

# DESARROLLO DE APLICACIONES WEB

## Unidad 12

### Interfaz gráfica de usuario

*1r DAW*

*IES La Mola de Novelda*

*Departament d'informàtica*

# ÍNDIX

1.- JavaFX.....	4
1.1.- Introducción.....	4
1.2.- Funcionamiento.....	4
1.3.- Características.....	4
2.- Entorno de trabajo.....	5
3.- Estructura de una Aplicación desarrollada con JavaFX.....	11
3.1.- Clase Application.....	11
3.2.- Estructura de una aplicación JavaFX.....	11
3.2.1.- Clase Stage.....	11
3.2.2.- Clase Scene.....	12
3.2.3.- Nodos - Gráficos de Escena.....	12
3.3.- Ejemplo práctico.....	13
4.- Ciclo de vida clase Application - Life cycle JAVA FX.....	13
4.1.- Ciclo de vida de clase Application.....	13
4.2.- Ciclo de vida clase Application - Métodos.....	13
4.3.- Ejemplo práctico.....	14
5.- Modelo – Vista – Controlador.....	15
5.1.- MVC.....	15
5.1.1.- Introducción.....	16
5.1.2.- Estructura MVC.....	16
5.1.3.- Funcionamiento y flujo del MVC.....	17
5.1.4.- Ejemplo práctico.....	17
6.- Scene Builder.....	18
7.- Práctica - 1.....	19
8.- Controller Initializable (Suma de dos números).....	23
8.1.- Práctica – 1 modificada.....	25
9.- Suma, resta, multiplicación y división de dos números.....	27
10.- Lista de Personas.....	27
11.- Lista de Personas . Modificar - eliminar.....	28
12.- Lista de Personas. Abriendo ventanas.....	29
13.- Modificar-Eliminar desde otra ventanas.....	29

14.- Filtrar y Ordenar registros.....	30
15.- Navegar entre ventanas.....	30
16.- Comunicación entre ventanas.....	31
17.- Juego de la bola.....	31

# Unidad : J<sub>AVA</sub>FX

## 1.- J<sub>AVA</sub>FX

### 1.1.-INTRODUCCIÓN

JavaFX es un conjunto de paquetes de gráficos y medios que permite a los desarrolladores diseñar, crear, probar, depurar e implementar aplicaciones de cliente enriquecidas que operan de manera consistente en diversas plataformas. Esta tecnología tiene muchas posibilidades, y no solo nos brinda soluciones para el desarrollo de aplicaciones de escritorio, sino en múltiples plataformas como web y móvil. Además, nos facilita la vida en cuanto al diseño de interfaces gráficas de usuario modernas.

### 1.2.-FUNCIONAMIENTO

JavaFX es una biblioteca de JAVA, por lo tanto para poder ejecutar las aplicaciones de JavaFX, deberemos tener **Java Runtime Environment (JRE)** y **JavaFX Runtime** instalados en nuestro equipo.

### 1.3.-CARACTERÍSTICAS

1. Acceso a la API de JAVA: JavaFX es una biblioteca de Java que consta de clases e interfaces que están escritas en código Java.
2. FXML Y Scene Builder: FXML es un lenguaje de marcado declarativo basado en XML para construir una interfaz de usuario de aplicación JavaFX. Un diseñador puede codificar en FXML o usar JavaFX Scene Builder para diseñar interactivamente la interfaz gráfica de usuario (GUI).
3. WebView: En ocasiones se necesita que nuestras aplicaciones tengan un módulo de software en el cual pudiéramos incrustar una página web sin necesidad de que el usuario de la aplicación tuviese que acceder al navegador. Esto se consigue utilizando el componente WebView, el cual me permite incorporar un módulo de página web a nuestra aplicación de escritorio.
4. Compatibilidad con JAVA SWING: Esta es una característica de suma importancia de JavaFX, ya que me permite incorporar componentes de Java SWING a través de un componente llamado SwingNode.

5. Controles de UI incorporados y CSS: Otra características de suma importancia es la posibilidad de poder cambiar la apariencia de los componentes con los que trabajamos a través de hojas de estilo o código CSS, lo cual nos brindará una potencia enorme a la hora de crear interfaces de Usuario(UI).
6. Aplicación MVC: Las aplicaciones desarrolladas con JavaFX, se desarrollan bajo el **patrón de diseño modelo vista controlador**. Con este modelo dividimos el código de la interfaz gráfica de la lógica de nuestra aplicación. De ahí la importancia que tienen els archivos FXML. La extensión FXML nos indicará una interfaz gráfica.

## 2.- ENTORNO DE TRABAJO

Una vez instalado i configurado Java e IntelliJ IDEA pasaremos a instalar JavaFX. Con la llegada de Java 11 el 2018, se eliminó JavaFx del JDK para que pudiera evolucionar como proyecto independiente de código abierto. Guiado por Oracle i la comunidad OpenJFX.

La versión OpenJFX de JavaFX que vamos a utilizar es una librería que tendremos que descargar i enlazar en nuestro IDE IntelliJ IDEA para poder desarrollar aplicaciones Java. La descarga la haremos desde la página oficial del OpenJFX:



Reference Documentation

<https://openjfx.io/>

• [Getting Started with JavaFX 11+](#)

En esa página iremos a **Getting Started with JavaFX 11+**. Al apartado **JavaFX i IntelliJ** vamos a elegir **Non-modular projects**. En nuestro caso vamos a generar un proyecto Java FX no modular y utilizaremos las herramientas del IDE para crearlo y ejecutarlo. La descarga la haremos haciendo un clic en el enlace **JavaFX SDK**.

Introduction  
Install Java  
Run HelloWorld using JavaFX  
Run HelloWorld via Maven  
Run HelloWorld via Gradle  
Runtime images  
JavaFX and IntelliJ  
Non-modular from IDE

### IDE

Follow these steps to create a JavaFX non-modular project and use the IDE tools to build it and run it. Alternatively, you can download a similar project from [here](#).

Download the appropriate **JavaFX SDK** for your operating system and unzip it to a desired location, for instance `Users/your-user/Downloads/javafx-sdk-24`.

1. Create a JavaFX project

A continuación, elegimos la versión, sistema operativo arquitectura y tipo adecuado:

### Downloads

JavaFX version	Operating System	Architecture	Type
<input type="text" value="24"/>	<input type="text" value="Linux"/>	<input type="text" value="x64"/>	<input type="text" value="SDK"/>

☐ Include archived versions

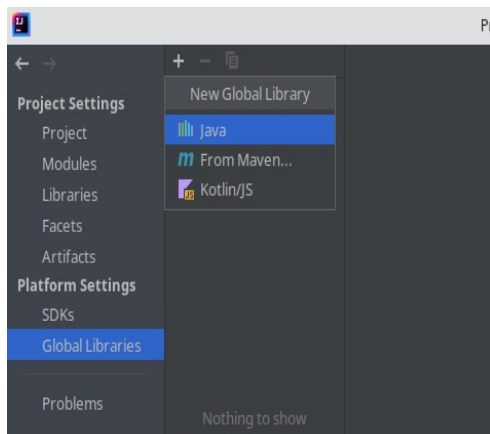
### Supported Platforms

OS	Version	Architecture	Type	Download
Linux	24	x64	SDK	<a href="#">Download (SHA256)</a>

Obtendremos una carpeta comprimida (openjfx-24\_linux-x64\_bin-sdk.zip, por ejemplo) que deberemos descomprimirla y ubicarla en un lugar de nuestro disco duro que sea permanente (puede ser en cualquier lugar). Como sugerencia se puede mover a la carpeta que contiene el JDK(en Linux suele ser /usr/lib/jvm y en Windows: C:/Archivos de Programa/Java, aunque se podría haber instalado el JDK en cualquier otro lugar).

A continuación, configuraremos el JavaFX SDK a las bibliotecas globales para mayor comodidad si estamos creando un proyecto nuevo desde IntelliJ IDEA.

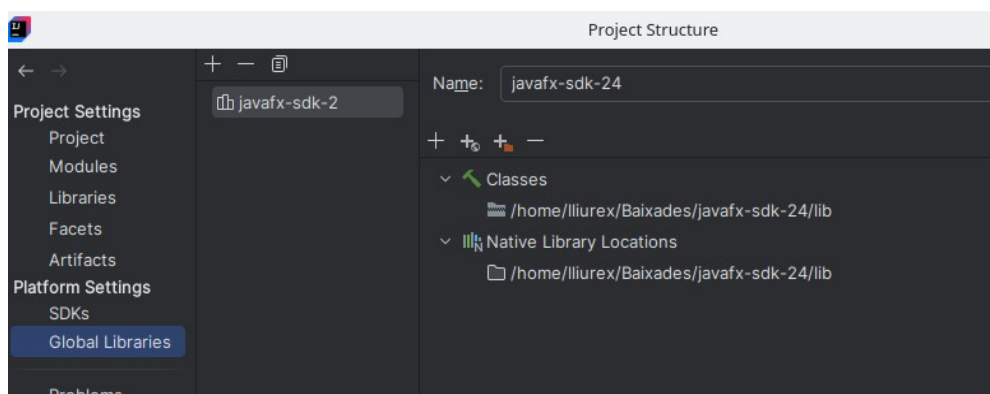
1. Creamos un proyecto nuevo en **New Project** como lenguaje Java y como sistema de construcción (compilación) el IDE IntelliJ, así como el JDK que vamos a utilizar.
2. Configuración de la bibliotecas generales. Añadimos la biblioteca JavaFX para crear un proyecto JavaFX. **File** → **Project structure** y seleccionamos **Globals Libraries**. Hacemos un clic en el **+** para incluir la librería.



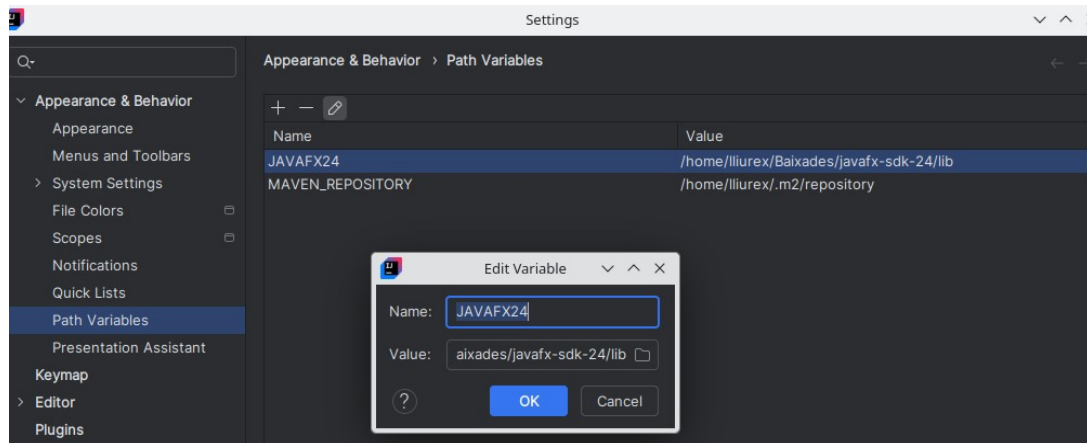
3. Indicamos el lugar donde se encuentra la librería JavaFX. En Linux será:

***/usr/lib/jvm/javafx-sdk-19.0.2.1/lib***

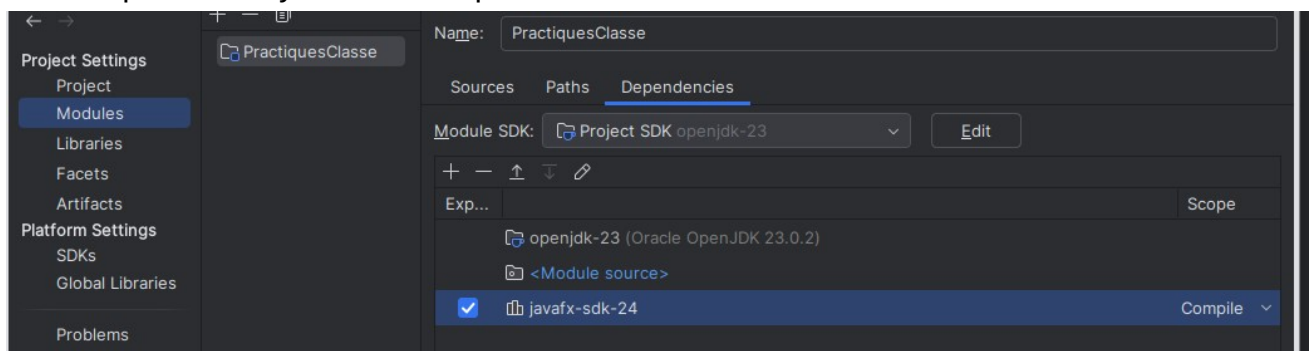
4. Una vez añadida la podremos ver en la siguiente figura:



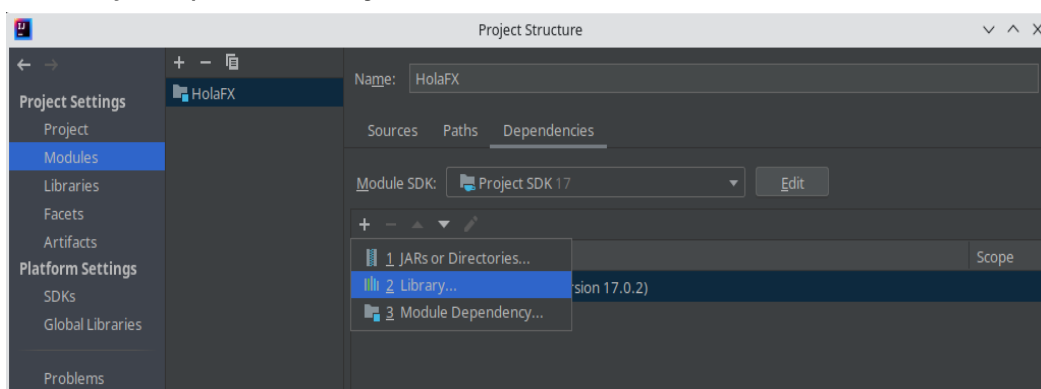
5. A continuación, configuramos la ruta de la variable (útil para configurar la VM). **File** → **Setting**. En el apartado **Appearance & Behavior** vamos a **Path Variables**. Añadimos la variable **JavaFX19** y seleccionamos la ruta donde se encuentra la librería del JavaFX ( **/usr/lib/jvm/javafx-sdk-19.0.2.1/lib**).



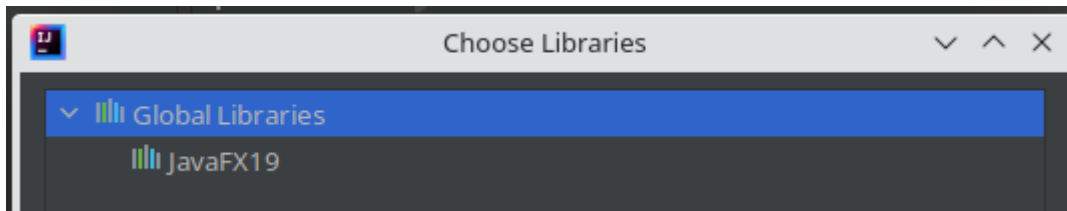
6. Ahora tendremos que indicar que una de sus dependencias sea JavaFX para que tenga efecto sobre el proyecto. Vamos de nuevo a **File** → **Project Structure**. Veremos la configuración del proyecto. Si vamos a **Libraries** no veremos la biblioteca del JavaFX. Pero si vamos a **Modules** podemos comprobar que en la pestaña **Dependencies** tenemos la librería como dependencia y tendremos que seleccionarla



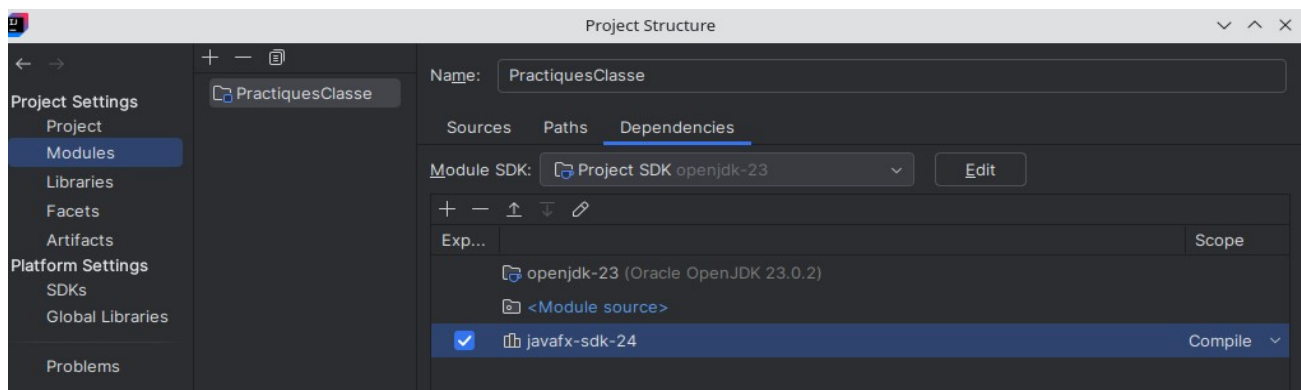
Cuando creamos un nuevo proyecto deberemos de activar esta dependencia haciendo un clic en el + y la opción **Library**.



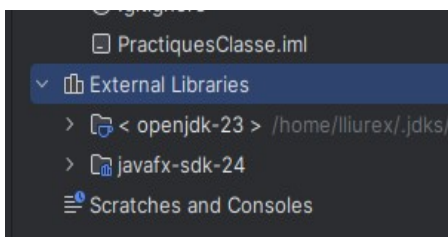
Nos saldrá la librería que tendremos que añadir para trabajar con JavaFX:



La seleccionamos (**JavaFX19**) y hacemos un clic en **Add Selected**. A continuación la marcamos con un tic.



Podemos comprobar como la librería JavaFx19 forma parte de nuestro proyecto.



7. Ahora crearemos una aplicación JavaFX (HolaFX) a partir del ejemplo que podemos encontrar en el enlace de la documentación de JavaFX (<https://openjfx.io/openjfx-docs/>):

JavaFX and IntelliJ

Non-modular from IDE

Non-modular with Maven

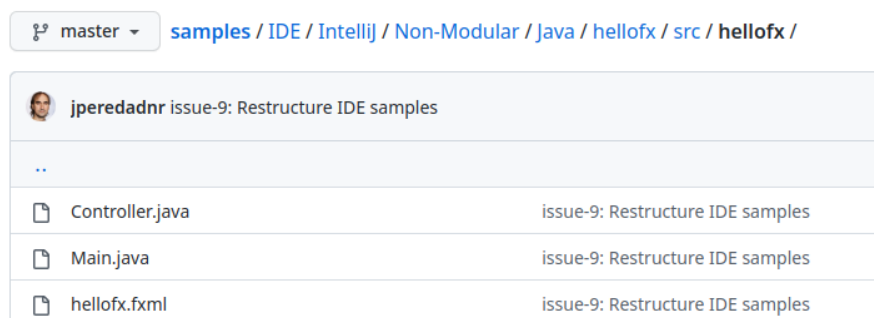
Non-modular with

### Non-modular projects

#### IDE

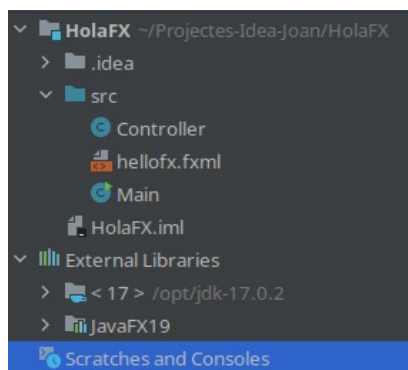
Follow these steps to create a JavaFX non-modular project and use the IDE tools to build it and run it. Alternatively, you can download a similar project from [here](#).

Al hacer un clic en **here** iremos a la página donde probaremos los archivos *Main.java*, *Controller.java* y *hellofx.fxml*.





Crearemos esos archivos con el mismo contenido en nuestro proyecto.

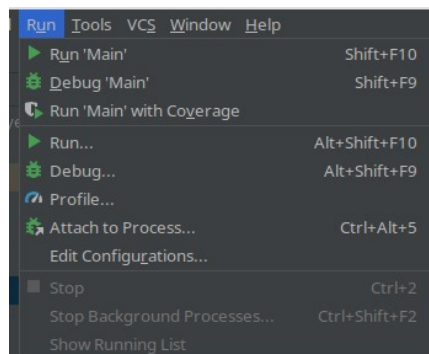
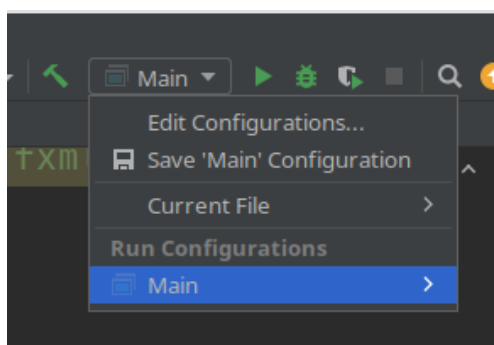


En el archivo `hellofx.fxml` deberemos modificar `fx:controller="hellofx.Controller"` por `fx:controller="Controller"` dentro de la etiqueta `<StackPane`. En nuestro caso, `Controller` no depende de ningún paquete. (TENDEREMOS QUE INDICAR LA RUTA DONDE ESTÁ `Controller`)

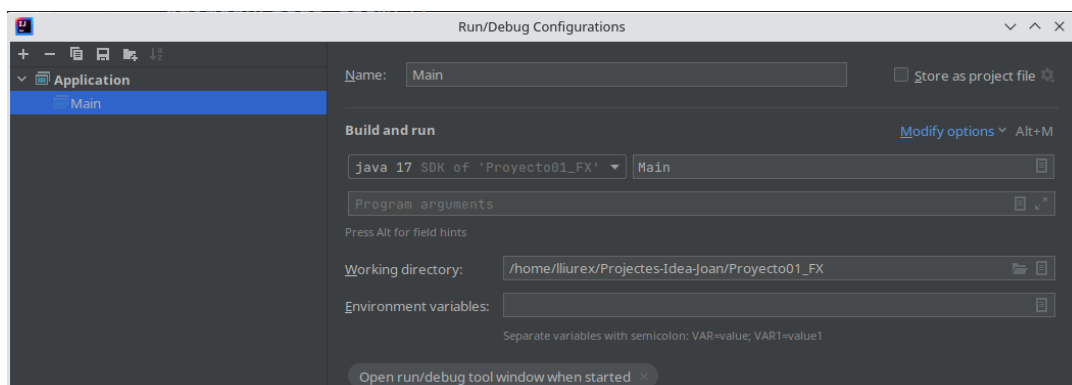
- Si ejecutamos el `Main` veremos el siguiente error, el cual indica que faltan componentes de tiempo de ejecución de JavaFX y que éstos son necesarios para ejecutar esta aplicación.

Error: JavaFX runtime components are missing, and are required to run this application

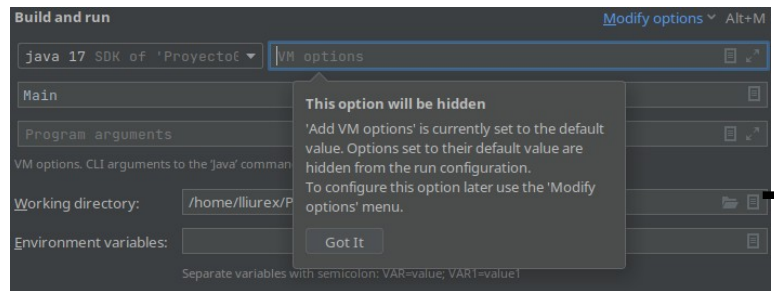
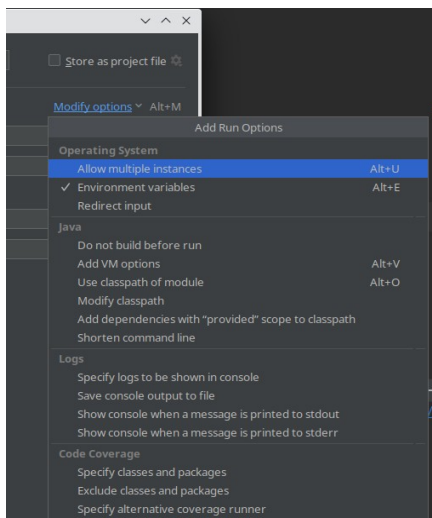
- El error lo solucionaremos editando la configuración de ejecución del proyecto.



Vamos a **Edit Configurations ...**



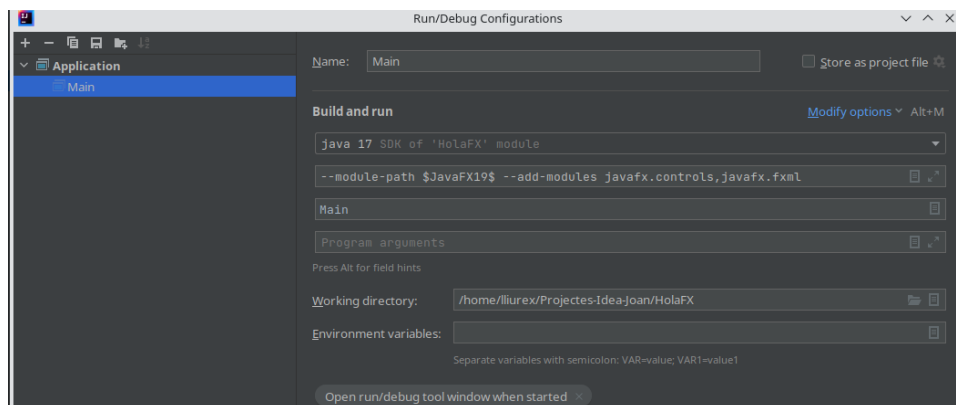
Agregaremos las opciones de la VM en el apartado **Modify options** → **Add VM options** y obtendremos la siguiente pantalla:



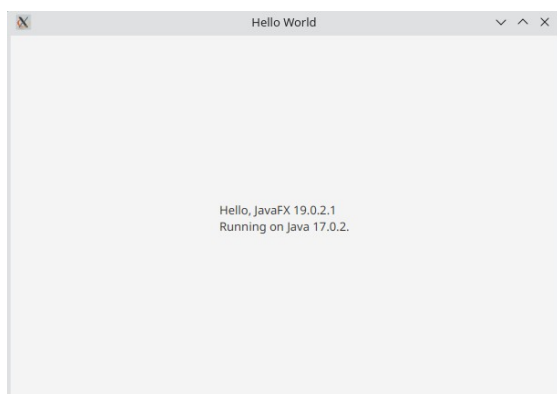
Ahora agregaremos al ruta del módulo (`--module-path`). A continuación la ruta donde se encuentra la librería JavaFx que definimos en la variable `JavaFX19` (a la derecha de donde estamos escribiendo podéis hacer un clic en Insertar Macros para visualizar esta variable) “`$JavaFX19`”. Después agregamos otros módulos (`--add-modules`) como los controladores de JavaFX (`javafx.controls`) y para crear archivos fxml usando el generador de escenas o codificarlo manualmente (`javafx.fxml`)

**`--module-path "$JavaFX19" --add-modules javafx.controls,javafx.fxml`**

**`--module-path /path/to/javafx-sdk-19/lib --add-modules javafx.controls,javafx.fxml`**  
10.



11. Volvemos a ejecutar el Main para ver si ya se ejecuta correctamente la aplicación JavaFX. El resultado debe ser el siguiente:



### 3.- ESTRUCTURA DE UNA APLICACIÓN DESARROLLADA CON JAVA FX.

#### 3.1.-CLASE APPLICATION

La clase **Application** perteneciente al paquete *javafx.application* es el punto de partida de cualquier aplicación desarrollada en JavaFX. Para crear una aplicación con esta tecnología la clase principal debe heredar de esta clase e implementar su método abstracto **start ()**, el cual permitirá inicializar la interfaz gráfica.

Esta clase posee tres métodos: **init()**, **start()** y **stop()** ,los cuales son los referentes al ciclo de vida de una aplicación JavaFX. A continuación ilustramos la estructura inicial de una aplicación desarrollada con JavaFX.

```
import javafx.application.Application;
public class EstructuraJavaFX extends Application {
}
```

En este código hemos creado la clase **EstructuraJavaFX** que hereda de la clase **Application**.

El método **start()** nos va a permitir inicializar la interfaz gráfica.

#### 3.2.-ESTRUCTURA DE UNA APLICACIÓN JAVA FX

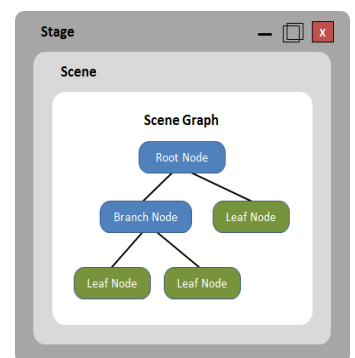
Una aplicación desarrollada con JavaFX, es decir, una clase **Application** está compuesta por tres componentes esenciales: **Stage** (escenario), **Scene** (escena) y **Nodes** (nodos).

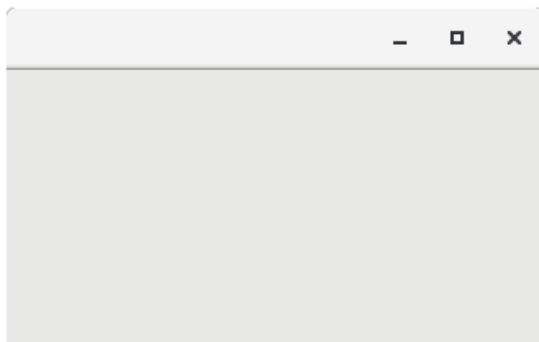
##### 3.2.1.- Clase Stage

La clase **Stage** o Escenario es una ventana en la cual agregaremos todos los objetos de nuestra interfaz gráfica. Por lo tanto, actúa como contenedor principal. Es el marco principal.

El **Stage** o escenario principal lo crea la plataforma JavaFX. El objeto tipo **Stage** es creado y se pasa como argumento al método **start ()** de la clase **Application**. Para poder visualizar dicha ventana debemos hacer uso del método **show()**. Una aplicación JavaFX puede tener múltiples objetos Stage.

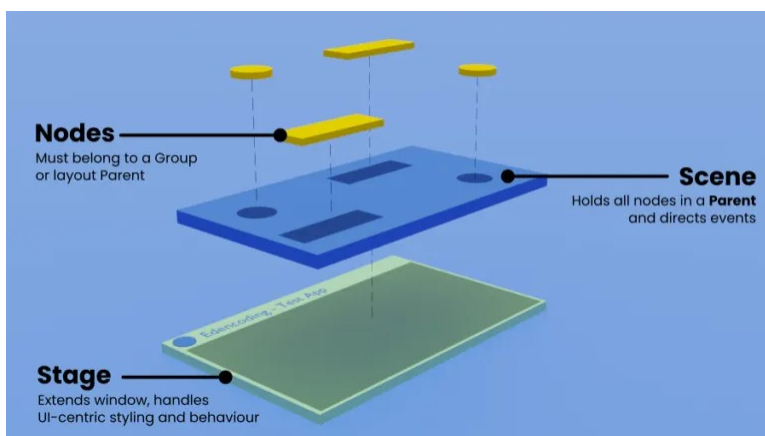
Para poder agregar objetos a este contenedor se necesitarán otro tipo de componentes como los **Scene** o los **Scene Graph**. Un ejemplo de un escenario sería el siguiente:





### 3.2.2.- Clase Scene

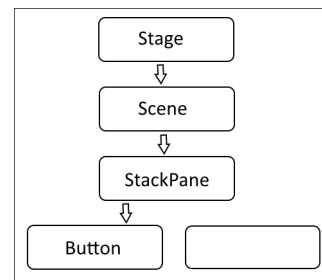
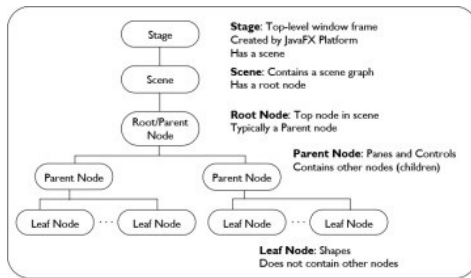
Cuando nos referimos a una escena o **Scene**, estamos haciendo referencia a los contenidos físicos (nodos) de una aplicación JavaFX. La clase **Scene** pertenece al paquete **javafx.scene** y nos proporciona todos los métodos para manejar un objeto de escena. Es necesario crear una escena para visualizar los contenidos en el escenario (**Stage**). Cuando se crea un objeto de tipo **Scene**, debemos pasar como argumento el nodo raíz el cual contendrá todos los nodos que se visualizarán en dicha interfaz e igualmente tenemos la posibilidad de asignar unas dimensiones (ancho y alto) a la escena.



### 3.2.3.- Nodos - Gráficos de Escena.

El grupo de gráficos de escena y nodos sería el nivel más bajo del diagrama que hemos ilustrado anteriormente. Un gráfico de escena es una estructura de datos la cual actúa en forma de árbol jerárquico y estos representan el contenido de una escena. En cambio, un nodo es un objeto visual de un gráfico de escena.

Por lo tanto, un nodo hace referencia a los componentes que ya conocemos los cuales pueden ser: etiquetas de texto (*Labels*), Botones (*Buttons*), Tablas (*Tables*), Contenedores (*Containers*). Un gráfico de Escena no es más que un conjunto de Nodos divididos en orden jerárquico.



### 3.3.-EJEMPLO PRÁCTICO

A continuación trabajaremos un ejemplo elaborado paso a paso donde aplicaremos los conceptos vistos.

1. Creamos un nuevo proyecto (EjemploPráctico) y le agregamos la librería JavaFX.
2. Creamos la clase Main que ha de heredar de la clase Application.
3. Generamos el método abstracto start() el cual carga todos los componentes u objetos de la interfaz gráfica. Éste recibe un argumento el cual será un objeto de tipo Stage.
4. Creamos el método main. Dentro de él invocamos al método launch(args), el cual se encarga de ejecutar todo lo que tenemos en el método **start()**.
5. Implementamos el método start().

## 4.- CICLO DE VIDA CLASE APPLICATION - LIFE CYCLE JAVAFX.

### 4.1.-CICLO DE VIDA DE CLASE APPLICATION.

La clase principal de nuestro proyecto siempre debe heredar de la clase **Application**. Esta clase posee los métodos del ciclo de vida que serán detallados en el siguiente punto.

Entendemos por ciclo de vida a las fases o etapas que tomará nuestra aplicación al momento de ser ejecutada. Todo software que nosotros desarrollemos independientemente de la tecnología siempre tendrá un orden cronológico el cual siempre tendrá un punto inicial y un punto final.

### 4.2.-CICLO DE VIDA CLASE APPLICATION - MÉTODOS.

Los métodos que controlan el ciclo de vida de la clase **Application** son:

1. **Método init():** Es invocado justo después de crear una instancia de la clase **Application**, es decir, el método **init** será lanzado justo después de ser ejecutado el método constructor y puede usarse para inicializaciones específicas de la aplicación. Algunos ejemplos de usos del

método **init** puede ser para realizar validaciones con la base de datos o cargar una configuración inicial de nuestro programa.

Aunque el método **init** no lo creemos en nuestra clase, al momento de la ejecución de nuestro programa se creará una instancia vacía de este método. Todo este proceso se hace de manera interna. Por lo tanto no es obligatoria la creación de este método, solo será necesario si requerimos la utilización de éste.

2. **Método start()**: Llamado después de ser creado el proceso que gestiona la interfaz gráfica de usuario de la aplicación. Este método nos proporciona un primer escenario (**Stage**) el cual hace referencia a la ventana principal. En este método se creará todo lo referente a la interfaz gráfica tales como: escenarios (**Stage**), escenas (**Scene**) y los nodos (**Nodes**). Es importante resaltar que en este método se cargaran todos los componentes de nuestra aplicación como por ejemplo: Controladores, Lógica de nuestra aplicación.
3. **Método stop()**: Cuando la aplicación está a punto de finalizar (por ejemplo, cuando el usuario ha pulsado el botón cerrar de la aplicación), JavaFX llama al método **stop** de la clase **Application**. En este, se pueden realizar las tareas de limpieza apropiadas para la aplicación. Por ejemplo, destruir o cerrar todas las conexiones con la base de datos.

A parte de los anteriores, existe el método estático **launch()** al cual le pasamos como parámetro el *array* de los argumentos del método **main()**. Este método nos permite lanzar las aplicaciones JavaFX. Internamente, el método **launch()** determina la clase a lanzar, que será la clase principal que hereda de la clase **Application**, donde se encuentra el método **main()**. Esa clase se encarga de inicializar el ciclo de vida de nuestra aplicación JavaFX, creando la instancia de la clase, llamando al método constructor, método **init()**, método **start()**, ... Más información la podemos encontrar en la documentación de JavaFX de la clase **Application** (<https://openjfx.io/javafx/24/javafx.graphics/javafx/application/Application.html>).

### 4.3.-EJEMPLO PRÁCTICO

A Continuación desarrollaremos un ejercicio bastante sencillo para entender el funcionamiento interno del ciclo de vida de la clase **Application**.

1. Creamos un nuevo proyecto (CicloVida) y le agregamos la librería JavaFX.
2. Creamos la clase Main que ha de heredar de la clase Application.

3. Creamos el método main. Dentro de él invocamos al método launch(args), el cual se encarga de ejecutar todo lo que tenemos en el método **start()**.
4. Generamos el método abstracto **start()** el cual carga todos los componentes u objetos de la interfaz gráfica. Éste recibe un argumento el cual será un objeto de tipo **Stage**. En este caso crearemos un botón (Button). Cuando pulsemos el botón imprimirá en la consola “Hola Mundo”. Antes imprimimos por pantalla “Método start” para saber que se ha invocado al método.
5. Ejecutamos la aplicación y vemos su resultado.
6. Ahora vamos a crear el método init() y que imprima por consola “Método init”. A continuación creamos el constructor de la clase y que imprima “Constructor clase”. Y por último creamos el método stop() y imprimimos “Método stop.”
7. En el método main tenemos el método **launch** el cual nos permite lanzar nuestra aplicación JavaFX. Este método no debe llamarse más de una vez. De lo contrario generará una excepción. Este método de manera interna es el que se encarga de ejecutar y poner en marcha nuestras aplicaciones de JavaFX.
8. Comprobar que se invocan los métodos siguiendo el orden establecido por el ciclo de vida de una aplicación JavaFX.
9. Modificar el evento del botón para que una vez pulsado termine la aplicación. (EjemploPracticoSalir)

## 5.- MODELO – VISTA – CONTROLADOR

En los anteriores ejemplos, hemos implementado en una misma clase el código para visualizar la ventana con sus nodos (etiqueta o botón) y las acciones a realizar una vez se pulsa un botón. Además, estos ejemplos son tan simples que no hemos recurrido a ningún tipo de almacenaje en un dispositivo de almacenamiento ni de ningún modelo de clases. Continuar desarrollando la implementación de una aplicación en una misma clase no es una buena práctica. Para poder afrontar mejor una aplicación gráfica, es mejor tener separado el código atendiendo a su funcionalidad. De esa manera, es más fácil de programar y de mantener i/o actualizar nuestros programas.

### 5.1.-MVC

### 5.1.1.- Introducción

Cuando tenemos que elaborar una aplicación compleja como por ejemplo una aplicación cliente – servidor, es muy ventajoso utilizar el **MVC**. Este modelo surgió por la necesidad de simplificar tanto la elaboración de las aplicaciones como su futuro mantenimiento. Para ello, una aplicación se divide en módulos o partes. Con la modularización dividimos una aplicación en partes, conectamos esas partes entre si y hacemos que todas ellas funcionen como una unidad. Su mayor ventaja es que si falla algo lo vamos a tener más localizado que si tenemos la aplicación implementada en un sólo módulo. Además, el programa puede estar funcionando sin el módulo que ha presentado el error. El MVC se parece mucho a esa modularización.

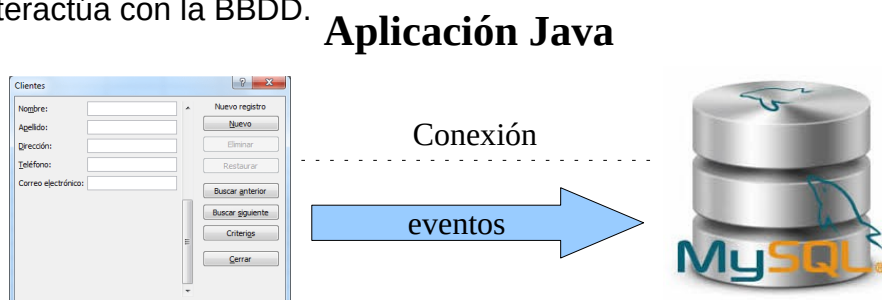
El MVC consiste en un patrón de arquitectura que separa la lógica del programa (datos), de la interfaz de usuario (ventanas) y las comunicaciones (eventos). Sus ventajas son:

- Modularización del programa.
- Reutilización del código.
- Mayor facilidad en el desarrollo. Es más fácil desarrollar una aplicación compleja.
- Mayor facilidad de mantenimiento. Tener el programa dividido en pequeños módulos es más fácil de mantener que si tenemos que modificar un módulo muy grande y complejo de miles de líneas de código.

El modelo MVC se utiliza en muchos entornos de trabajo con muchas tecnologías

### 5.1.2.- Estructura MVC

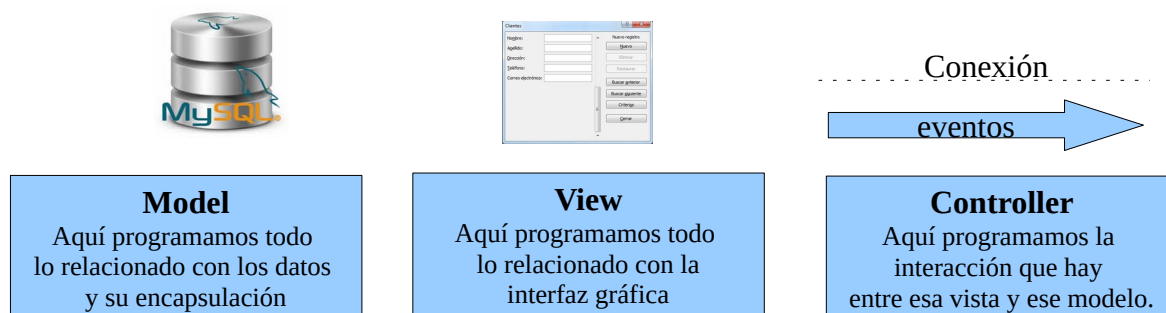
Supongamos una aplicación java en la cual tenemos una interfaz gráfica y una BBDD. Nuestra interfaz tiene que conectar con la BBDD y tiene que interactuar con ella a través de eventos. Por ejemplo, en el momento en que el usuario hace un clic en el botón de consulta o de enviar, es cuando se interactúa con la BBDD.



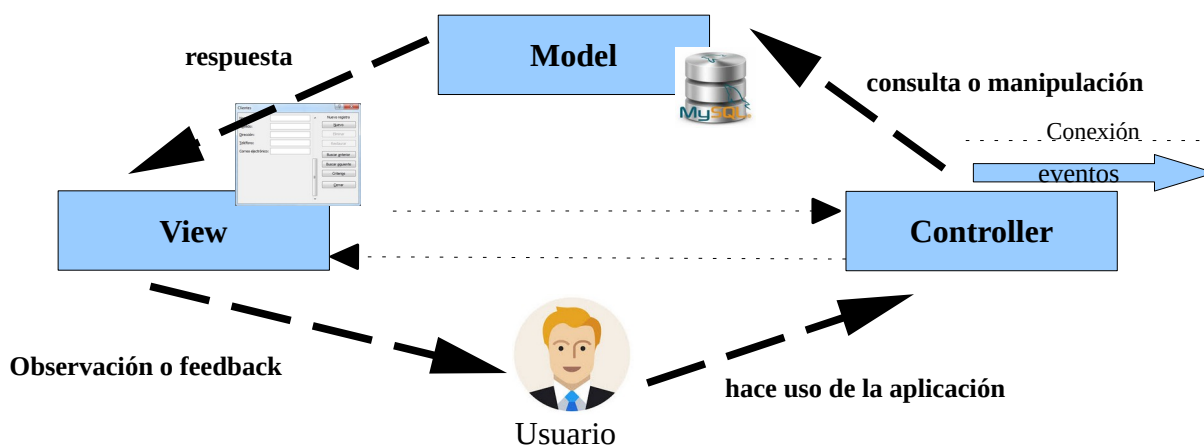
El MVC para una aplicación consiste en crear tres partes bien diferenciadas.



## Aplicación Java



### 5.1.3.- Funcionamiento y flujo del MVC



Vista y controlador están íntimamente relacionadas entre sí, y no tanto con el modelo. Parte gráfica y la conexión con los eventos están muy relacionados. El usuario usa la aplicación mediante eventos. Un usuario usa la aplicación cuando pincha en una casilla de texto (se produce un evento), pulsa un botón (se produce un evento), .. Entonces, el usuario hace uso de la aplicación a través de eventos que están dentro del módulo controlador. Una vez el usuario realiza el evento, ese controlador lo que hace es una consulta o manipulación en el modelo. Por ejemplo, cuando el usuario pulsa el botón enviar, se está utilizando el evento enviar el cual hace una petición de consulta o de manipulación de esa base de datos. Por lo tanto, algo ocurre en el modelo. El modelo recibe esa petición de consulta o manipulación, la procesa y da una respuesta a la vista para que ésta la pueda enseñar al usuario (observación/feedback). Ése sería el flujo que sigue un MVC.

### 5.1.4.- Ejemplo práctico

A partir del ejemplo HolaFX descargado en el apartado 2, adáptalo a un modelo vista controlador creando los paquetes correspondientes. Tendrás que adaptarlo a la nueva estructura de carpetas/paquetes (model, controller, view, application)

## 6.- SCENE BUILDER

Para mejorar la iteración con nuestros proyectos JavaFx vamos a instalar dentro nuestro IDE de un plugin y fuera de él de una herramienta que nos va a permitir manejar y interactuar mejor con los ficheros entre ellos el de fxml. La herramienta se llama **Scene Builder** la cual nos va a permitir trabajar y gestionar de forma visual con la vistas de nuestro proyecto JavaFX a través de ficheros fxml.

En primer lugar iremos a su página web para descarga su última versión:

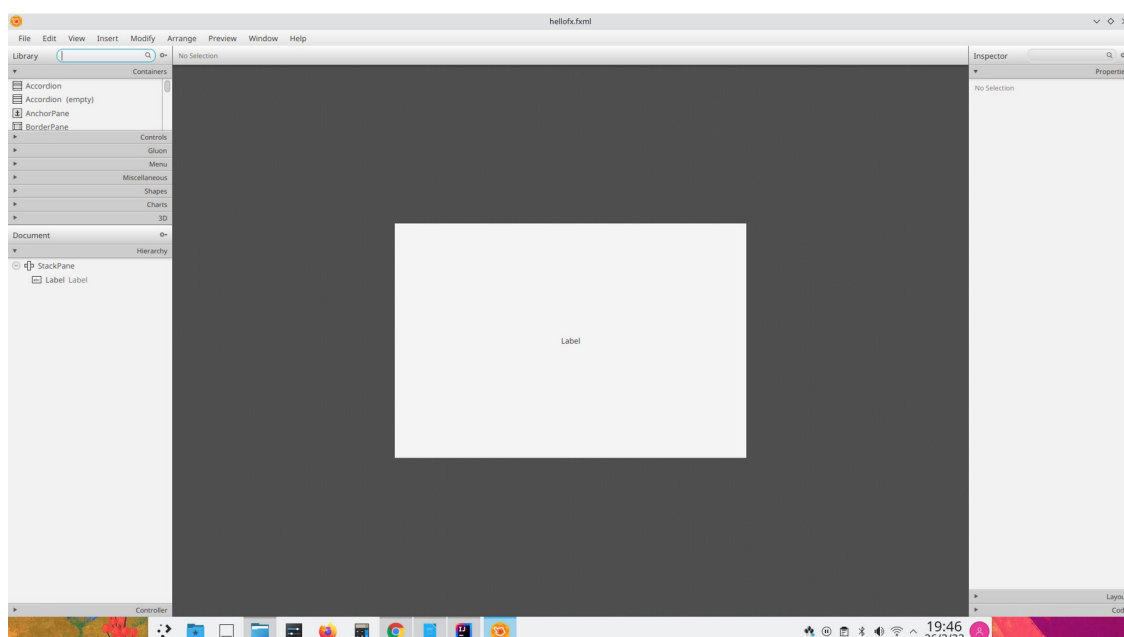
**<https://gluonhq.com/products/scene-builder/>**

Una vez elegida la plataforma en la que la queremos instalar (en Lliurex sería el paquete **deb**), lo descargamos y lo instalamos (doble clic en el **.deb**).

Una vez completada la instalación, pasaremos a configurar en nuestro IDE la asociación de los ficheros **fxml** con el editor que proporciona el creador de escenas **Scene Builder**. Los pasos a seguir son:

- **File** → **Setting** → **Languages & Frameworks** → **JavaFX**. En **Path to Scene Builder** elegimos el binario donde se ha instalado la aplicación Scene Builder (/opt/scenebuilder/bin/SceneBuilder).

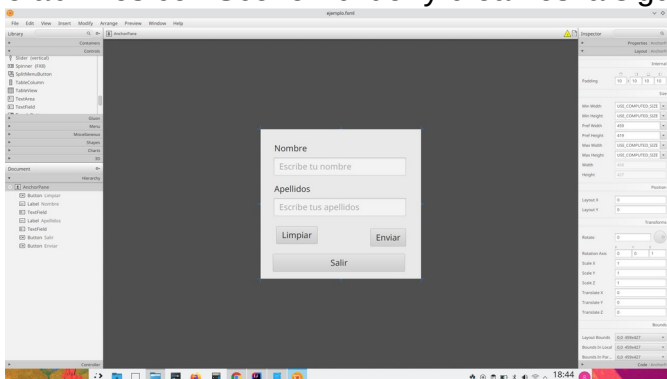
Una vez ligado la aplicación Scene Builder en el IDE, con el botón derecho encima del nombre de la vista le indicaremos que lo abra con el Scene Builder. Si lo hacemos con el archivo **hellofx.fxml** el resultado será:



## 7.- PRÁCTICA - 1

Vamos a realizar nuestra primera aplicación JavaFX con GUI entorno gráfico en Java con IntelliJ y SceneBuilder.

1. Creamos el proyecto de JavaFX llamado Practica1. (tenemos que añadir la librería)
2. Creamos la estructura MVC con el paquete application.
3. Creamos la clase principal Main en el paquete application. Ha de heredar de la clase **Application** (<https://openjfx.io/javadoc/19/javafx.graphics/javafx/application/Application.html>). En la documentación tienes toda la información de esta clase abstracta (también puedes ver el apartado 4).
4. Dentro de la clase principal deberemos de implementar el método **start** (bombilla roja) y el método **main**, el cual tendrá que llamar al **launch** que generará o instanciará esta aplicación (**Application**) y por lo tanto llamaría a **start** (ver apartado 4). En el método start imprimimos por pantalla “Metodo start”.
5. Ejecutamos la aplicación (recordar añadir lo componentes de JavaFX necesarios para ejecutar la aplicación en la configuración de ejecución del proyecto ). Vemos que ha sido lanzada ya que aparece en el terminal “Metodo start”. Finalizamos la aplicación con un clic en el botón rojo, ya que no tenemos implementado nada más en el método **start**.
6. Vamos a diseñar un formulario (ventana) de extensión **FXML** con IntelliJ y Scene Builder de nombre “**ejemplo.fxml**” dentro de la parte de las vistas (**view**). Fijaros que por defecto pone como controlador **fx:controller="view.Ejemplo"**. Cada vista tiene asociada un controlador. En la url [https://openjfx.io/javadoc/19/javafx.fxml/javafx/fxml/doc-files/introduction\\_to\\_fxml.html](https://openjfx.io/javadoc/19/javafx.fxml/javafx/fxml/doc-files/introduction_to_fxml.html) tenemos una introducción a fxml donde podemos ver algunas de las etiquetas y atributos más utilizados
7. Lo abrimos con Scene Builder y creamos la siguiente interfaz:



8. Existen numerosas propiedades para cada control (componente) que podemos establecer tanto en el modo gráfico de Scene Builder como editando directamente el fichero **.fxml** que se genera. Cada componente tendrá sus propiedades
9. Nombraremos los **TextField** y los **Button**, para luego poder usarlos y hacer referencia a ellos en el controlador de la vista generada. Nos iremos a la pestaña **Code** y en **fx:id** introduciremos el nombre para cada componente. En **View** → **Show Sample Controller Skeleton** podemos ver un ejemplo de como ha de ser el controlador de la vista generada.
10. Una vez añadidos los componentes visuales y sus nombres identificativos podremos establecer los eventos de cada uno que queramos programar. En nuestro caso vamos a programar los tres botones. Lo vamos a hacer desde el Scene Builder. El siguiente ejemplo es para el botón limpiar. Haremos lo mismo con los otros dos botones. Al igual que hemos comentado que existen multitud de propiedades para un componente (control), también existen multitud de eventos. Cada componente tendrá los suyos.

```
package view;

import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;

public class Ejemplo {

    @FXML
    private Button btnEnviar;

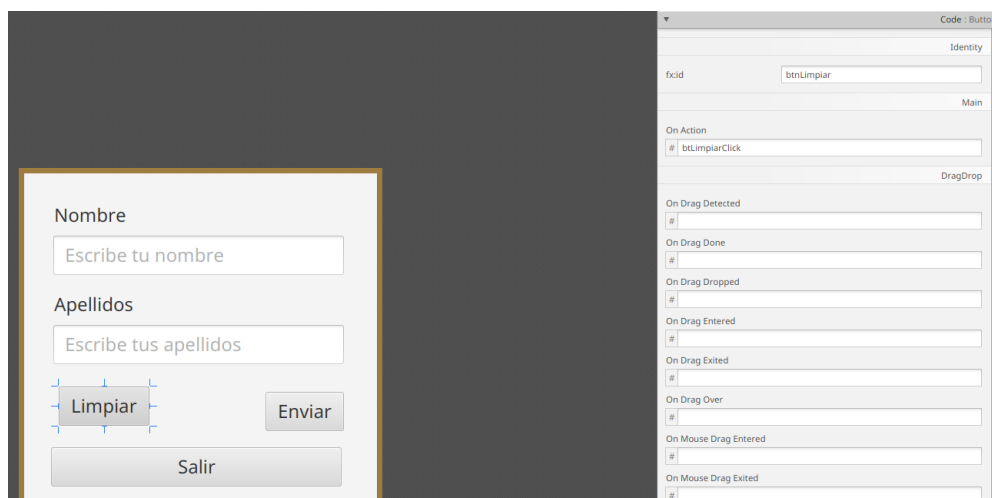
    @FXML
    private Button btnLimpiar;

    @FXML
    private Button btnSalir;

    @FXML
    private TextField txtApellidos;

    @FXML
    private TextField txtNombre;

}
```



11. En **View** → **Show Sample Controller Skeleton** podemos ver un ejemplo de como ha de ser el controlador de la vista generada una vez establecido los eventos para cada botón.

```
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;

public class Ejemplo {

    @FXML
    private Button btnEnviar;

    @FXML
    private Button btnLimpiar;

    @FXML
    private Button btnSalir;

    @FXML
    private TextField txtApellidos;

    @FXML
    private TextField txtNombre;

    @FXML
    void btnLimpiarClick(ActionEvent event) {

    }

    @FXML
    void btnEnviarClick(ActionEvent event) {

    }

    @FXML
    void btnSalirClick(ActionEvent event) {

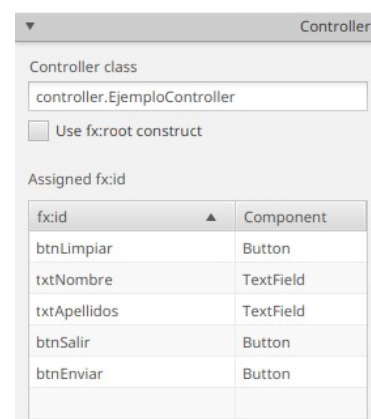
    }

}
```

12. Además, en el archivo **ejemplo.fxml** podemos ver el código **xml** introducido (tenemos que grabar desde Scene Builder para ver reflejado las modificaciones). En negrita vemos el nombre del controlador asociado a la vista, la identificación y el evento asociado a un control del tipo **Button**. Desde este archivo podemos modificar el formulario (ventana), aunque resultará mucho más complicado y hemos de tener conocimientos de XML y de composición de escenas para JavaFX. Mucho mejor usar Scene Builder.

```
<AnchorPane          prefHeight="419.0"          prefWidth="459.0"          xmlns="http://javafx.com/javafx/19"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="view.Ejemplo">
  <children>
    <Button fx:id="btnLimpiar" layoutX="45.0" layoutY="277.0" mnemonicParsing="false" onAction="#btLimpiarClick"
text="Limpiar">
      <font>
        <Font size="24.0" />
      </font>
    </Button>
```

13. Para cambiar el nombre del controlador, vamos a la pestaña **Controller** de la derecha y indicamos la ruta y el nombre correcto. Por ejemplo **controller.Ejemplo**. El cambio se verá reflejado en la vista **fxml**.



14. Desde el archivo **ejemplo.fxml** podemos decirle que nos genere la clase del controlador de la vista si nos situamos encima de `fx:controller="controller.Ejemplo"`, pero nos la crearía vacía. Mejor ir de nuevo a **View** → **Show Sample Controller Skeleton** en Scene Builder y grabar el archivo en el paquete **controller**.

15. Una vez que hemos añadido los componentes visuales, sus ID y sus eventos, es hora de programar lo que queramos que haga cada uno de ellos. En nuestro ejemplo, mostraremos una ventana para que el usuario introduzca el nombre y los apellidos. Una vez introducidos, cuando pulse en el botón «Enviar», mostrará los datos en un mensaje. Si pulsa en «Limpiar» borrará los datos introducidos y si pulsa el botón «Salir» cerrará la aplicación. Todo lo haremos en el controlador de la vista, es decir, en el archivo **EjemploController.java**.

16. Ahora nos queda implementar la clase principal para iniciar la escena JavaFX al arrancar la aplicación. Si nos fijamos en el código del ejemplo “HolaFX” podemos ver lo siguiente:

```
Parent root = FXMLLoader.load(getClass().getResource("../view/hellofx.fxml"));
primaryStage.setTitle("Hello World");
primaryStage.setScene(new Scene(root));
primaryStage.show();
```

- La primera línea corresponde a la carga del fichero **fxml** : ***FXMLLoader.load***. Se utiliza la clase ***FXMLLoader*** con el método ***load()***.
- Para indicar dónde está nuestro fichero utilizaremos ***getClass().getResource("hellofx.fxml")*** que nos va a proporcionar esa *url* directa a partir del fichero.
- En nuestro caso tendremos que indicar la url del archivo ***ejemplo.fxml*** que se encuentra en el paquete ***view***. También cambiaremos el título del escenario

```
@Override
public void start(Stage primaryStage) throws Exception {
    System.out.println("Método start.");
    Parent root = FXMLLoader.load(getClass().getResource("../view/ejemplo.fxml"));
    primaryStage.setTitle("Práctica 1");
    primaryStage.setScene(new Scene(root));
    primaryStage.show();
}
```

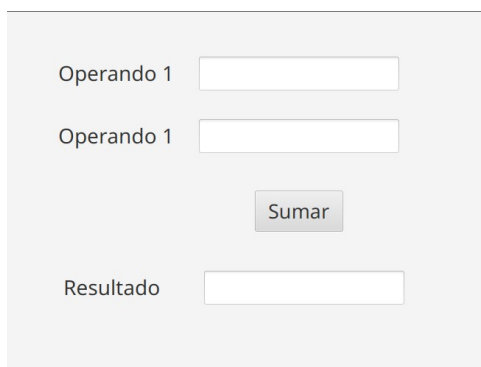
17. Comprobamos si la ejecución es correcta.

## 8.- CONTROLLER INITIALIZABLE (SUMA DE DOS NÚMEROS)

En la práctica anterior hemos visto como utilizar SceneBuilder para crear nuestras vistas, pero también para crear nuestro controlador. A parte de realizar el diseño de la ventana indicando los nodos que va a contener nuestra escena, también hemos asignado un identificador (fx:id) a los nodos para poderlos referenciar en el código a implementar en el controlador relacionado con la vista. De igual forma, hemos identificado los eventos que van a tener los nodos, así como la ubicación y nombre del controlador asociado.

Con esta práctica vamos a aprender una nueva forma de relacionar o controlar los eventos, que nos dará mucho más control en la inicialización i relación con el controlador .

1. Realiza el proyecto llamado SumaDeDosNumeros con JavaFX.
2. Crea los paquetes necesarios para el modelo MVC.
3. Crea la siguiente ventana o vista principal en el paquete adecuado (SumaVista.fxml)

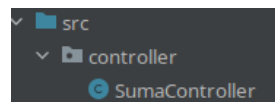
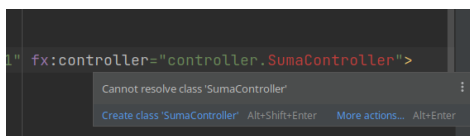


The image shows a JavaFX window with a light gray background. It contains two input fields, each preceded by the text 'Operando 1'. Below these is a button labeled 'Sumar'. At the bottom, there is an output field preceded by the text 'Resultado'.

4. Crea los identificadores para la lectura de los operandos (txtOp1, txtOp2), el botón (btnSumar) y visualizar el resultado (txtResultado). Grava la vista.
5. En SceneBuilder podemos ver que como nombre de controlador aparece view.SumaVista. Cámbialo por controller.SumaController. Grava la vista i visualízala en el IDE. Veremos la siguiente línea.

```
fx:controller="controller.SumaController">
```

6. Si colocamos el cursor encima de controller.SumaController veremos que nos permite crear la clase controlador en el paquete controller. Es otra forma de crear el controlador. Lo crea sin ningún atributo (identificadores de los nodos de la escena).



Recordar que cada archivo FXML debe tener asociado una clase Java a la que se suele denominar "controladora". En esta clase se deben declarar las instrucciones Java que se desean ejecutar cuando se inicialice el interfaz gráfico contenido en el archivo FXML, o también, por ejemplo, declarar el método que debe ejecutarse cuando el usuario haga clic en un botón, o el código necesario para cambiar un campo de texto, rellenar una lista de datos, etc

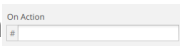
7. Si queremos que aparezcan los atributos FXML deberemos visualizar el esqueleto de la vista que hemos creado en SceneBuilder, seleccionarlos y copiarlos en nuestro controlador. Es otra manera de relacionar el controlador SumaController.java con la vista SumaVista.fxml.
8. Ahora tendremos que implementar los mecanismos para poder lanzar un evento (en la práctica anterior lo hacíamos en el mismo SceneBuilder). En nuestro caso, tenemos que indicar en el controlador SumaController la relación del botón Sumar con el manejador de eventos (*handleButtonAction*). En definitiva, en el momento se ejecute el controlador tendremos que indicar que ese botón identificado con su fx:id tendrá asociado un evento (aquello que anteriormente hemos hecho en SceneBuilder). Para ello vamos a implementar la interfaz **Initializable** cuya documentación se encuentra en la página:

<https://docs.oracle.com/javase/8/javafx/api/javafx/fxml/Initializable.html#skip.navbar.top>

Cuando una clase implementa esta interfaz, estamos indicando que queremos añadir una inicialización a nuestro controlador una vez se carga la vista fxml asociado a él.

Como interfaz, tendremos que implementar los métodos asociados a ella:

```
@Override
public void initialize(URL url, ResourceBundle resourceBundle) {
}
```

Las instrucciones Java que indiquemos dentro de este método se ejecutarán cada vez que se cargue el interfaz gráfico durante la ejecución de la aplicación. Por lo tanto, aquí tendremos que incorporar la acción a realizar cuando hagamos un clic en el botón de identificador btnSumar (en SceneBuilder lo hicimos en ). Para asociar un evento al botón lo indicaremos con this:: (la palabra reservada this hace referencia al objeto y :: se llama operador de referencia de método y sirve para hacer referencia a un método)

En esta inicialización no sólo pondremos aquello que tiene una relación directa con



ScenBuilder, sino que también podemos realizar otras acciones como por ejemplo temas de datos, de visualizaciones que sean dependientes para esa inicialización correcta de nuestro modelo o de nuestros eventos. Para comprobar que una vez cargada la vista se ejecuta su controlador asociado, vamos a imprimir por consola el nombre del controlador que ha cargado la vista. A continuación, asociamos un evento (será un método) al botón.

9. La implementación a realizar será la siguiente:

```
public class SumaController implements Initializable {  
    //atributos @FXML  
    @Override  
    public void initialize(URL url, ResourceBundle resourceBundle) {  
        System.out.println("Inicualizo SumaController");  
        btnSumar.setOnAction(this::sumar);  
    }  
}
```

10. Con esa implementación estamos indicando que cada vez que se pulse el botón con fx:id btnSumar, se tiene que lanzar el evento sumar tantas veces como vceces pulsemos el botón Sumar.

11. Por último implementamos el método sumar (ponemos el cursor encima del método y nos lo creará) que recibirá el evento producido al pulsar el botón sumar. La estructura será la siguiente:

```
private void sumar(ActionEvent event) {  
}
```

Antes de implementar el método sumar, vamos a crearnos un modelo de ejemplo que utilice una clase llamada Suma que nos sirva para poder realizar la suma pero utilizando el modelo creado. De esa manera entenderéis mejor el modelo MVC.

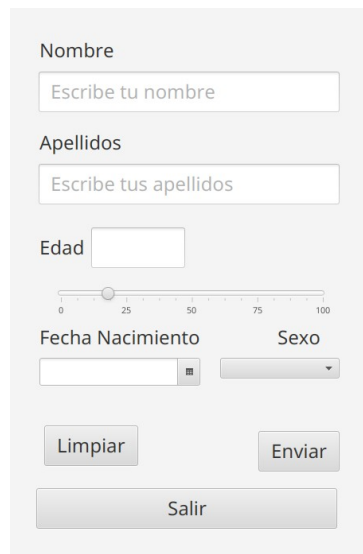
## 8.1.-PRÁCTICA – 1 MODIFICADA

Modifica la práctica 1 hecha en el apartado 7 para que la ventana sea la siguiente. En ella hemos añadido el componente ComboBox, el cual es uno de los controles más comúnmente usados en la creación de interfaces gráficas. Podemos ver que se ha añadido como `ComboBox<?> cboxSexo;` donde ? Indica el tipo de datos a mostrar.

Su forma más simple es utilizada para mostrar una lista desplegable de cadenas de texto. En nuestro caso vamos a indicar que es un enumerado.

La clase Slider representa un componente de control deslizante. El método `valueProperty()` devuelve un objeto de propiedad (property object) que representa el valor actual del slider. Podemos

indicar que se produzca una acción cuando se cambie el valor del control deslizante. Esto se hace agregando un oyente a esa propiedad, utilizando el método `addListener()`.



The image shows a JavaFX form with the following elements:

- Nombre**: A text input field with the placeholder text "Escribe tu nombre".
- Apellidos**: A text input field with the placeholder text "Escribe tus apellidos".
- Edad**: A numeric input field with a range slider below it. The slider has markers at 0, 25, 50, 75, and 100.
- Fecha Nacimiento**: A date picker control.
- Sexo**: A dropdown menu.
- Limpiar**: A button to clear the form.
- Enviar**: A button to submit the form.
- Salir**: A button to exit the application.

Modifica el ejercicio para que el usuario introduzca valores correctos (expresiones regulares).  
Amplia la información con el correo electrónico

## 9.- SUMA, RESTA, MULTIPLICACIÓN Y DIVISIÓN DE DOS NÚMEROS

Realiza el siguiente proyecto donde creamos un ToggleGroup y le indicamos a cada radio button que es de ese grupo. Si todos son del mismo grupo, conseguimos que todos sean iguales y dejarán de ser independientes. Controla los datos introducidos con expresiones regulares.



## 10.- LISTA DE PERSONAS

En esta práctica vamos a utilizar el componente TableView que contiene columnas y el componente TableColumn para añadirlas al TableView. Tanto a la tabla como a cada columna les pondremos un nombre y un id. El concepto que vamos a utilizar es que cuando pulsemos el botón agregar persona, ésta deberá de aparecer en la tabla. Una vez añadido un TableView, veremos que aparece un ?, el cual indica un objeto genérico. En nuestro caso indicaremos que es el objeto Persona. En las TableColumn eliminamos <?,?> ya que al ser una columna de esa tabla ya cogería a Persona por defecto.

Recordar que para ver si dos Personas son iguales deberemos generar el hashCode() y el equals() (con el botón derecho los podemos generar automáticamente dentro de la clase Persona).

Para generar una lista de personas deberemos de utilizar una estructura donde almacenarlas. El lenguaje de programación Java cuenta con colecciones como: List, Set, Map. La API JavaFX extiende estas colecciones con las interfaces: ObservableList, ObservableSet, ObservableMap, respectivamente, con el objetivo de proporcionar a las colecciones el soporte para la notificación de cambios e invalidación de acuerdo a como se hace en JavaFX.

Estas colecciones se encuentran en el paquete javafx.collections y para crearlas debemos usar la clase FXCollections que nos provee de distintos métodos estáticos para crear la colección que deseemos.

En nuestro caso vamos a utilizar la interfaz ObservableList. JavaFX ObservableList es la

implementación especial de la interfaz List de JavaFX SDK. Básicamente es una lista observable que brinda la opción de adjuntar oyentes para cambiar el contenido de la lista. ObservableList tiene un papel importante en el desarrollo de JavaFX porque se usa para componentes principales como TableView, ComboBox, etc.

ObservableList es como un ArrayList pero que pertenece a JavaFX. Su diferencia principal es que informa a JavaFX cuando se ha producido un cambio. De esa manera, JavaFX podrá actualizar la vista. En cambio, si utilizamos ArrayList, no podremos saber cuando se ha producido un cambio.

Por lo tanto, cuando vayamos a visualizar una lista de objetos en una TableView o bien en otra vista, deberemos de utilizar una colección Observable, es decir, una colección que contiene oyentes i otros componentes necesarios para interaccionar con la vista.

A continuación tenéis la vista a generar para realizar esta práctica. En ella deberemos de crear una tabla con columnas. Luego creamos el modelo para asignarlo a la tabla. Almacenaremos los objetos del modelo en un ObservableList y asociaremos los atributos del objeto con los de la tabla. Una vez añadido un objeto a la ObservableList tendremos que setearlo para que se visualice el cambio en la tabla.

The screenshot shows a JavaFX application window with a light gray background. On the left side, there is a form with three text input fields labeled "Nombre", "Apellidos", and "Edad". Below these fields is a button labeled "Agregar Persona". To the right of the form is a TableView with three columns: "Nombre", "Apellidos", and "Edad". The table is currently empty, displaying the text "No content in table".

Una vez te funcione, modifícalo para que acepte valores correctos (expresiones regulares).

## 11.- LISTA DE PERSONAS . MODIFICAR - ELIMINAR

The screenshot shows a JavaFX application window titled "GESTIÓN DE PERSONAS". It features a form on the left with input fields for "Nombre", "Apellidos", and "Edad", and a button labeled "Agregar Persona". To the right is a TableView with columns "Nombre", "Apellidos", and "Edad", which is currently empty and shows "No content in table". At the bottom of the window, there are two buttons: "Modificar" and "Eliminar".

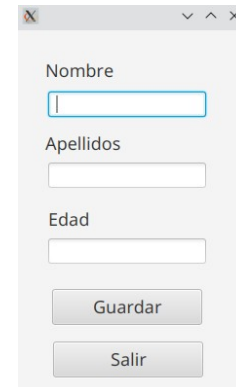
## 12.- LISTA DE PERSONAS. ABRIENDO VENTANAS

En esta práctica vamos a dividir en dos ventanas la práctica del apartado 10 (Lista de Personas). Las dos vistas con las que trabajaremos son:

PersonasVista.fxml



PersonasDialogVista.fxml

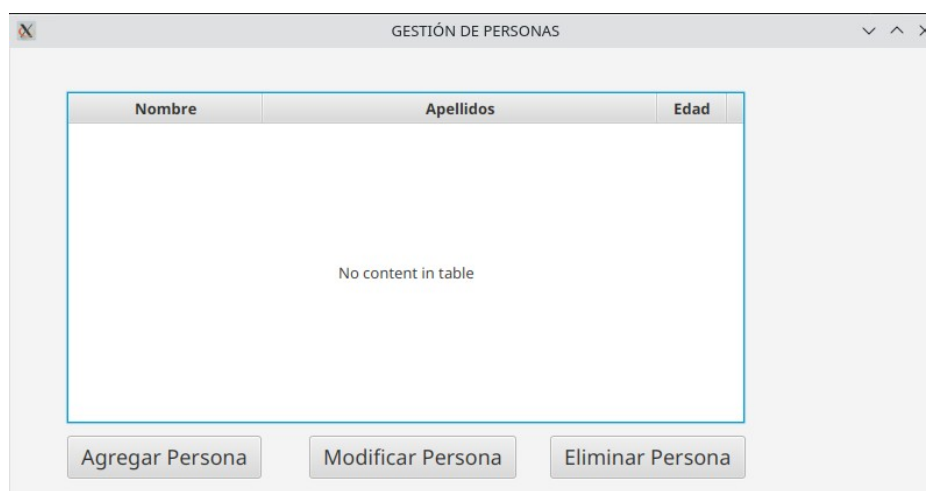


En hacer un clic en Agregar, aparecerá la ventana para agregar una persona. Una vez introducida, si pulsamos guardar se cerrará la ventana y volverá a la anterior donde aparecerá la persona introducida.

En este caso, cada vista tendrá su controlador. El modelo continua siendo el mismo (la clase Persona).

## 13.- MODIFICAR-ELIMINAR DESDE OTRA VENTANAS

Amplia el ejercicio anterior con las opciones de Modificar y Eliminar.



## 14.- FILTRAR Y ORDENAR REGISTROS

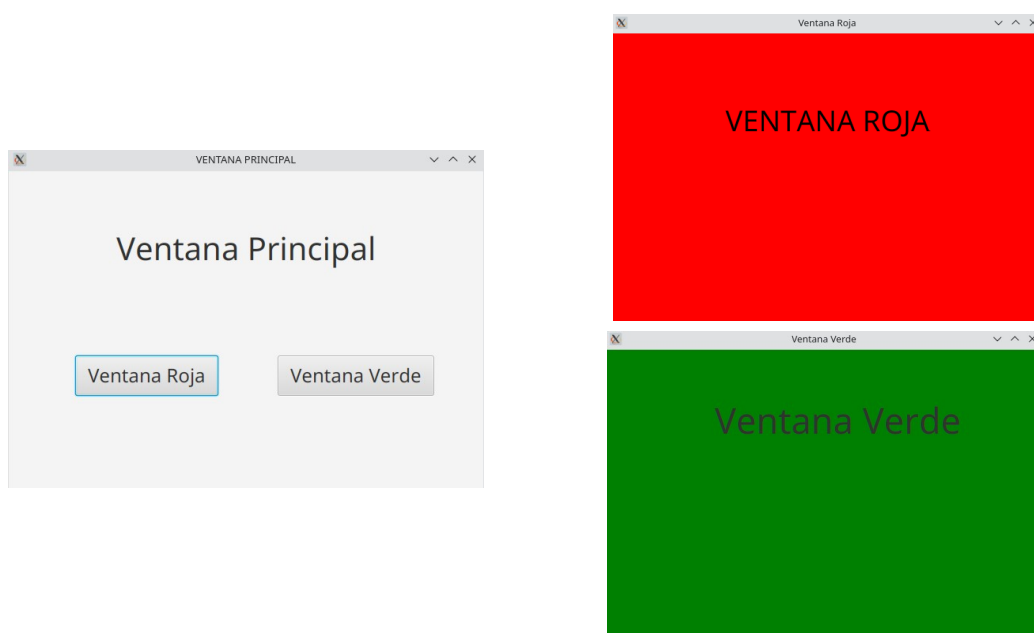
En este caso, utilizaremos un TextField llamado txtFiltrarPorNombre y programaremos el evento onKeyReleased. Con ello vamos a indicar que mientras vamos escribiendo vamos filtrando.

De las prácticas anteriores sabemos que todas las personas las tenemos almacenadas en un ObservableList. La idea es tener otro ObservableList de personas en donde colocaremos las personas que vamos filtrando.

Además, una vez hecho el filtro, tanto si agrego, modifico o elimino, si no cumple con el filtro no debe aparecer en la lista del filtro, pero siempre se debe hacer del original.

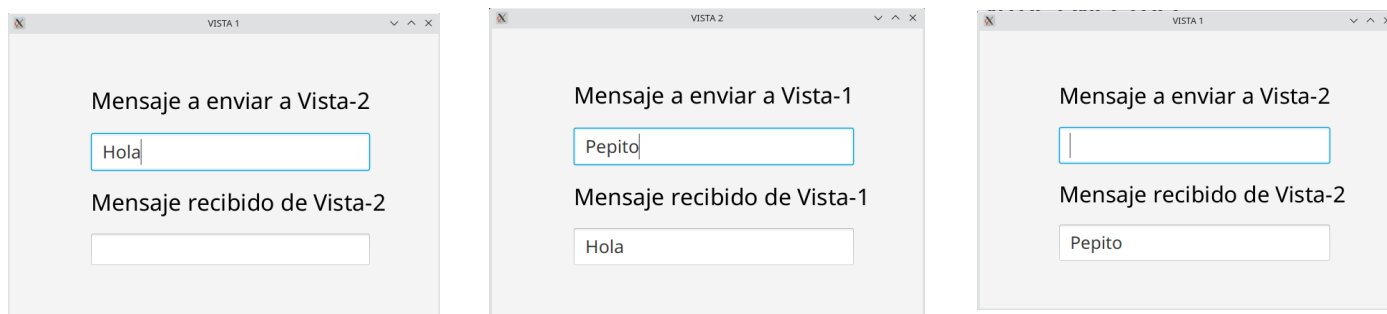


## 15.- NAVEGAR ENTRE VENTANAS



## 16.- COMUNICACIÓN ENTRE VENTANAS

En este ejercicio veremos como podemos comunicar diferentes ventanas enviando información de una a la otra. Nos basaremos en las siguientes ventanas simples:



## 17.- JUEGO DE LA BOLA

Realiza la siguiente práctica donde tenemos que mover la bola por la tabla (*container GridPane*) haciendo clic en los botones (otra tabla *GridPane*) y en las flechas del teclado (*on Key Pressed*).

