

UD07. Interfaces Gráficas de Usuario con JavaFX

DAM1-Programación 2024-25

Objetivos	2
1. Introducción	3
1.1. ¿Qué es una interfaz gráfica de usuario (GUI)?	3
1.2. Historia de las GUI en Java: AWT, Swing y JavaFX	3
1.3. Ventajas de utilizar JavaFX	3
2. Instalación y configuración de JavaFX	4
3. Estructura básica de una aplicación JavaFX	5
3.1. Clase Application	5
3.2. Stage, Scene, Node	5
3.3. Ejemplo: "Hola, mundo" con Java FX	6
4. Controles, eventos y layouts en JavaFX	7
4.1. Controles comunes (Button, Label, TextField, etc.)	7
4.2. Eventos y manejadores de eventos	9
4.3. Contenedores o Layouts (HBox, VBox, GridPane, etc.)	12
4.4. Para seguir ampliando	14
5. JavaFX y CSS	15
6. FXML, Scene Builder y MVC	16
6.1. Scene Builder	16
6.2. Patrón Modelo Vista Controlador (MVC)	17
6.3. Creando una aplicación MVC con Scene Builder y FXML	17
Ejemplo Saludo	18
Ejemplo Empresas - CRUD	20
Anexos y Enlaces	21
¿Hace falta el método main() en JavaFX?	21
Como ajustar el tamaño de la ventana a una nueva escena	21
Botones por defecto	21
Orden de tabulación	21
Como ejecutar un jar de aplicación JavaFX desde la consola de Windows	21
Como generar un ejecutable .exe a partir de un .jar	22

Objetivos

1. Comprender los conceptos básicos de las interfaces gráficas de usuario (GUI) en Java.
2. Crear aplicaciones de escritorio básicas utilizando JavaFX.
3. Utilizar componentes y layouts comunes en JavaFX para diseñar interfaces de usuario.
4. Practicar la creación de interfaces gráficas de usuario mediante ejercicios resueltos y propuestos.

- [Getting Started with JavaFX](#) (openjfx.io - Configuración del entorno)
 - DOC: [Overview \(JavaFX 25\)](#)
- [JavaFX Tutorial](#) (jenkov.com)
- [JavaFX Tutorial](#) (www.tutorialspoint.com)
- [Tutoriales JavaFX](#)

1. Introducción

1.1. ¿Qué es una interfaz gráfica de usuario (GUI)?

Una interfaz gráfica de usuario (GUI) es un conjunto de elementos visuales y controles que permiten a los usuarios interactuar con un programa de forma visual e intuitiva. Los elementos de una GUI incluyen ventanas, botones, menús, cuadros de texto, imágenes, entre otros.

1.2. Historia de las GUI en Java: AWT, Swing y JavaFX

Java ha tenido tres principales opciones para la creación de interfaces gráficas de usuario: AWT (Abstract Window Toolkit), Swing y JavaFX. **AWT** fue la primera librería de GUI que se desarrolló para Java, pero fue limitada en términos de capacidad y personalización. Luego vino **Swing**, que es una extensión de AWT que proporciona una mayor flexibilidad y personalización en la creación de interfaces gráficas de usuario. Finalmente, **JavaFX** se introdujo en 2008 como una nueva plataforma de GUI que permite la creación de aplicaciones ricas en contenido multimedia y gráficos.

1.3. Ventajas de utilizar JavaFX

JavaFX ofrece varias ventajas en comparación con Swing y AWT. Primero, permite la creación de interfaces gráficas de usuario **más atractivas y modernas** con gráficos en 2D y 3D, animaciones y efectos visuales. Además, JavaFX está más integrado con la **tecnología web**, lo que permite la integración de contenido web en las aplicaciones. También es **más fácil** de usar para desarrolladores con experiencia en HTML, CSS y JavaScript. Por último, JavaFX se basa en una arquitectura de escena que permite la **separación de la presentación y la lógica de la aplicación**, lo que facilita la creación y el mantenimiento de aplicaciones grandes y complejas.

2. Instalación y configuración de JavaFX

Desde la versión 11 de Java, Oracle no incluye JavaFX en el SDK. Hay que descargarlo aparte.

OpenJFX is an open source, next generation client application platform for desktop and embedded systems for use with the JDK. This is the open source project where we develop JavaFX. OpenJFX is free software, licensed under the GNU General Public License with Classpath Exception. The OpenJFX Project is part of the OpenJDK open-source Community.

- **Download OpenJFX Builds**
 - Gluon: [JavaFX - Gluon](#)
 - Oracle: [JavaFX 20 GA Release](#)
- [Getting Started with JavaFX](#) (openjfx.io - Configuración del entorno)

La recomendación más sencilla para empezar a desarrollar en JavaFX es usar (descargar y configurar) un JDK de Azul Zulu que incluya las librerías de JavaFX. Si no disponemos de permisos de instalación podemos descargar la versión portable .zip

- **Download Azul JDKs**
 - [Zulu Gets Graphical With OpenJFX - Azul Systems Blog](#)

1. **Instalación del SDK de JavaFX.** Para desarrollar aplicaciones en JavaFX, se necesita instalar el SDK de JavaFX.
 - a. (Opcional) Configurar variable de entorno **JAVA_HOME**
2. **Configuración del entorno de desarrollo (IDE)**
3. **Creación de un proyecto JavaFX** y comprobación de que detecta las clases de JavaFX

3. Estructura básica de una aplicación JavaFX

- [JavaFX Tutorial](http://jenkov.com) (jenkov.com)
- [JavaFX Tutorial](http://www.tutorialspoint.com) (www.tutorialspoint.com)

3.1. Clase Application

- [Your First JavaFX Application](#)

Para crear una aplicación JavaFX debemos “extender” la clase [Application](#).

La clase Application contiene el método abstracto [start\(\)](#) que deberemos implementar a continuación con el contenido de la aplicación.

Podemos lanzar la aplicación invocando el método [launch\(\)](#) desde el método main().

```
public class App extends Application {  
  
    public static void main(String[] args) throws Exception {  
        launch();  
    }  
  
    @Override  
    public void start(Stage stage) throws Exception {  
        // Contenido de la aplicación  
        // ...  
    }  
}
```

3.2. Stage, Scene, Node

- [Stage](#), [Scene](#) y [Node](#)

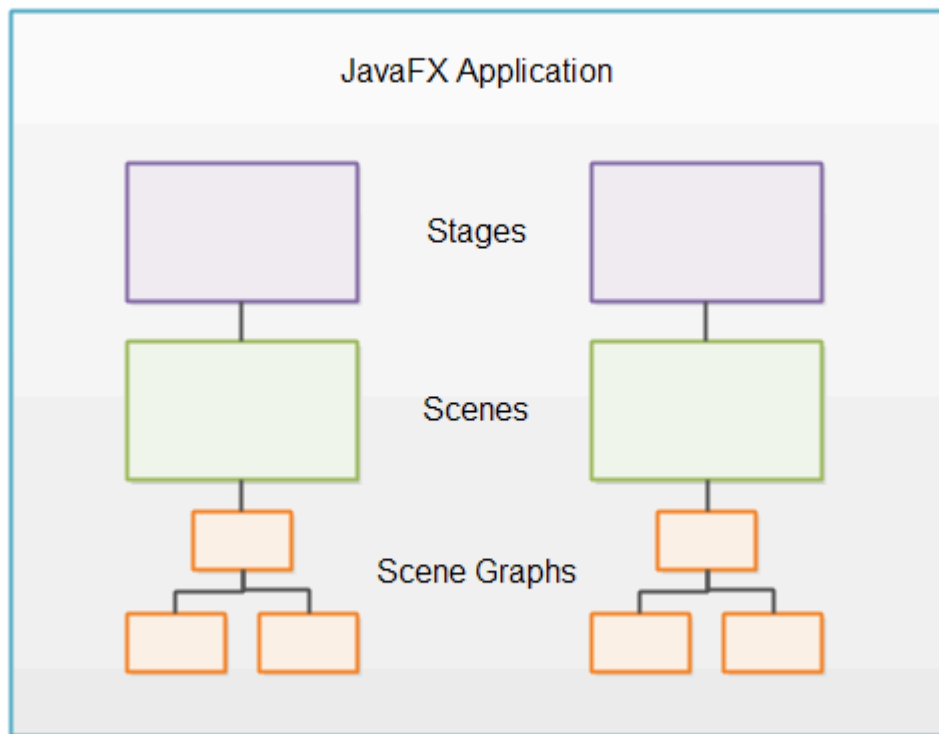
La creación de aplicaciones en JavaFX utiliza la analogía de un escenario donde se representan escenas compuestas por una variedad de componentes.

La clase [Stage](#) representa al escenario, es decir, la ventana en la que se representan las escenas. El método start() incluye un parámetro de tipo Stage que referencia a la ventana principal de la aplicación.

La clase [Scene](#) representa la escena que se “representa” en el escenario o ventana. Solo podemos añadir una escena a la vez en un escenario, pero podemos crear muchas e intercambiarlas.

La clase [Node](#) representa los elementos que podemos añadir a la escena. Los nodos pueden ser de muchos tipos, controles y contenedores por ejemplo, y se organizan en una jerarquía en forma de árbol, llamada grafo de nodos o de la escena. Los nodos que pueden

contener otros nodos son subclases de la clase [Parent](#). Uno de esos nodos toma el papel de **nodo raíz (root)** del grafo y será el que se añada a la escena.



3.3. Ejemplo: "Hola, mundo" con Java FX

1. Crear la clase principal que debe extender la clase `Application` y que será el punto de entrada a la aplicación.
2. Implementar método abstracto `start()` que se ejecutará al lanzar la aplicación.
 - a. Crear grafo de nodos, indicando el raíz (root)
 - b. Crear escena (scene) y añadir al escenario (stage)
 - c. Mostrar escenario
3. Invocar método `launch()` desde el `main()`. (En algunas circunstancias este paso puede no ser necesario)

Ejemplo1. Crear una ventana simple con un botón ([Button](#)) que, al hacer clic, muestre un mensaje en consola.

Amplía el programa para que al pulsar cambie el texto del botón y vaya contando el número de pulsaciones.

Ejemplo2. Crear una ventana con un campo de texto ([TextField](#)) y un botón que, al hacer clic ([setOnAction\(\)](#)) imprima el texto por consola. Amplía para que imprima el texto en una etiqueta ([Label](#)) en la pantalla.

Investiga como utilizar un contenedor como nodo raíz de la escena, por ejemplo [VBox](#), para organizar los controles.

Crea un programa que imprima un saludo

4. Controles, eventos y layouts en JavaFX

4.1. Controles comunes (Button, Label, TextField, etc.)

Algunos controles comunes en JavaFX son:

- [Button](#)
- [Label](#)
- [TextField](#)
- [TextArea](#)
- `CheckBox`
- `ChoiceBox`
- [ComboBox](#)
- `ColorPicker`
- `DatePicker`
- [ImageView](#)
- [ListView](#)
- `ToggleButton`
- [FileChooser](#)
- etc.

Ejemplo3. Crear una ventana con dos campos de texto y un botón que, al hacer clic, muestre la suma de los números ingresados en un tercer campo de texto o en una etiqueta.

Modifica el programa para que la suma se actualice al modificar el contenido de los campos de texto de los operandos. Pista: usa [setOnKeyTyped\(\)](#)

Conversor. Crea una aplicación que permita convertir cantidades de un sistema de medida a otro. Por ejemplo, de euros a dólares (1 € = 1,13 \$), de libras a kilos (1 libra = 0.4535924 kilos), de millas a kilómetros (1 milla = 1.609344 kilómetros), etc. Experimenta con nuevos controles para ampliar el programa y que permita convertir en ambos sentidos o cambiar de un tipo de conversión a otra.

Ejemplo4_ColorFondo. Crear una ventana con un campo de texto y un botón que, al hacer clic, cambie el color de fondo de la ventana (o de una etiqueta) al color especificado en el campo de texto. (Investiga [setBackground\(\)](#) de la clase [Region](#) o [setStyle\(\)](#) de la clase [Node](#) para cambiar el color de fondo de un nodo).

Ejemplo5_ColorCombo. Crear una ventana con una lista desplegable de colores ([ComboBox](#)) y un botón que, al hacer clic, cambie el color de fondo de la ventana al color seleccionado en la lista desplegable.

Para prescindir del botón y cambiar el color con sólo seleccionarlo en el desplegable, investiga el evento [onAction](#) del `ComboBox` que se dispara al modificar la propiedad [value](#).

Ejemplo6. Crear una ventana con un área de texto ([TextArea](#)), un campo de texto y un botón que, al hacer clic, agregue el texto del campo de texto a la lista de elementos en el área de texto.

Ejemplo7_ListaNotas. Crea una aplicación que permita introducir en cuadros de texto las notas de un grupo de alumnos, una a una, y que al mismo tiempo las vaya mostrando en orden alfabético de nombre de alumno en un control de tipo lista ([ListView](#)) y que muestre también la media de la clase, la nota más alta y la más baja.

Crea otros ejemplos de aplicaciones para probar otros controles

- **Cumple.** Crea una aplicación que permita introducir la fecha de tu cumpleaños ([DatePicker](#)) y calcule y muestre en una etiqueta cuantos días faltan.
- Crea una aplicación que permita seleccionar un color ([ColorPicker](#)) y realice una acción con ese color (cambiar el fondo, color de letra, hacer un dibujo...)
- Utiliza [TreeView](#) para mostrar las características de un grupo de personajes de rol.
- Utiliza [TableView](#) para mostrar la misma información anterior.

Sobre el control/clase [ListView<T>](#):

- Puede contener una lista de cualquier tipo de elementos T.
- Los elementos se añaden, eliminan, gestionan, etc. mediante una [ObservableList<T>](#) a la que se puede acceder con el método [getItems\(\)](#).
- Los elementos se visualizarán según el método `toString()` de su clase, y cualquier cambio sobre la lista se visualizará automáticamente en la interfaz gráfica.
- Para capturar la selección de elementos de la lista, bien a través del ratón, teclado, etc. y ejecutar código en respuesta se debe acceder a la propiedad [SelectionMode](#).

Por ejemplo: el siguiente código imprime en consola el nuevo objeto recién seleccionado en la lista:

```
listView.getSelectionModel().selectedItemProperty().addListener((obs, oldSelection, newSelection) ->
    System.out.println(newSelection));
```


4.2. Eventos y manejadores de eventos

Una de las características más importantes de JavaFX es la capacidad de manejar eventos de entrada del usuario, como clics de mouse, pulsaciones de teclas, movimientos de mouse, etc.

- Un **evento** en JavaFX es una acción realizada por el usuario, como hacer clic en un botón, mover el mouse, etc.
- Un **manejador de eventos** en JavaFX es un objeto con un método que se ejecuta cuando se produce un evento.

Clases e interfaces principales

- La clase [Event](#) es la superclase de todos los eventos en JavaFX. Contiene información básica sobre el evento, como su origen y su tipo.
- La interfaz [EventHandler<T extends Event>](#) es una interfaz funcional genérica que define un método `handle()` que se ejecuta cuando se produce un evento. La letra "T" representa el tipo de evento que manejará el método.

Tipos de Eventos

En JavaFX, existen diferentes tipos de eventos que el usuario puede provocar y que desde el código podemos "manejar" para generar una respuesta. A continuación, se describen algunos de los tipos de eventos más comunes:

- Evento de Acción ([ActionEvent](#)): Este evento se produce cuando se realiza una acción, como hacer clic en un botón, seleccionar un elemento de un menú, etc.
- Evento de Entrada ([InputEvent](#))
 - Evento de Teclado ([KeyEvent](#)): Este evento se produce cuando se pulsa o se suelta una tecla del teclado.
 - Evento de Mouse ([MouseEvent](#)): Este evento se produce cuando se realiza una acción con el mouse, como hacer clic en un objeto, mover el mouse sobre un objeto, arrastrar un objeto, etc.
- Evento de Ventana ([WindowEvent](#)): Este evento se produce cuando se produce un cambio en una ventana, como cuando se cierra, se minimiza o se maximiza.

Como capturar un evento en un objeto (Stage, Scene, Node..) e implementar el manejador asociado

Para responder a un evento en JavaFX, debemos usar un objeto que implemente la interfaz [EventHandler](#). Esta interfaz incluye un único método abstracto, **handle()**, donde escribiremos el código en respuesta al evento.

Para capturar un evento en un objeto (un nodo, una escena o una ventana) e invocar al manejador cuando dicho evento se produzca podemos usar el método del objeto **addEventHandler()**. Este método tiene dos parámetros: el tipo de evento ([EventType](#)) y el objeto manejador. Para instanciar el objeto manejador podemos usar una clase anónima o una expresión lambda. Por ejemplo:

```
/** Botón que al hacer click imprime un mensaje en consola */
```

```

/** Creando el manejador con una clase anónima */

Button boton = new Button("Haz clic aquí");
EventHandler<Event> manejador = new EventHandler<>() {
    @Override
    public void handle(Event event) {
        System.out.println("Has hecho click!");
    }
};
boton.addEventHandler(MouseEvent.MOUSE_CLICKED, manejador);

/** El mismo ejemplo anterior usando una expresión lambda para
implementar el manejador */

Button boton = new Button("Haz clic aquí");
boton.addEventHandler(MouseEvent.MOUSE_CLICKED,
    e -> System.out.println("Has hecho click!"));

```

Otra manera de establecer un manejador de evento para un control es usando su método setter respectivo. Este método tiene la siguiente forma **setOnXXX()** donde **XXX** corresponde al tipo de evento para el cual deseamos agregar el controlador. Por ejemplo, el código anterior podríamos reescribirlo así:

```

/** El mismo ejemplo anterior usando el método setter del manejador */

Button boton = new Button("Haz clic aquí");
boton.setOnMouseClicked(e -> System.out.println("Has hecho click!"));

```

Otros ejemplos:

```

/** Campo de texto que al presionar las teclas del teclado imprime un mensaje en
consola con el código de tecla pulsada */

TextField campoTexto = new TextField();
campoTexto.setOnKeyPressed((KeyEvent evento) ->
    System.out.println("Se ha pulsado la tecla " + evento.getCode()));

```

Múltiples objetos pueden responder a un evento.

En una interfaz gráfica puede haber varios objetos del grafo de la escena (del árbol de nodos) que respondan a un mismo evento. En estos casos *el evento se propagará desde el nodo en que se originó hasta el nodo raíz ejecutando en orden todos los manejadores de eventos que se hayan definido.*

Por otra parte también existen los denominados **filtros de eventos** que se definen y asocian a los objetos de forma similar a los manejadores de eventos pero en este caso con el método **addEventFilter()**. Se pueden añadir filtros para un mismo evento a distintos objetos del grafo de la escena. *El código de los filtros se ejecutará antes que los manejadores y siguiendo el orden de propagación desde el nodo raíz hasta el nodo origen del evento.*

A lo largo de estas dos secuencias de ejecución de los filtros y luego de los manejadores podemos detener la propagación del evento con el método **consume()**.

Secuencia de ejecución de filtros y manejadores de evento:

1. Se ejecutan los **filtros** de evento desde el nodo raíz hasta el nodo origen del evento.
2. Se ejecutan los **manejadores** de evento desde el nodo origen hasta el raíz.

Esta propagación se detiene si invocamos el método **consume()** del evento.

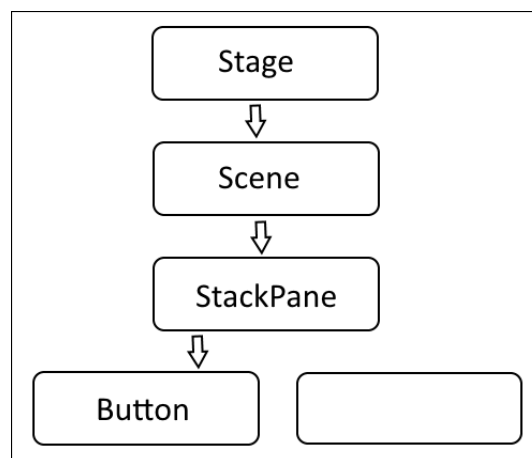
Ejemplo:

```
Button btn = new Button();
btn.setText("Button");
btn.addEventHandler(MouseEvent.MOUSE_CLICKED, e -> System.out.println("Button
Event Handler"));

StackPane root = new StackPane();
root.getChildren().add(btn);
root.addEventFilter(MouseEvent.MOUSE_CLICKED, e -> System.out.println("StackPane
Event Filter"));

Scene scene = new Scene(root, 300, 250);

primaryStage.setTitle("JavaFX Manejo de Eventos");
primaryStage.setScene(scene);
primaryStage.show();
```



Secuencia de captura del evento (*EventFilter*).

A continuación se llevará a cabo la secuencia inversa de propagación del evento (*EventHandler*).

En el siguiente ejemplo se utiliza `consume()` para impedir que se introduzcan caracteres no numéricos en un campo de texto:

```
/** Campo de texto que al presionar las teclas del teclado imprime
 * un mensaje en consola con el código de tecla pulsada */
TextField campoTexto = new TextField();
campoTexto.setOnKeyPressed((KeyEvent evento) ->
System.out.println("Se ha pulsado la tecla " + evento.getCode()));
```

```
Scene scene = new Scene(campoTexto, 400, 400);

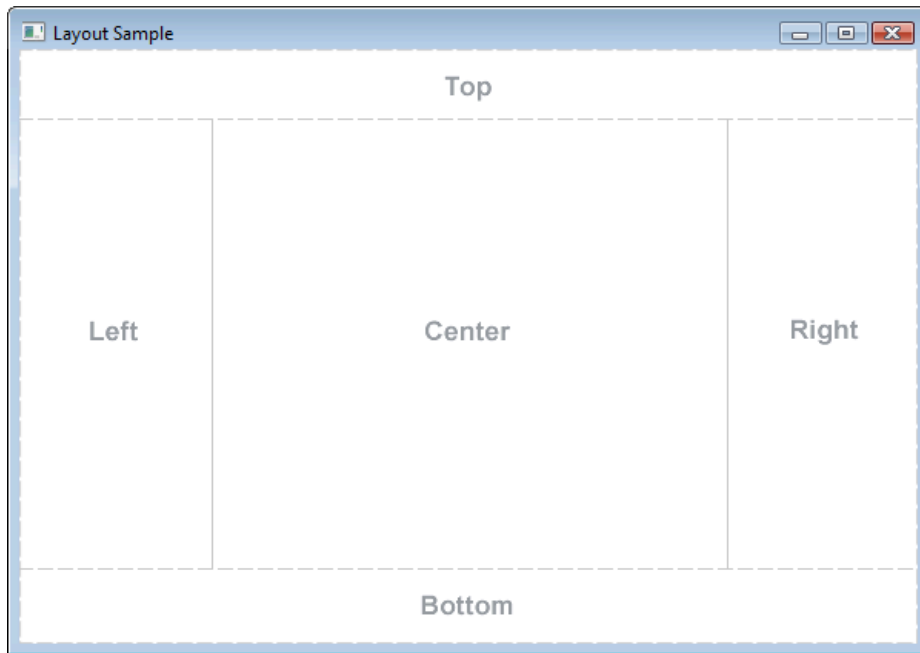
/** Filtro de evento que detiene la propagación de un evento de teclado
 * cuando el caracter escrito no es un número */
scene.addEventFilter(KeyEvent.KEY_TYPED, e -> {
    if (!Character.isDigit(e.getCharacter().charAt(0))) {
        e.consume();
    }
});
```

4.3. Contenedores o *Layouts* (HBox, VBox, GridPane, etc.)

JavaFX proporciona varios contenedores básicos que se utilizan para organizar y mostrar nodos en la interfaz gráfica de usuario. Los contenedores básicos más comunes son los siguientes:

- **[Pane](#)**: es un contenedor genérico que se utiliza como base para otros contenedores más específicos. Los nodos se agregan al Pane utilizando el método `getChildren().add(Node)`.
- **[HBox](#)**: es un contenedor que organiza los nodos horizontalmente. Puedes agregar nodos utilizando el método `getChildren().add(Node)`.
- **[VBox](#)**: es un contenedor que organiza los nodos verticalmente. Puedes agregar nodos utilizando el método `getChildren().add(Node)`.
- **[GridPane](#)**: es un contenedor que organiza los nodos en filas y columnas. Puedes agregar nodos a una celda del GridPane utilizando los métodos `add(Node, column, row)` o `add(Node, column, row, colSpan, rowSpan)`, donde `column` y `row` son las coordenadas de la celda en la que se agregará el nodo.
- **[TilePane](#)**: es un contenedor que organiza los nodos en una cuadrícula fija, donde todos los nodos tienen el mismo tamaño. Puedes agregar nodos utilizando el método `getChildren().add(Node)`.
- **[AnchorPane](#)**: es un contenedor que permite ubicar nodos en una posición absoluta dentro de su área. Puedes establecer las posiciones de los nodos utilizando los métodos `setTopAnchor(Node, Double)`, `setBottomAnchor(Node, Double)`, `setLeftAnchor(Node, Double)` y `setRightAnchor(Node, Double)`.
- **[BorderPane](#)**: es un contenedor que divide su área en cinco regiones: la superior, la inferior, la izquierda, la derecha y la central. Puedes agregar nodos a cualquiera de estas regiones utilizando los métodos `setTop(Node)`, `setBottom(Node)`,

setLeft(Node), setRight(Node) y setCenter(Node).



- **FlowPane:** es un contenedor que organiza sus nodos en un flujo donde los nodos se ajustan automáticamente en distintas filas en función del ancho del contenedor. Puedes agregar nodos utilizando el método `getChildren().add(Node)`.
- **StackPane:** es un contenedor que apila sus nodos uno encima del otro. Puedes agregar nodos utilizando el método `getChildren().add(Node)`.
- **DialogPane:** es un contenedor especializado que se utiliza para crear diálogos de usuario personalizados. Un diálogo es una ventana emergente que se utiliza para mostrar información, solicitar entrada del usuario o realizar una acción. El `DialogPane` proporciona un diseño y una estructura predefinidos para un diálogo, incluyendo botones de acción estándar como Aceptar, Cancelar y Cerrar. El `DialogPane` es un contenedor diseñado para ser utilizado con la clase `javafx.scene.control.Dialog`. Puedes personalizar el contenido del `DialogPane` agregando nodos a su propiedad `content`, que es un `ObservableList<Node>`.

Actividad 8. Crear una ventana que muestre un tablero de ajedrez, es decir una cuadrícula de 8 x 8 “casillas” blancas y “negras” cuadradas.

Actividad 9. Ejercicio con **HBox**:

Crema una aplicación que contenga un `HBox` con dos `ImageView`. Los `ImageView` deben mostrar dos imágenes diferentes. Al hacer clic en cada imagen, se debe mostrar un mensaje en la consola que indique qué imagen ha sido pulsada.

Actividad 10. Ejercicio con **VBox**:

Crema una aplicación que contenga un `VBox` con tres botones, cada uno de ellos con un texto diferente (por ejemplo, "Botón 1", "Botón 2" y "Botón 3"). Al hacer clic en cada botón, se debe mostrar un mensaje en la consola que indique qué botón ha sido pulsado.

Ejemplo11_BorderPane. Ejercicio con **BorderPane**:

Crea una aplicación que contenga un BorderPane con un [menú](#) en la parte superior y un panel en el centro. El menú debe tener dos opciones: "Rojo" y "Verde". Al seleccionar la opción "Rojo", el panel debe cambiar su color de fondo a rojo; al seleccionar la opción "Verde", el panel debe cambiar su color de fondo a verde.

Actividad 12. Ejercicio con **FlowPane**:

Crea una aplicación que contenga un FlowPane con varias ImageView. Las ImageView deben mostrar diferentes imágenes. Al hacer clic en cada imagen, se debe mostrar un mensaje en la consola que indique qué imagen ha sido pulsada.

Actividad 13. Ejercicio con **GridPane**:

Crea una aplicación que contenga un GridPane con varios botones. Los botones deben estar distribuidos en el GridPane de manera que formen una cuadrícula. Al hacer clic en cada botón, se debe mostrar un mensaje en la consola que indique qué botón ha sido pulsado y en qué posición se encuentra dentro del GridPane.

Actividad 14. Ejercicio con **DialogPane**:

Crea una aplicación que utilice DialogPane para mostrar varios cuadros de diálogo con diferentes configuraciones..

4.4. Para seguir ampliando

- Cuadros de diálogo y ventanas emergentes
- Dibujar en [Canvas](#)
- Audio y Vídeo
- Contenido web con [WebView](#)
- etc.

5. JavaFX y CSS

- [JavaFX CSS Styling](#)
- [JavaFX CSS Reference Guide](#)
- [Introducción a CSS en JavaFX](#)

El objetivo de JavaFX CSS es permitir que los desarrolladores web familiarizados con CSS para HTML utilicen CSS para personalizar y desarrollar temas para controles JavaFX de una manera natural.

Con este fin, a todos los nombres de propiedades CSS de JavaFX se les añade el prefijo **"-fx-"**.

Utilizar CSS para dar estilo a aplicaciones JavaFX ayuda a separar el estilo (apariencia) del código de la aplicación. Esto produce un código más limpio y hace que sea más fácil cambiar el estilo de la aplicación. Así, no es necesario modificar el código Java para cambiar el estilo. También puedes cambiar el estilo para muchos componentes a la vez, utilizando hojas de estilo CSS compartidas.

Ejemplos:

Podemos aplicar un estilo a un único nodo con el método `setStyle()`

```
Button btnCrear = new Button("Crear");  
btnCrear.setStyle("-fx-color: red;");
```

Podemos aplicar una hoja de estilos a toda una escena con `getStyleSheet().add()`

`CrearPersonaje.css`

```
Button {  
    -fx-color: blue;  
}
```

...

```
scene.getStylesheets().add("CrearPersonaje.css");
```

6. FXML, Scene Builder y MVC

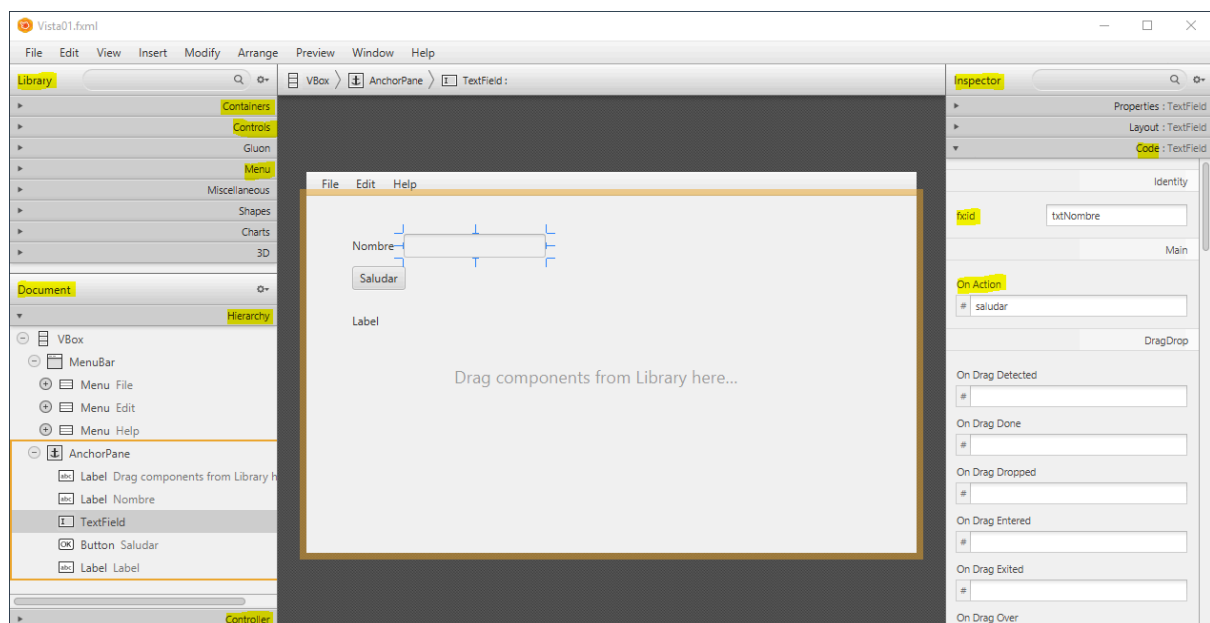
- [JavaFX FXML](#)
- [Mastering FXML: About This Tutorial | JavaFX 2 Tutorials and Documentation](#)
- [JavaFX Scene Builder User Guide](#)
- [Part II: Building a JavaFX Application Using Scene Builder \(Release 2\)](#)

JavaFX FXML es un formato XML que permite diseñar interfaces gráficas de manera similar a las interfaces web en HTML. FXML te permite separar el código de diseño de la interfaz JavaFX del resto del código de la aplicación. Esto limpia tanto el código de diseño como el resto del código de la aplicación.

FXML se puede utilizar tanto para componer el diseño de toda la GUI de una aplicación, como solo una parte de ella como, por ejemplo, el diseño de una parte de un formulario, pestaña, diálogo, etc.

6.1. Scene Builder

Scene Builder es una herramienta de diseño gráfico que se puede utilizar con cualquier IDE y que permite diseñar y editar interfaces gráficas de usuario JavaFX de forma visual, generando el correspondiente fichero FXML.



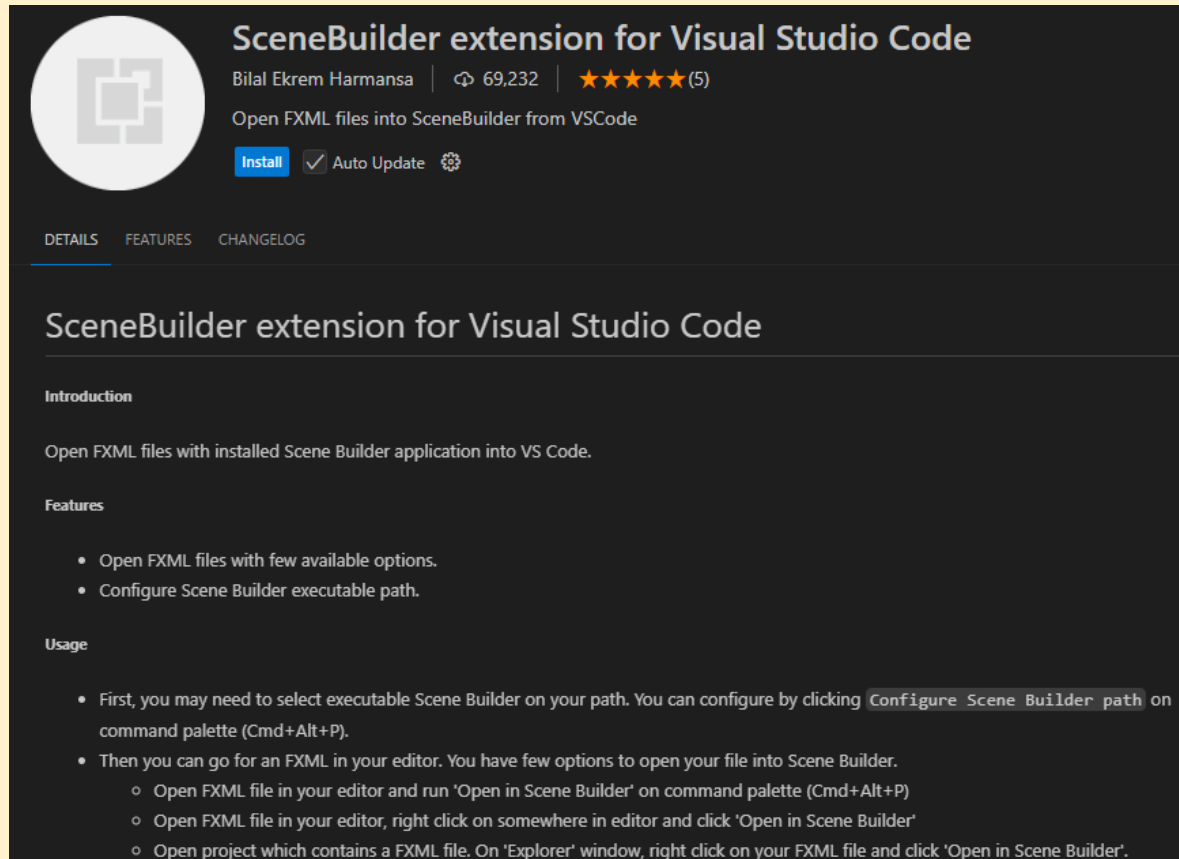
Ejemplo de diseño de vista en GlueX Scene Builder

Descarga e instala Scene Builder:

- [Scene Builder \(GlueX\)](#)
- [JavaFX Scene Builder 1.x Archive](#)

Configurar IDE

- **NetBeans:** Configurar JavaFX. Permite:
 - abrir archivos FXML en el Scene Builder.
 - crear y actualizar el controlador.
 - [Using Scene Builder with NetBeans IDE](#)
- **Visual Studio Code:** [SceneBuilder extension](#): Configurando la ruta del ejecutable permite abrir archivos FXML en el Scene Builder con un par de clicks.



SceneBuilder extension for Visual Studio Code
 Bilal Ekrem Harmansa | 69,232 | ★★★★★ (5)
 Open FXML files into SceneBuilder from VSCode
 Install Auto Update

SceneBuilder extension for Visual Studio Code

Introduction

Open FXML files with installed Scene Builder application into VS Code.

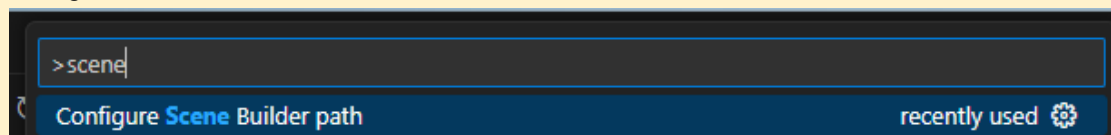
Features

- Open FXML files with few available options.
- Configure Scene Builder executable path.

Usage

- First, you may need to select executable Scene Builder on your path. You can configure by clicking `Configure Scene Builder path` on command palette (Cmd+Alt+P).
- Then you can go for an FXML in your editor. You have few options to open your file into Scene Builder.
 - Open FXML file in your editor and run 'Open in Scene Builder' on command palette (Cmd+Alt+P)
 - Open FXML file in your editor, right click on somewhere in editor and click 'Open in Scene Builder'
 - Open project which contains a FXML file. On 'Explorer' window, right click on your FXML file and click 'Open in Scene Builder'.

Configurar a través de la Paleta de Comandos del menú Ver



Indicar el ejecutable SceneBuilder.exe

6.2. Patrón Modelo Vista Controlador (MVC)

El patrón Modelo-Vista-Controlador (MVC) es un patrón de diseño de software que permite separar la lógica de negocio y la presentación de una aplicación, facilitando el mantenimiento y la escalabilidad. En JavaFX, podemos aplicar este patrón siguiendo estas tres componentes principales:

- **Modelo:** Representa la lógica de negocio y los datos de la aplicación. El modelo es independiente de la interfaz gráfica y no tiene conocimiento de las vistas y controladores.

- **Vista:** Representa la interfaz gráfica y se encarga de mostrar los datos del modelo al usuario. En JavaFX, las vistas son archivos FXML que describen la estructura y el diseño de la interfaz gráfica.
- **Controlador:** Gestiona la interacción entre el modelo y la vista. El controlador se encarga de actualizar el modelo cuando el usuario interactúa con la vista y actualizar la vista cuando el modelo cambia.

6.3. Creando una aplicación MVC con Scene Builder y FXML

Un esquema de trabajo habitual es el siguiente:

1. **Creamos la clase principal** que debe extender la clase `Application` y que será el punto de entrada a la aplicación y desde donde cargaremos y mostraremos la vista principal.
2. **Creamos el modelo** sobre el que funcionará la aplicación y que estará compuesto de una o varias clases que representan la lógica de negocio o los datos sobre los que trabajará la aplicación. Podemos instanciar los objetos principales del modelo en la misma clase principal.
3. **Diseñamos cada vista y creamos los ficheros FXML** correspondientes usando **Scene Builder** o escribiendo el código XML directamente. En cada vista indicamos el nombre de la **clase controladora** y asignamos identificadores a los controles que vayamos a necesitar referenciar en el código así como los métodos de los manejadores de eventos que vayamos a capturar e implementar.
4. **Creamos cada clase controladora** asociada a cada vista, declarando los controles y los métodos y, si es necesario, implementando la interfaz `Initializable` para definir el código que se ejecutará al inicio.

Para cargar las vistas utilizaremos la clase [FXMLLoader](#) cuyo método `load()` (estático o de instancia) cargará un fichero fuente FXML, indicado en la propiedad `location` definida en el constructor del objeto o con el método `setLocation()`, y devolverá el nodo raíz del grafo de nodos. Otro método importante del objeto `FXMLLoader` es `getController()` que devolverá una referencia al objeto controlador de la vista cargada.

Ejemplo Saludo

- **Crea la vista**, es decir, el fichero `VistaSaludo.fxml` en Scene Builder
 - Coloca los contenedores y controles.
Por ejemplo, un *AnchorPane* que contenga un campo de texto (*TextField*) donde el usuario introducirá un nombre, un botón (*Button*) que generará el saludo, y una etiqueta (*Label*) que mostrará el saludo y establece sus propiedades iniciales.
 - Nombra con un identificador apropiado (en Inspector > Code > `fx:id`) a los nodos que necesites referenciar desde el código (el campo de texto y la etiqueta).
 - Nombra los manejadores de los eventos que quieras capturar.
Por ejemplo “saludar” para el evento `On Action` del botón.
 - Nombre la clase controladora (en Document > Controller > Controller class)

- Guarda el fichero FXML en la carpeta del proyecto.
- **Crea el programa principal.**
 Extenderá la clase Application e implementará el método start(), dentro del cual utilizaremos la clase FXMLLoader para cargar la vista.

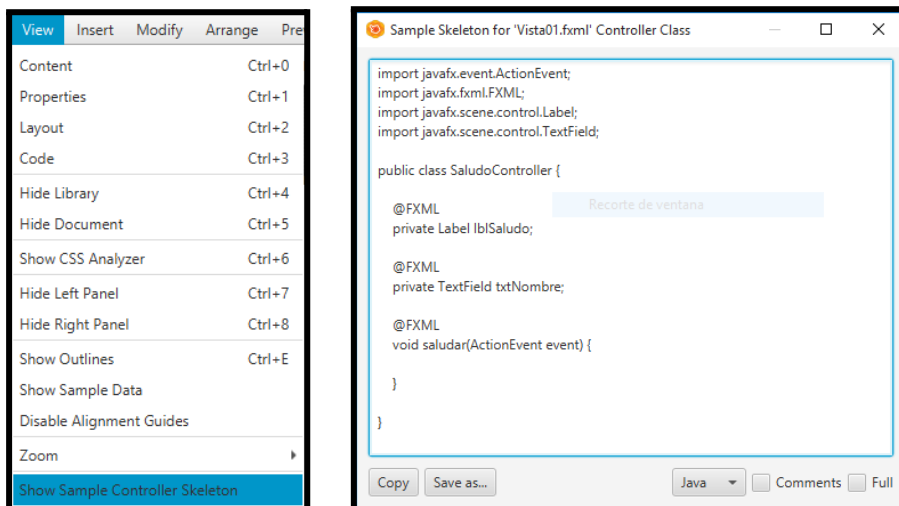
En ejemplo puede ser éste:

```
public class Saludo extends Application {

    public static void main(String[] args) {
        launch();
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Saludo");
        Scene scene = new Scene(FXMLLoader.load(Saludo.class.getResource("Vista01.fxml")));
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

- **Crea el controlador e implementa los métodos necesarios.**
 La opción del menú View > Show Sample Controller Skeleton facilita la creación inicial del controlador generando el código necesario con la declaración de los controles y métodos con identificadores.



El código completo puede ser como el que sigue:

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

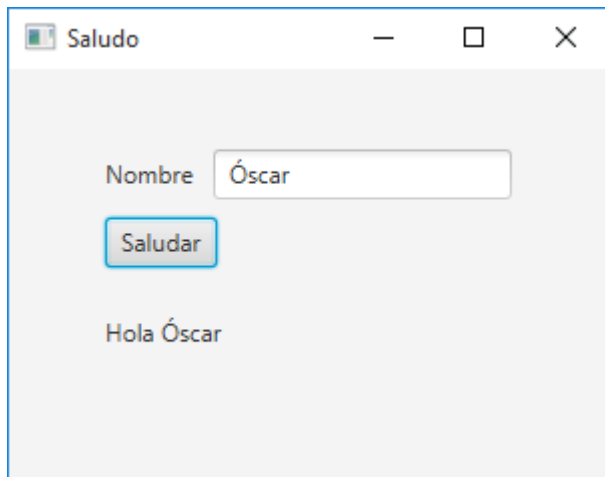
public class SaludoController {

    @FXML
    private Label lblSaludo;

    @FXML
    private TextField txtNombre;

    @FXML
    void saludar(ActionEvent event) {
        lblSaludo.setText("Hola " + txtNombre.getText());
    }
}
```

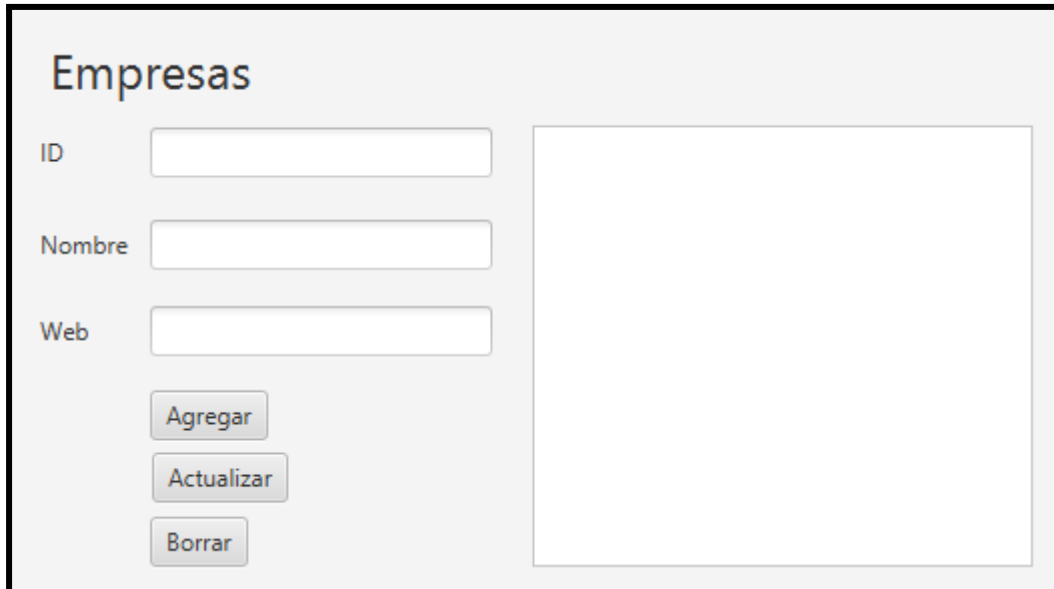
- El resultado final al ejecutar el programa, escribir un nombre y pulsar un botón:



Ejemplo Empresas - CRUD

EmpresaCRUD

Diseña una sencilla interfaz gráfica para gestionar las operaciones de altas, bajas y modificaciones de un listado de empresas. Crea un formulario como el siguiente:



- El formulario contiene 3 TextField, 3 Button y 1 ListView.
- Al iniciarse la aplicación se cargará un listado inicial de empresas en el ListView a la derecha.
- Al seleccionar una empresa se mostrarán sus datos en los campos del formulario de la izquierda.
- Al pulsar el botón Agregar se intentará añadir una nueva empresa con los datos de los campos de texto, actualizando la lista.
- Al pulsar el botón Actualizar se intentarán actualizar los datos de la nueva empresa mostrada en los campos de texto, actualizando la lista.
- Al pulsar el botón Borrar se intentará borrar la empresa mostrada en los campos de texto, actualizando la lista.
- Al pulsar cualquier de los botones se mostrará un cuadro de diálogo empergente con el resultado de la operación.

Inicializar vistas

Para inicializar una vista, por ejemplo con datos cargados de inicio en algún control, podemos hacerlo a través de su clase controladora. Para hacerlo la clase deberá implementar la interfaz **javafx.fxml.Initializable** e incluir el código de inicialización en el método abstracto **initialize()**.

Una solución:

<https://github.com/aguadodev/DAM1-Programacion-22-23/tree/master/UD07/Empresas>

Extras:

- Añadir una barra de menús en la parte superior con el menú Archivo y los elementos de menú: Nuevo, Abrir, Guardar, Guardar como... y Salir

- Como fichero de texto?? CSV??
 - Como ficheros binarios??
- Cargar los datos iniciales de un fichero y guardarlos al terminar.
- Añadir un control que permita acceder a las webs de las empresas. Y visualizarlas sin salir de la aplicación? => [JavaFX WebView](#)
- Ampliar el número de campos, etc.

Anexos y Enlaces

¿Hace falta el método main() en JavaFX?

Algunos alumnos que trabajan con IntelliJ entregan proyectos JavaFX que no incluyen el método main() y que no se pueden ejecutar desde el entorno de VScode.

- [JavaFX: ¿El método main es obligatorio o no? - Stack Overflow en español](#)

Como ajustar el tamaño de la ventana a una nueva escena

```
// Obtiene la ventana que contiene la escena y ajusta el tamaño  
scene.getWindow().sizeToScene();
```

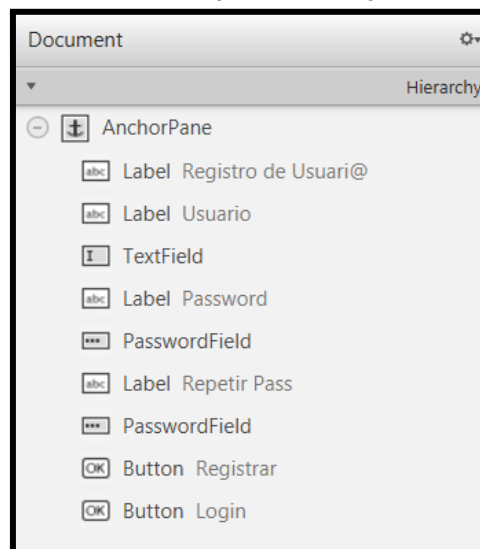
Botones por defecto

Botones que se activarán al pulsar las teclas ENTER o ESC.

- [Button \(JavaFX 19\)](#)
 - defaultButton
 - cancelButton

Orden de tabulación

En SceneBuildder el orden de tabulación a través de los controles de una escena viene dado por el orden de los controles en el panel de jerarquía. Ejemplo:



Como ejecutar un jar de aplicación JavaFX desde la consola de Windows

```
java -jar --module-path ruta_a_jdk/lib --add-modules  
javafx.controls,javafx.fxml fichero.jar
```

Ejemplo:

```
java -jar --module-path C:\Java\javafx-sdk-17.0.6\lib --add-modules  
javafx.controls,javafx.fxml Rol-JavaFX.jar
```

Como generar un ejecutable .exe a partir de un .jar

- [Launch4j](#)
- <https://docs.google.com/document/d/1YtzOne5-P59Duk6Cqpmggicpw-d4Jk99-4m3pEQjGBs/edit#>
- <https://docs.google.com/document/d/1kTB48mzvDbHRpwOrYQl88ls29u32HWFhe1v-A2iDQ/edit#>