sin el prefijo "get". Por ejemplo, si creamos un método llamado `getTotalPrice()`, debemos utilizar el nombre "totalPrice" en la columna.

Una vez definida nuestra clase objeto, podemos crear o cargar una colección de libros con nuestra clase `FXCollections`:

```

ObservableList<Book> data = FXCollections.observableArrayList(
    new Book("111111", "Ender's game", "Orson Scott Card"),
    new Book("222222", "The adventures of Tom Sawyer", "Mark Twain"),
    new Book("333333", "The never ending story", "Michael Ende")
);
table.setItems(data);

```

Visualizaremos algo como esto:

ISBN	Title	Author
111111	Ender's game	Orson Scott Card
222222	The adventures of Tom Sawyer	Mark Twain
333333	The never ending story	Michael Ende

Añadir nuevas filas a una *TableView* es tan sencillo como añadir nuevos objetos a la lista asociada a ella. En nuestro ejemplo, cada vez que añadamos o eliminemos datos a nuestra lista observable (datos variables), la *TableView* se actualizará automáticamente.

Podemos utilizar el método **add** de la interfaz *ObservableList* para añadir elementos a la lista (por ejemplo, cuando hacemos clic en un botón):

```
@FXML
private void buttonAction(ActionEvent event)
{
    data.add(new Book("44444", "Misery", "Stephen King"));
}
```

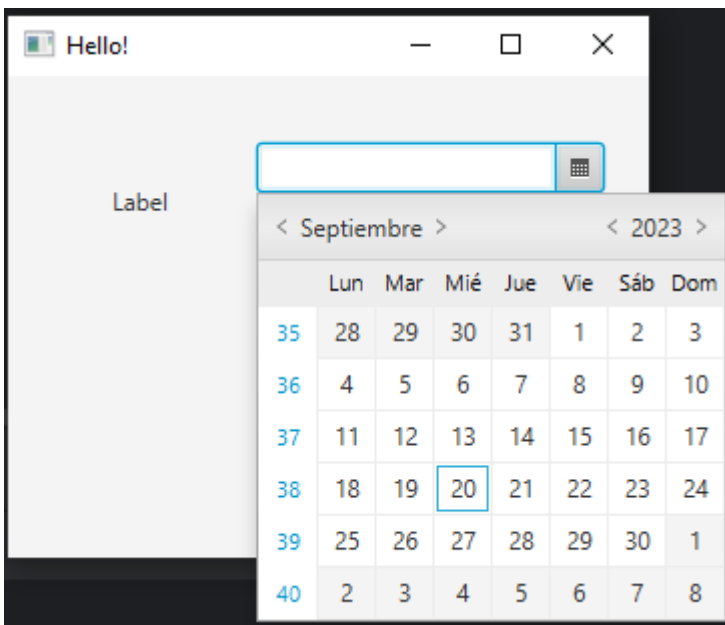
También podemos utilizar el método **remove** para eliminar un elemento (o un conjunto de elementos) de una posición (o rango). Esta línea elimina el último elemento de la lista/tabla:

```
data.remove(data.size()-1);
```

Para comprobar qué elemento (fila) de la tabla está actualmente seleccionado, podemos utilizar los mismos métodos que utilizamos con el control ListView. Típicamente, obtendremos el índice del elemento actualmente seleccionado, y obtendremos el objeto completo del arraylist:

```
int index = table.getSelectionModel().getSelectedIndex();
Book selBook = data.get(index);
```

2.2.10. Date picker



El control DatePicker JavaFX permite al usuario seleccionar una fecha de un calendario emergente que aparece cuando pulsamos su botón derecho.

Podemos acceder al TextField dentro del control DatePicker mediante el método `getEditor()`. Una vez que tenemos acceso a ese campo de texto, podemos obtener su valor fácilmente. También podemos obtener el objeto `LocalDate` que representa la fecha seleccionada utilizando `getValue()`.


```

@FXML private DatePicker datepicker;

@FXML
private void dateChangeAction(ActionEvent event)
{
    // This will execute every time a new date is selected
    LocalDate date = datepicker.getValue();
}

public void initialize()
{
    datepicker.setShowWeekNumbers(false); // Do not show week numbers
    datepicker.setValue(LocalDate.now()); // By default today is selected
    datepicker.setEditable(false); // The user can't edit the text field
}

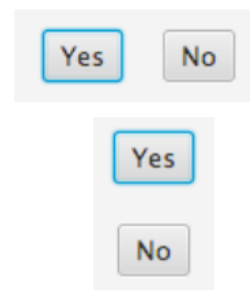
```

2.3. Organizando los controles. El paquete "layout"

Una de las tareas más tediosas a la hora de diseñar una aplicación gráfica es ordenar los controles en la ventana. De esta tarea se encarga el gestor de disposición o panel de disposición. Ahora vas a saber que existen varios tipos y dependiendo de tu elección(es), los controles pueden disponerse de muchas maneras diferentes.

Algunos de los paneles de disposición más comunes en JavaFX son:

- **HBox:** arranges controls horizontally, one next to the other.
- **VBox:** arranges controls vertically, one above/below the other.
- **FlowPane:** arranges controls next to each other until there is no more space (vertically or horizontally). Then, it goes to next row (or column, depending on its configuration) to keep on arranging more controls.
- **BorderPane:** this layout divides the pane into five regions: top, bottom, left, right and center, and we can add a control (or a pane with some controls) in each region.
- **AnchorPane:** this layout enables you to anchor nodes to the top, bottom, left side, right side, or center of the pane. As the window is resized, the nodes maintain their position relative to their anchor point. Nodes can be anchored to more than one position and more than one node can be anchored to the same position.



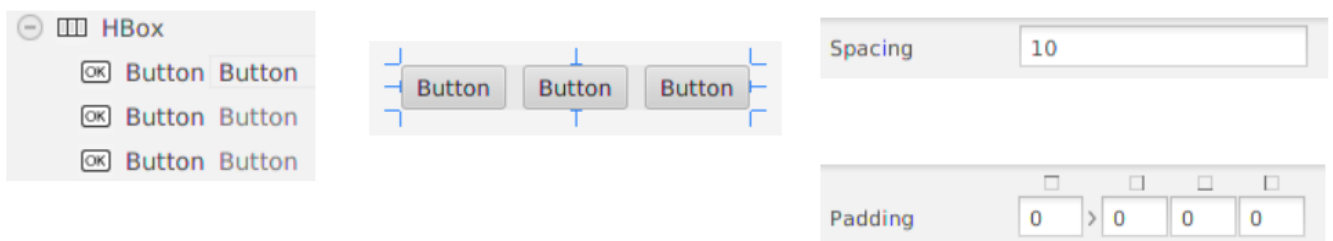
Existen otros tipos como GridPane (que crea un tipo de tabla para colocar los controles).

También puedes combinarlos como quieras. A continuación, veremos algunos ejemplos.

https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm

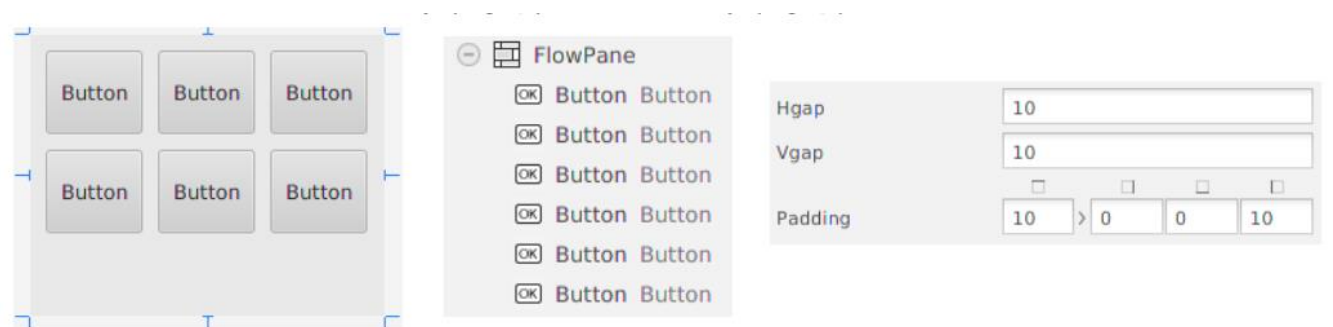
2.3.1. Trabajando con paneles Hbox y VBox

Estos paneles de diseño contienen uno o más nodos dispuestos verticalmente (VBox) u horizontalmente (HBox). Podemos establecer algunas propiedades en el editor como spacing (espacio entre elementos dentro del panel) o padding (espacio entre uno de los bordes del panel y los elementos que están dentro de él).



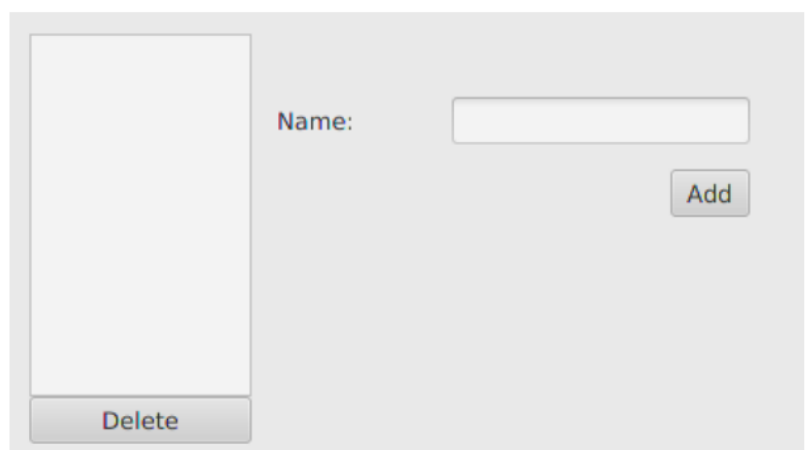
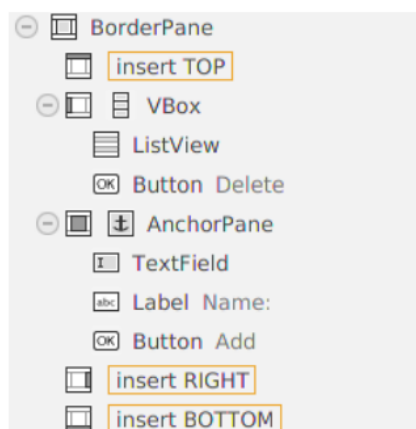
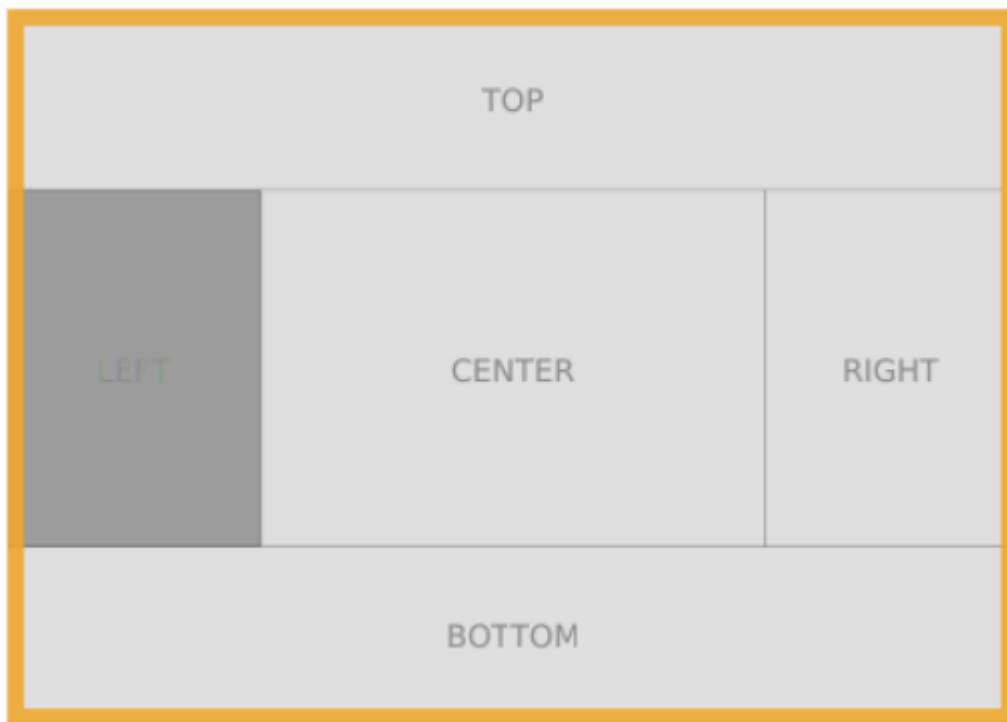
2.3.2. Trabajando con el FlowPane

La principal diferencia con HBox y VBox es que en un FlowPane podemos controlar el espacio entre elementos horizontalmente (Hgap) y verticalmente (Vgap).



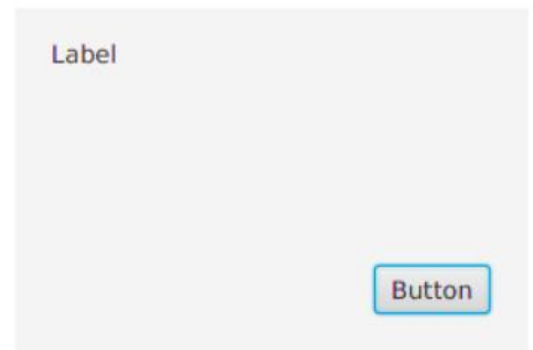
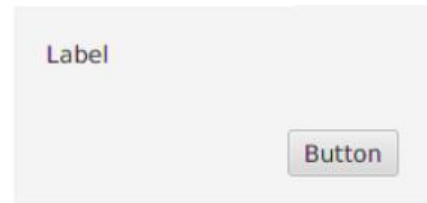
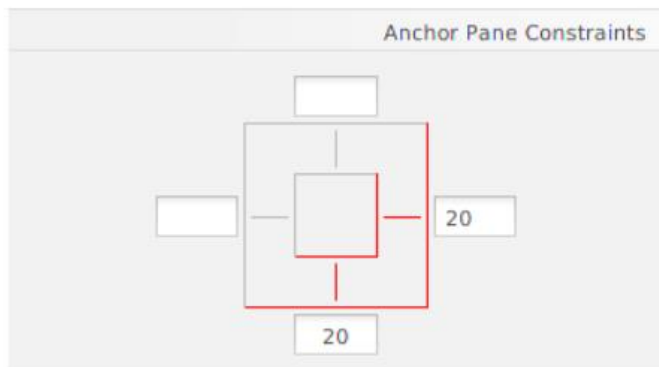
2.3.3. Trabajando con el BorderPane

Con un BorderPane, podemos disponer elementos en su interior en cinco lugares diferentes (podemos dejar algunos de ellos vacíos). Normalmente queremos insertar otros contenedores como MenuBar, VBox, HBox, FlowPane, etc. en esos lugares



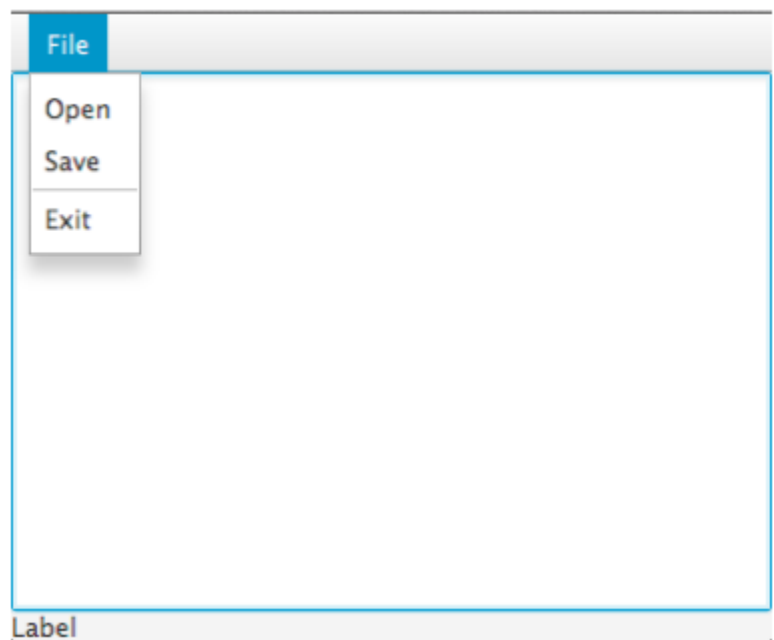
2.3.4. Trabajando con el AnchorPane

Un AnchorPane es el más flexible ya que nos permite colocar elementos en cualquier lugar. Podemos establecer una distancia fija de un elemento a uno (o más) de los lados del AnchorPane (arriba, derecha, inferior, izquierdo), de modo que cuando la ventana (y el AnchorPane dentro de ella) crece, el elemento siempre mantiene siempre la misma distancia al borde (o bordes) del contenedor.



Ejercicio 1

Crea un proyecto llamado **NotepadJavaFX**. Utiliza un `BorderPane` (400 pixels de ancho y 300 pixels de alto), y añade un menú en la parte superior, un `TextArea` en el centro y una etiqueta en la parte inferior. El menú debe tener un menú `File` con cuatro elementos de menú: `Open`, `Save`, un elemento separador y `Exit`. Debe parecerse a la imagen de la derecha.



Ejercicio 2

Crea un proyecto llamado **Calculadora** con la siguiente apariencia (utiliza los layouts que creas convenientes para conseguirlo):

1st number:

Choose operation:

2nd number:

Result:

La lista desplegable puede hacerse con un *ChoiceBox* o un *ComboBox*. Debe tener las opciones "+", "-", "*", "/" (los cuatro operadores aritméticos básicos).

3.Eventos

Si sólo añadimos controles a nuestra aplicación JavaFX (botones, etiquetas, cuadros de texto...) no podremos hacer nada más que clicar y escribir en ella. No habrá carga de ficheros, guardado de datos, ni ninguna operación con los datos que tecleemos o añadamos a la aplicación.

Para permitir que nuestra aplicación responda a nuestros clics y tecleos, necesitamos definir manejadores de eventos. Un evento es algo que ocurre en nuestra aplicación. Hacer clic con el ratón, pulsar una tecla, o incluso pasar el ratón sobre la ventana de la aplicación, son ejemplos de eventos. Un manejador de eventos es un método (u objeto con un método) que responde a un evento ejecutando algunas instrucciones.

Por ejemplo, podemos definir un manejador que, cuando un usuario pulse un botón determinado, tome los valores numéricos de unos cuadros de texto, los sume y muestre el resultado.

3.1. Tipos de eventos principales

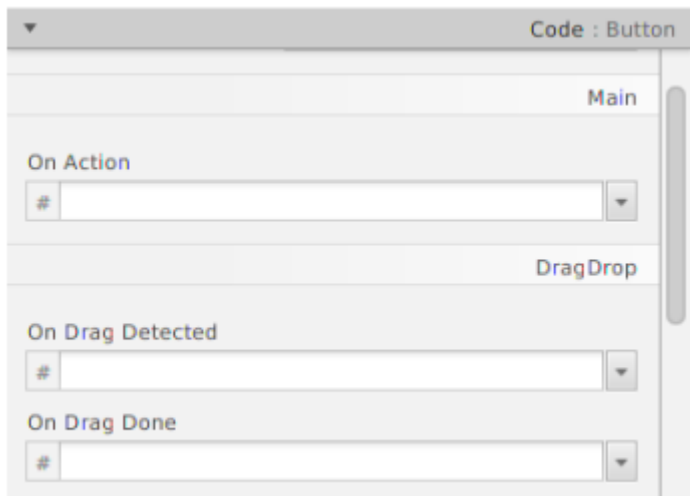
Cada evento producido en nuestra aplicación es una subclase de la clase Evento. Algunos de los tipos (subclases) de eventos más comunes son:

- **ActionEvent:** típicamente se crea cuando el usuario hace clic en un botón o en un elemento de menú (y también cuando se desplaza hasta el botón o elemento de menú y pulsa la tecla Intro).
- **KeyEvent:** se crea cuando el usuario pulsa una tecla.
- **MouseEvent:** se crea cuando el usuario hace algo con el ratón (pulsar un botón, mueve el ratón...).
- **TouchEvent:** se crea cuando el usuario toca algo en la aplicación (en dispositivos que permiten la entrada táctil)
- **WindowEvent:** se crea cuando el estado de la ventana cambia (por ejemplo, se maximiza, minimiza o cierra).

3.2. Definiendo manejadores y conectándolos con eventos

Ahora que sabemos qué es un evento, y sus principales subtipos, veamos cómo definir manejadores para controlarlos. Como verás, hay muchas formas de definir manejadores, y puedes elegir cualquiera de ellos dependiendo de las características de tu aplicación.

3.2.1. Definiendo manejadores en FXML con Scene Builder

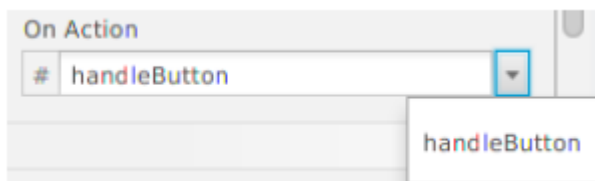


La mayoría de los eventos más comunes (pero no todos) pueden conectarse a un controlador de eventos con Scene Builder (FXML). Para ello, seleccionamos el nodo y a la derecha (pestaña código) veremos los diferentes tipos de eventos que podemos vincular a un método en el controlador.

Para manejar un evento, creamos un método en el controlador. Este método debe tener la anotación `@FXML` y recibirá un objeto derivado de `Event` (como `ActionEvent`) con el detalle del evento.

```
@FXML
private void handleButton(ActionEvent event)
{
    // Code
}
```

Una vez que creamos un método en el controlador, podemos seleccionarlo desde el Scene Builder y adjuntarlo a un evento. El evento más común es `Action`, que se dispara cuando un usuario hace clic (o pulsa enter o espacio cuando el control tiene el foco) en un botón o control similar.



3.2.2. Definiendo eventos con código

Para definir un manejador por código necesitamos un objeto que implemente la interfaz `EventHandler` (con un método llamado `handle` que reciba un objeto derivado de `Event`). De esta forma definimos un evento de acción para un botón por código (sin Scene Builder):

```

public class ExampleController
{
    @FXML private Label label;
    @FXML private Button button;

    public void initialize()
    {
        button.setOnAction(new EventHandler<ActionEvent>()
        {
            @Override
            public void handle(ActionEvent event)
            {
                label.setText("Hello World!");
            }
        });
    }
}

```

Dado que se trata de una interfaz funcional, podemos utilizar una expresión lambda para implementarla, por lo que el código puede ser más corto:

```

public class ExampleController
{
    @FXML private Label label;
    @FXML private Button button;

    public void initialize()
    {
        button.setOnAction((ActionEvent event) -> {
            label.setText("Hello World!");
        });
    }
}

```


3.3. Ejemplos

3.3.1. **ActionEvent** para cambiar el texto de un botón

Esta aplicación tiene un botón en el centro, y el texto del botón cambia cada vez que pulsamos sobre él. Si el texto del botón es "Hola", cambiará a "Adiós" y viceversa:

```
public class ExampleController
{
    @FXML private Button button;

    public void initialize()
    {
        button.setOnAction((ActionEvent event) -> {

button.setText(button.getText().equals("Hello")?"Goodbye":"Hello")
;
        });
    }
}
```

3.3.2. **MouseEvent** para cambiar el color de fondo de una etiqueta

Este ejemplo tiene una etiqueta en el centro de la ventana, y si pasamos el ratón sobre ella, ésta cambia su color de fondo a rojo, mientras que, si movemos el ratón fuera de la etiqueta, ésta recupera su color de fondo original.

```
public class ExampleController
{
    @FXML private Label label;

    public void initialize()
    {
        label.setOnMouseEntered(
            (e) -> label.setStyle("-fx-background-color:green;"));
        label.setOnMouseExited(
            (e) -> label.setStyle("-fx-background-color:none;"));
    }
}
```

Hemos necesitado dos controladores de eventos, uno para el ratón cuando entra en la etiqueta (que cambia el color de fondo a verde), y otro para el ratón cuando sale de la etiqueta (no pone color de fondo). Mira cómo usamos algún tipo de estilo CSS para cambiar el color de fondo. Hablaremos de esto más adelante en esta unidad.

3.3.3. KeyEvent para clonar el texto de un campo en una etiqueta

En este ejemplo se usa un KeyEvent para capturar cada tecla pulsada en un text field y copiar su contenido en una etiqueta.

```
public class ExampleController
{
    @FXML private TextField textField;
    @FXML private Label label;

    public void initialize()
    {
        textField.setOnKeyTyped(e -> label.setText(textField.getText()));
    }
}
```

Cuando escribamos algo en el campo de texto, el método `setOnKeyTyped` se disparará, y por tanto se ejecutará la función lambda. Simplemente copia el texto actual del campo de texto a la etiqueta.

3.3.4. ChangeEvent en una ListView cuando la selección cambia

En este ejemplo, cuando el usuario selecciona un elemento en el ListView. La aplicación mostrará en una etiqueta qué elemento está seleccionado actualmente. Podemos hacerlo de dos formas diferentes.

En el primer ejemplo, el evento recibirá el elemento seleccionado anteriormente y el elemento seleccionado actualmente (utilizando una clase anónima):

```
public class ExampleController
{
    @FXML private ListView<String> listView;
    @FXML private Label label;

    public void initialize()
    {
        listView.getItems().addAll("House", "Car", "Speaker", "Computer");
        listView.getSelectionModel().selectedItemProperty().addListener(
            new ChangeListener<String>()
            {
                @Override
                public void changed(ObservableValue<? extends String> obs,
                                    String oldValue, String newValue)
                {
                    if(newValue != null)
                    {
                        label.setText("You have selected: " + newValue);
                    } else {
                        label.setText("Nothing is selected");
                    }
                }
            }
        );
    }
}
```

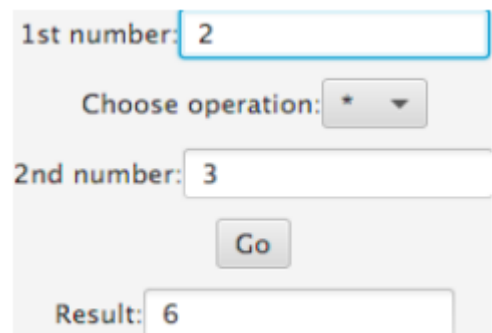
También podemos crear un evento que recibe el índice previo y el actual (esta vez usando una expresión lambda):

```
listView.getSelectionModel().selectedIndexProperty().addListener(  
    (observable, oldIndex, newIndex) -> {  
        if (newIndex.intValue() != -1)  
        {  
            label.setText("You have selected: " +  
                listView.getSelectionModel().getSelectedItem());  
        } else {  
            label.setText("Nothing is selected");  
        }  
    }  
);
```

Ejercicio 3

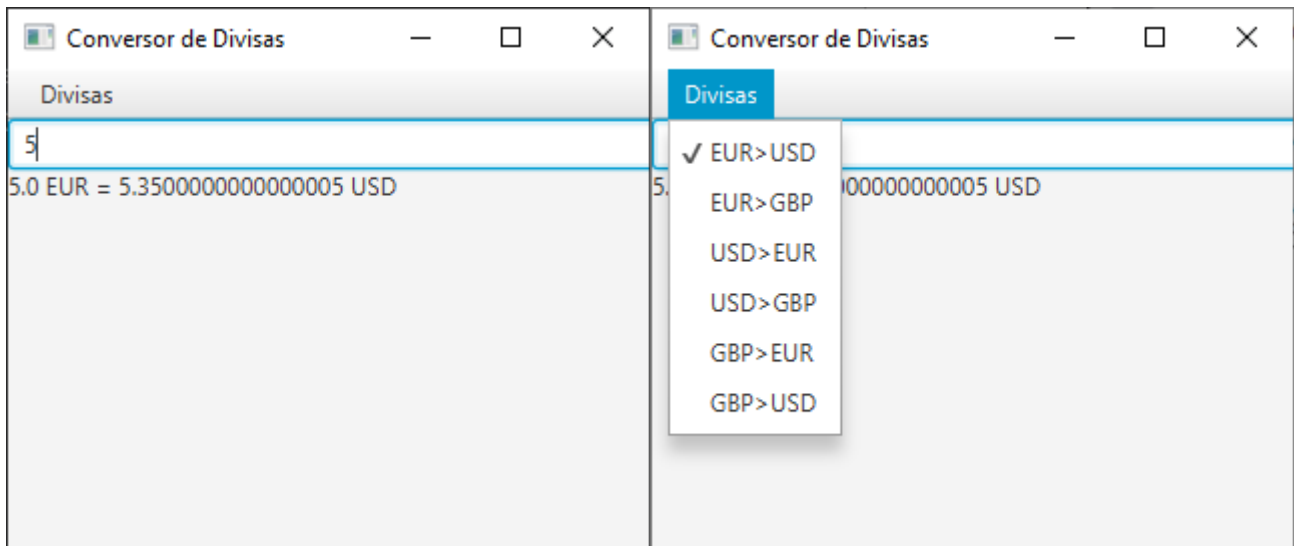
Crema un proyecto llamado **CalculatorEvent** que será una copia del proyecto Calculadora del Ejercicio 2. Añade un *ActionEvent* al botón para que, al pulsarlo, obtenga los números escritos en los dos primeros campos de texto, y el tipo de operación del cuadro de elección, y luego imprima el resultado en el último campo de texto. Por ejemplo, si el 1er número es 2, el 2do número es 3 y la operación es "*", el resultado debería mostrar 6.

Si no hay escrito ningún número, debes asumir que hay un 0.



Ejercicio 4

Crear un proyecto llamado **CurrencyConverter** que nos permita convertir entre tres tipos diferentes tipos de divisas: euro (EUR), dólar (USD) y libra esterlina (GBP). Habrá un menú para elegir una de las seis combinaciones posibles, utilizando *RadioMenuItems*: EUR>USD(opción por defecto), EUR>GBP, USD>EUR, USD>GBP, GBP>EUR y GBP>USD. Debajo habrá un campo de texto y una etiqueta. Cada vez que escribamos algo en el campo de texto, el programa debe convertir el importe automáticamente a la divisa dada, y mostrar el resultado en la etiqueta. Por ejemplo, si hemos elegido EUR>GBP, y sabemos que 1 EUR = 0,8 GBP, entonces cuando escribimos "12" en el campo de texto, el programa debe tener este aspecto:



Para completar la aplicación, añade un *ActionEvent* a cada *RadioButtonItem* que borre el texto del campo de texto y de la etiqueta, para iniciar una nueva conversión con nuevas monedas.

Para ayudarte a terminar el programa, asume que los cambios de moneda son los siguientes:

- 1 EUR = 1.07 USD
- 1 EUR = 0,87 GBP
- 1 USD = 0,82 GBP