

Interfaces Gráficas



Juan Carlos Pérez Rodríguez

Sumario

| | |
|---|----|
| Introducción..... | 4 |
| AWT y Swing..... | 5 |
| Swing..... | 7 |
| Uso del IDE para crear Interfaces gráficos..... | 8 |
| Contenedores..... | 14 |
| Eventos..... | 16 |
| Modelo Vista Controlador..... | 18 |
| MVC con Swing..... | 23 |
| JavaFX..... | 28 |
| Primer proyecto de Ejemplo de JavaFX..... | 37 |
| Separación de paquetes MVC en Netbeans JavaFX..... | 43 |
| Segundo proyecto: Calculadora..... | 47 |
| Pasos para crear la vista..... | 51 |
| Tercer proyecto: Coches FX..... | 63 |
| Aregar imágenes a un proyecto..... | 65 |
| Capturando un evento de teclado..... | 67 |
| Desplazando un objeto en un Pane..... | 68 |
| Creando ventana emergente..... | 68 |
| Cuarto Proyecto: Ponderaciones..... | 69 |
| Creando cuadro de diálogo personalizado..... | 72 |
| Quinto Proyecto: Ahorcado..... | 74 |
| Sexto Proyecto: Elecciones Dhont, Proporcional..... | 77 |
| Sexto Proyecto: Snake..... | 79 |
| Séptimo proyecto: Hundir la flota..... | 81 |
| Anexo empaquetar una aplicación gráfica..... | 84 |
| Anexo: Crear un jar ejecutable en Netbeans 13 JavaFX..... | 85 |
| Anexo Nomenclatura de controles Swing..... | 98 |

Juan Carlos Pérez Rodríguez

Introducción

Imaginemos un programa que se encarga de borrar todos los ficheros con más de 10 años de antigüedad, previamente haciendo una copia de seguridad que sube la copia a un servicio en la nube.

Ese programa no necesita prácticamente interacción con el usuario, y si somos estrictos en el enunciado realmente no necesita ninguna.

El caso que se acaba de describir nos sirve para ver que la interacción con el usuario es una interfaz. Un programa puede funcionar perfectamente sin necesidad de tal interfaz.

Antes de que se hicieran comunes sistemas operativos como Windows, etc los programas interactuaban con el usuario mediante modo consola, esto es, mediante órdenes al ordenador con comandos por teclado. Hoy en día eso ya no es así y el uso de interfaces gráficas de usuario se han convertido en un componente importante en programación.

Definición: La interfaz gráfica de usuario, conocida también como GUI (del inglés graphical user interface), es un programa informático que actúa de interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. Su principal uso, consiste en proporcionar un entorno visual sencillo para permitir la comunicación con el sistema operativo de una máquina.

Java proporciona dos bibliotecas de clases para crear interfaces gráficas de usuario: AWT y Swing. También hablaremos de Java FX

AWT y Swing

En el paquete estándar de Java, contamos con dos opciones para crear interfaces gráficas de usuario:

1. AWT -Abstract Window Toolkit.
2. Swing.

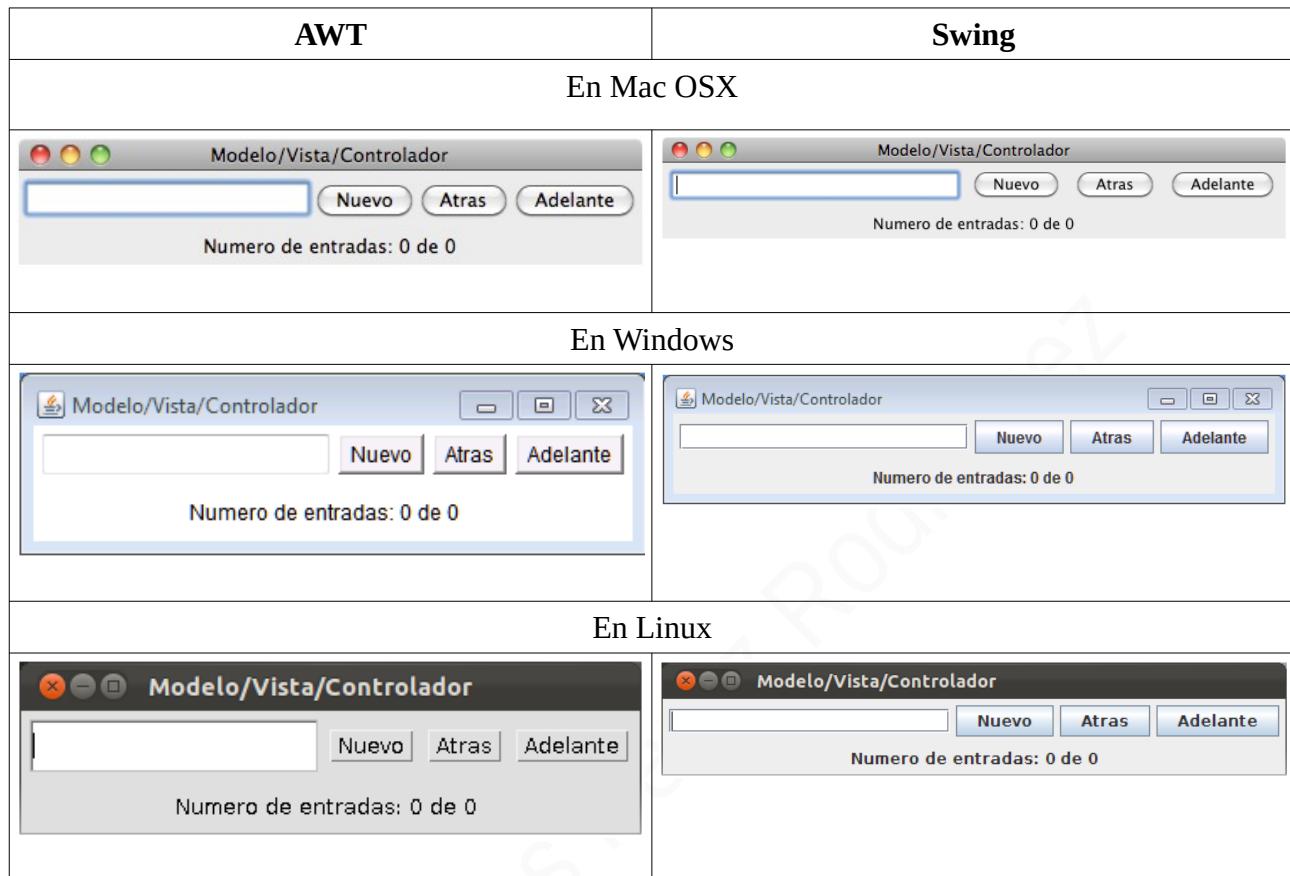
AWT es una biblioteca pesada -heavy weight-, mientras que Swing es una biblioteca ligera - light weight- de componentes.

La idea de pesada o ligera, en este caso, está relacionada con la dependencia de Java con el sistema operativo, para visualizar y gestionar los elementos de la interface gráfica de usuario.

En el caso de AWT, la creación, visualización y gestión de los elementos gráficos depende del SO. Es el propio SO quien dibuja y gestiona la interacción sobre los elementos.

En el caso de Swing, es Java quien visualiza y gestiona la interacción del usuario sobre los elementos de la interface gráfica.

Veamos una comparativa:



Se puede observar que mediante AWT la apariencia de la aplicación recáe en el sistema operativo. Mientras que con Swing hay mayor parecido.

Por cada componente AWT existe un componente Swing equivalente, cuyo nombre empieza por J, que permite más funcionalidad siendo menos pesado. Por ejemplo, en ambas bibliotecas tenemos un clase para crear ventana Frame en el caso de AWT y JFrame en el caso de Swing. **Fíjate que si es una clase del paquete Swing su nombre empieza por J.**

Existen otras clases que son el paquete AWT pero se utilizan en Swing, por ejemplo, los eventos y escuchadores. Como no tienen representación gráfica, en Swing se reutilizan los de AWT.

Swing

Cuando se vio que era necesario mejorar las características que ofrecía AWT, distintas empresas empezaron a sacar sus controles propios para mejorar algunas de las características de AWT. Así, Netscape sacó una librería de clases llamada Internet Foundation Classes para usar con Java, y eso obligó a Sun (todavía no adquirida por Oracle) a reaccionar para adaptar el lenguaje a las nuevas necesidades.

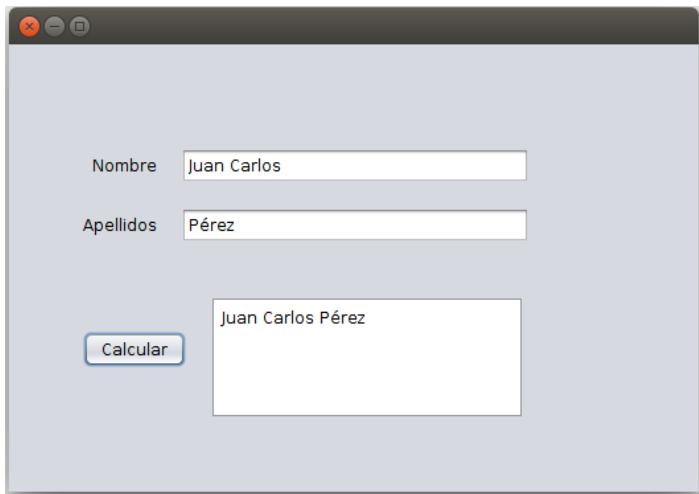
Se desarrolló en colaboración con Netscape todo el conjunto de componentes Swing que se añadieron a la JFC.

Swing es una librería de Java para la generación del GUI en aplicaciones.

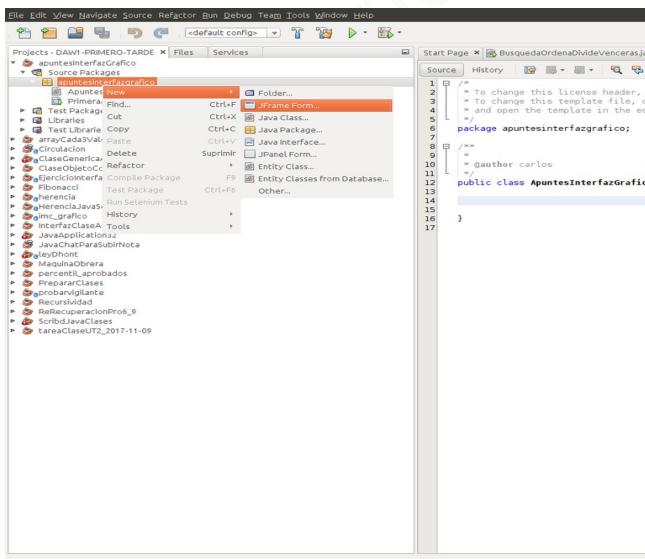
Swing se apoya sobre AWT y añade JComponents. La arquitectura de los componentes de Swing facilita la personalización de apariencia y comportamiento, si lo comparamos con los componentes AWT.

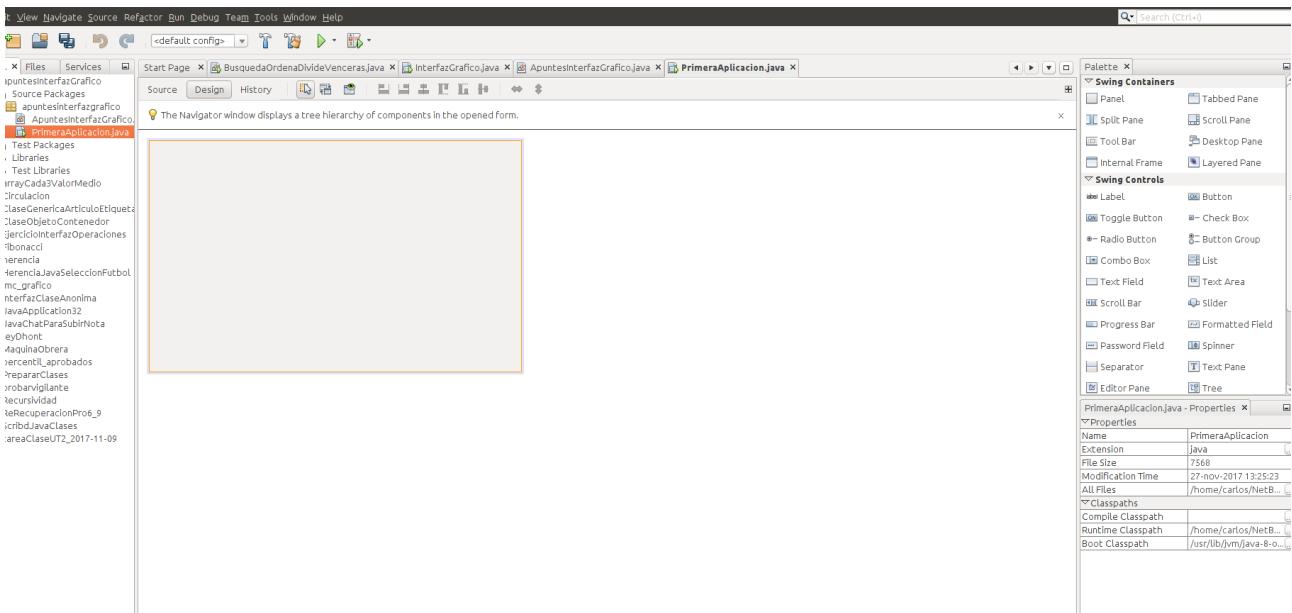
Uso del IDE para crear Interfaces gráficos

Como una primera aproximación vamos a realizar una aplicación que se introduzca nombre y apellidos para obtener la composición de ambos:



Primero haremos como hasta ahora: Crear un proyecto netbeans. Pero ahora haremos botón derecho sobre el paquete creado por el IDE → New → JFrameForm





Arrastrar controles Swing a nuestro Jframe: dos label donde escribiremos Nombre, Apellidos. Dos jTextField para introducir esos datos. Un JButton calcular y un JTextArea

Una vez arrastrados y colocados donde así estimemos debemos ponerle un identificador significativo a los controles que vamos a manejar desde código: txtNombre, txtApellidos, btnCalcular, txaResultado respectivamente. Se ha elegido tomar las tres primeras consonantes más significativas de cada control:

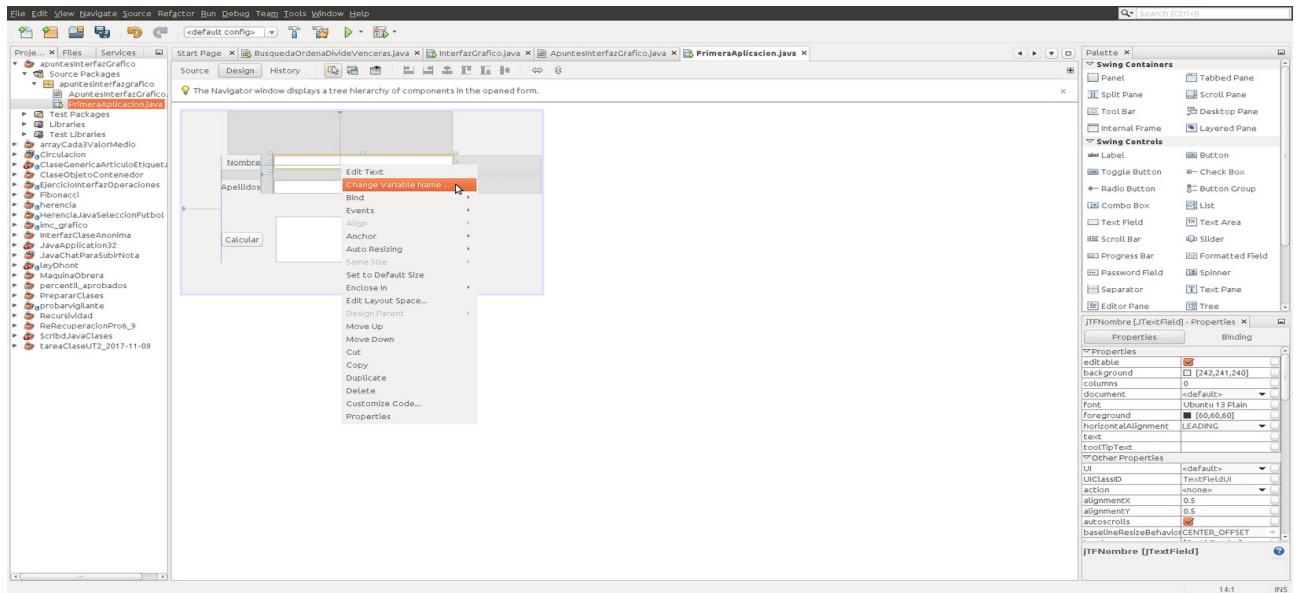
txt → jTextField

btn → JButton

txa → JTextArea

Observar que no se incluye como significativa la letra j al ser todos los controles Swing con esa letra al comienzo

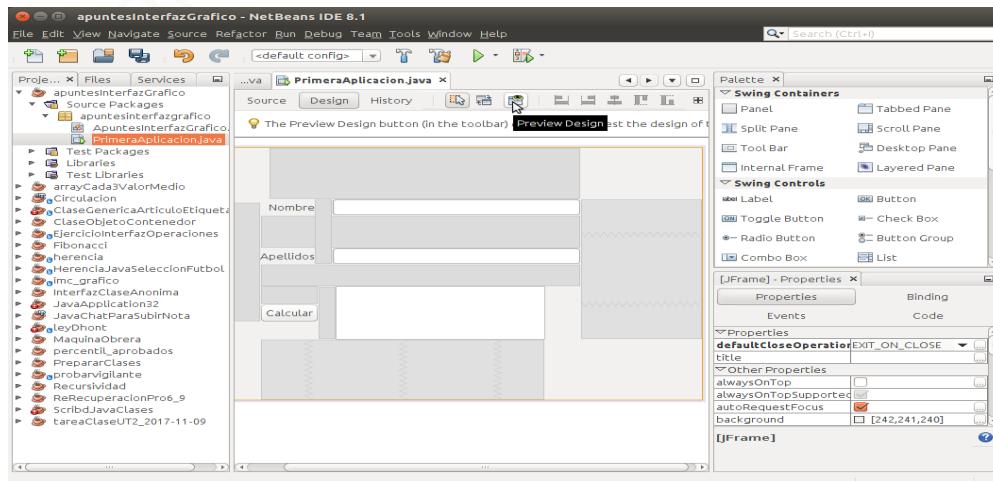
Para establecer esos nombres posiblemente el procedimiento más cómodo es: botón derecho sobre el control → Change Variable Name



Observar que en el mismo menú aparece la opción de editar el texto que muestra el control

En la parte derecha de la imagen arriba expuesta se observa la ventana Properties que nos permite ajustar colores y demás elementos de presentación del control

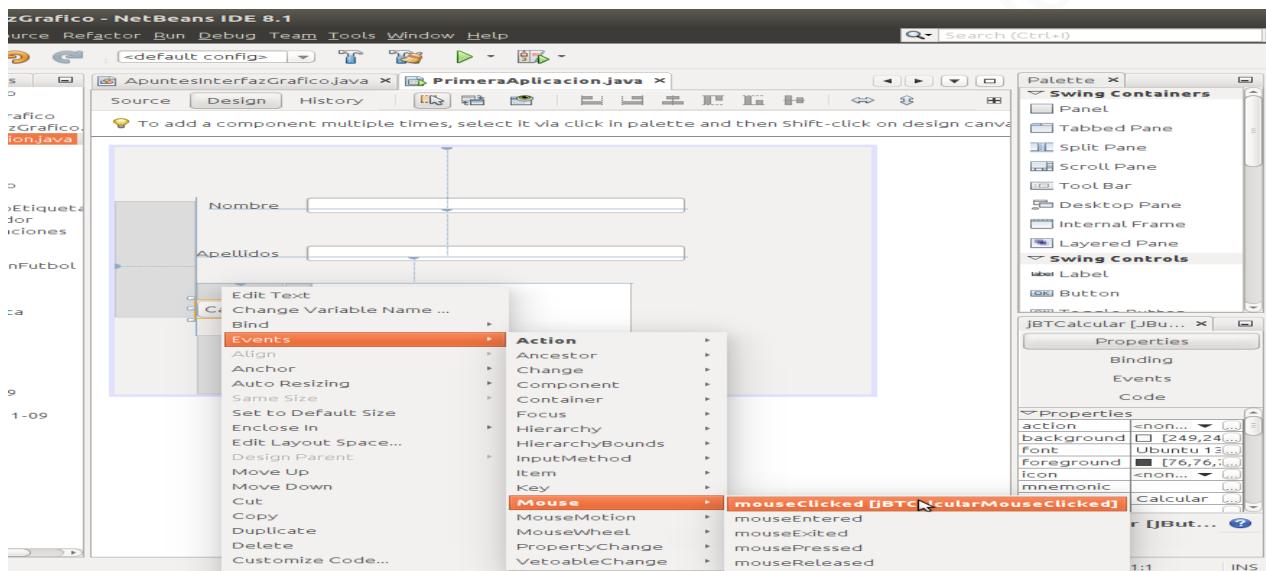
Siempre podemos observar como va quedando el diseño de nuestra interfaz pulsando sobre “Preview Design” Que es el ícono con forma de ojo:



Al botón debemos asignarle un evento para que cuando hagamos click nos ejecute la acción que queremos

La forma más rápida es pulsando doble click sobre el propio botón que nos llevará al método que se ha generado automáticamente para cualquier opción. Sin embargo es mejor especificar exactamente el evento que queremos establecer. Para ello:

botón derecho sobre el control → Events → Mouse → mouseClicked



Nos habrá llevado al código al nombre del método que habrá creado para gestionar el evento de click. Observar que hay texto en color gris. Esa parte es toda generada por el IDE automáticamente respecto a nuestro diseño del interfaz y no debemos modificarlo

En el punto donde nos ha ubicado el IDE escribiremos:

```
private void btnCalcularMouseClicked(java.awt.event.MouseEvent evt) {  
    txaResultado.setText(txtNombre.getText() + " " + txtApellidos.getText());  
}
```

Observar que utilizamos el método: setText del textarea resultado para establecer el texto de unir el nombre junto con el apellido

Hacemos uso de los métodos getText() de los controles textfield para nombre y apellidos para obtener una String con lo que haya introducido el usuario en esos controles de texto. Luego simplemente lo concatenamos

● **Práctica 1:** Reproducir lo que se ha explicado en el ejemplo y hazlo en el IDE luego ejecuta la aplicación e introduce tu nombre y apellidos y pulsa el botón
Toma captura de pantalla de la ventana de ejecución

Contenedores

Qué componentes se usan para contener a los demás?

En Swing esa función la desempeñan un grupo de componentes llamados **contenedores** Swing.

Existen dos tipos de elementos contenedores:

- **Contenedores de alto nivel** o "peso pesado".
 - Marcos: `JFrame` y `JDialog` para aplicaciones
 - `JApplet`, para [applets](#).
- **Contenedores de bajo nivel** o "peso ligero". Son los paneles: `JRootPane` y `JPanel`.

Cualquier aplicación, con interfaz gráfico de usuario típica, comienza con la apertura de una ventana principal, que suele contener la barra de título, los botones de minimizar, maximizar/restaurar y cerrar, y unos bordes que delimitan su tamaño.

Esa ventana constituye un marco dentro del cual se van colocando el resto de componentes que necesita el programador: menú, barras de herramientas, barra de estado, botones, casillas de verificación, cuadros de texto, etc.

Esa ventana principal o marco sería el contenedor de alto nivel de la aplicación.

Toda aplicación de interfaz gráfica de usuario Java tiene, al menos, un contenedor de alto nivel.

Los contenedores de alto nivel extienden directamente a una clase similar de AWT, es decir, `JFrame` extiende de `Frame`. Es decir, realmente necesitan crear una ventana del sistema operativo independiente para cada uno de ellos.

Los demás componentes de la aplicación no tienen su propia ventana del sistema operativo, sino que se dibujan en su objeto contenedor.

En los ejemplos anteriores del tema, hemos visto que podemos añadir un `JFrame` desde el diseñador de NetBeans, o bien escribiéndolo directamente por código. De igual forma para los componentes que añadamos sobre el.

Ahora vamos a fijarnos en el código generado (en gris)

```
jBTCalcular.addMouseListener(new java.awt.event.MouseAdapter() {  
    public void mouseClicked(java.awt.event.MouseEvent evt) {  
        jBTCalcularMouseClicked(evt);  
    }  
});
```

Se puede observar que se está recurriendo a una clase anónima para implementar un Listener en este caso a los eventos del ratón. Vemos que nos generar un método llamado en el caso del ejemplo: `jBTCalcularMouseClicked(evt)` que será el que nosotros pongamos código

Vamos a ver pues los eventos:

Eventos

¿Qué es un **evento**?

Es todo hecho que ocurre mientras se ejecuta la aplicación. Normalmente, llamamos evento a cualquier interacción que realiza el usuario con la aplicación, como puede ser:

- pulsar un botón con el ratón;
- hacer doble clic;
- pulsar y arrastrar;
- pulsar una combinación de teclas en el teclado;
- pasar el ratón por encima de un componente;
- salir el puntero de ratón de un componente;
- abrir una ventana;
- etc.

¿Qué es la **programación guiada por eventos**?

Imagina la ventana de cualquier aplicación, por ejemplo la de un procesador de textos. En esa ventana aparecen multitud de elementos gráficos interactivos, de forma que no es posible que el programador haya previsto todas las posibles entradas que se pueden producir por parte del usuario en cada momento.

Con el control de flujo de programa de la **programación imperativa**, el programador tendría que estar continuamente leyendo las entradas (de teclado, o ratón, etc) y comprobar para cada entrada o interacción producida por el usuario, de cual se trata de entre todas las posibles, usando estructuras de flujo condicional (**if-then-else**, **switch**) para ejecutar el código conveniente en cada caso. Si piensas que para cada opción del menú, para cada botón o etiqueta, para cada lista desplegable, y por tanto para cada componente de la ventana, incluyendo la propia ventana, habría que comprobar todos y cada uno de los eventos posibles, nos damos cuenta de que las posibilidades son casi infinitas, y desde luego impredecibles. Por tanto, de ese modo es imposible solucionar el problema.

Para abordar el problema de tratar correctamente las interacciones del usuario con la interfaz gráfica de la aplicación hay que cambiar de estrategia, y la **programación guiada por eventos es una buena solución**, veamos cómo funciona el modelo de gestión de eventos.

Hoy en día, la mayoría de sistemas operativos utilizan interfaces gráficas de usuario. Este tipo de sistemas operativos están **continuamente monitorizando el entorno para capturar y tratar los eventos** que se producen.

El sistema operativo informa de estos eventos a los programas que se están ejecutando y entonces cada programa decide, según lo que se haya programado, qué hace para dar respuesta a esos eventos.

Cada vez que el usuario realiza una determinada acción sobre una aplicación que estamos programando en Java, un clic sobre el ratón, presionar una tecla, etc., se produce un evento que el sistema operativo transmite a Java.

Java crea un objeto de una determinada clase de evento, y este evento se transmite a un determinado método para que lo gestione.

El **modelo de eventos de Java está basado en delegación**, es decir, la responsabilidad de gestionar un evento que ocurre en un objeto fuente la tiene otro objeto **oyente**.

Las **fuentes de eventos** (event sources) son objetos que detectan eventos y notifican a los receptores que se han producido dichos eventos. Ejemplos de fuentes:

- Botón sobre el que se pulsa o pincha con el ratón.
- Campo de texto que pierde el foco.
- Campo de texto sobre el que se presiona una tecla.
- Ventana que se cierra.
- Etc.

En el apartado anterior de creación de interfaces con ayuda de los asistentes del IDE, vimos lo fácil que es realizar este tipo de programación, ya que el IDE hace muchas cosas, genera código automáticamente por nosotros.

Modelo Vista Controlador

El patrón MVC

- **Un modelo:** Es la capa donde se trabaja con los datos, por tanto contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos los tendremos habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes selects, updates, inserts, etc.
- **Varias vistas:** Las vistas, como su nombre nos hace entender, contienen el código de nuestra aplicación que va a producir la visualización de las interfaces de usuario, o sea, el código que nos permitirá renderizar los estados de nuestra aplicación en HTML. En las vistas nada más tenemos los códigos HTML y PHP que nos permite mostrar la salida. En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generará la salida, tal como nuestra aplicación requiera.
- **Varios controladores:** Contiene el código necesario para responder a las acciones que se solicitan en la aplicación, como visualizar un elemento, realizar una compra, una búsqueda de información, etc. En realidad es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación.
- Las vistas y los controladores suelen estar muy relacionados
 - Los controladores tratan los eventos que se producen en la interfaz gráfica (vista)

Esta separación de aspectos de una aplicación da mucha flexibilidad al desarrollador

Flujo de control:

1. El usuario realiza una acción en la interfaz
2. El controlador trata el evento de entrada
3. El controlador notifica al modelo la acción del usuario, lo que puede implicar un cambio del estado del modelo
4. Se genera una nueva vista. La vista toma los datos del modelo
5. La interfaz de usuario espera otra interacción del usuario, que comenzará otro nuevo ciclo

MVC en Java Swing

- Modelo:
 - El modelo lo realiza el desarrollador
- Vista:
 - Conjunto de objetos de clases que heredan de `java.awt.Component`
- Controlador:
 - El controlador es el thread de tratamiento de eventos, que captura y propaga los eventos a la vista y al modelo
 - Clases de tratamiento de los eventos (a veces como clases anónimas) que implementan interfaces de tipo `EventListener` (`ActionListener`, `MouseListener`, `WindowListener`, etc.)

Vamos a realizar una aplicación y de paso ver algunos controles mediante Swing.

La pantalla de ejemplo:



En la aplicación realizada, el código auto-generado (en gris) sería lo más parecido a hablar de la vista.

El código que establecemos en los métodos que gestionan los eventos serían el controlador (modifican la vista introduciendo la información del usuario y del modelo. Se comunican con el modelo para obtener los resultados)

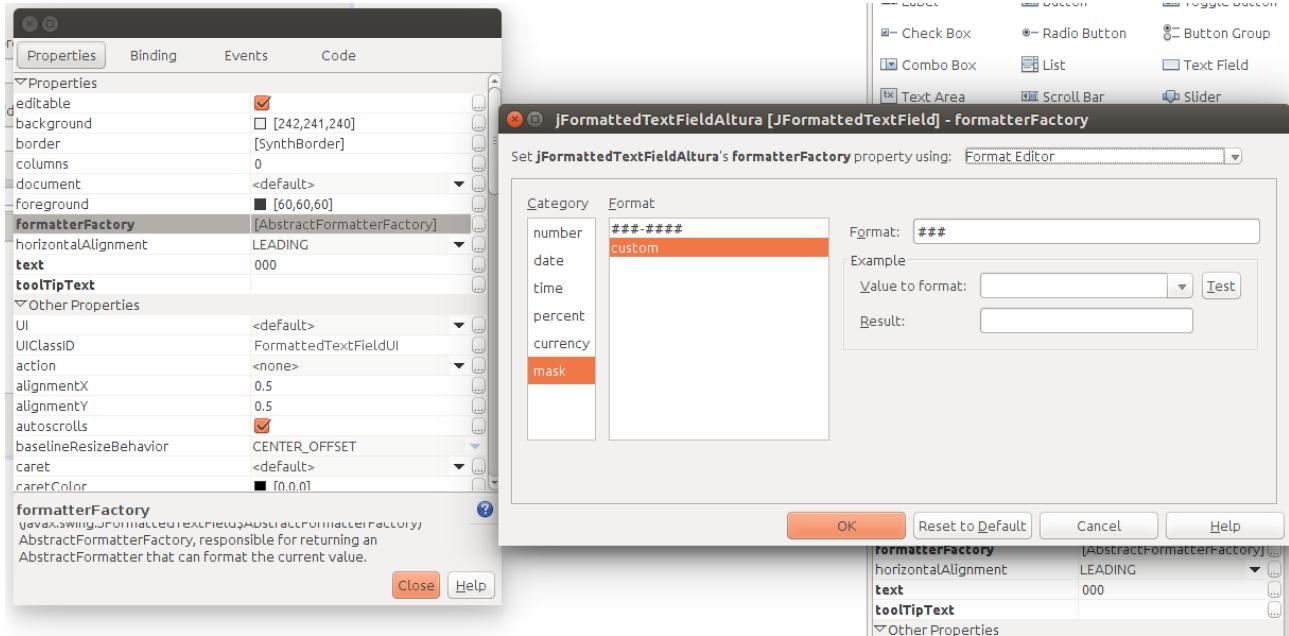
El modelo estaría compuesto por los ficheros Persona, Hombre, Mujer.

Para poner el nombre en la ventana vamos a title en el JFrame y ponemos: "Cálculo Peso Ideal"

Para conseguir que cuando marquemos una opción Hombre quite la selección en Mujer y viceversa incorporamos un ButtonGroup. Luego agregamos los dos RadioButton a ese buttongroup

Hay 3 jFormattedTextField uno para edad otro para altura y otro para peso

Veamos un ejemplo de como poner la máscara:



En las propiedades del objeto vamos al apartado FormattedFactory y pulsamos en el símbolo tres puntos. Allí seleccionamos mask -->custom

###

hace referencia a que deben ponerse 3 dígitos de otra forma no acepta ninguna otra entradas

Para establecer así como nosotros queramos el String resultante para determinar una cantidad específica de decimales y demás:

```
StringBuilder sbuf = new StringBuilder();
Formatter fmt = new Formatter(sbuf);
fmt.format("PI = %.3f%n", Math.PI);
System.out.println(sbuf);
```

Con lo anterior se consigue especificar 3 decimales al mostrar PI mediante una String



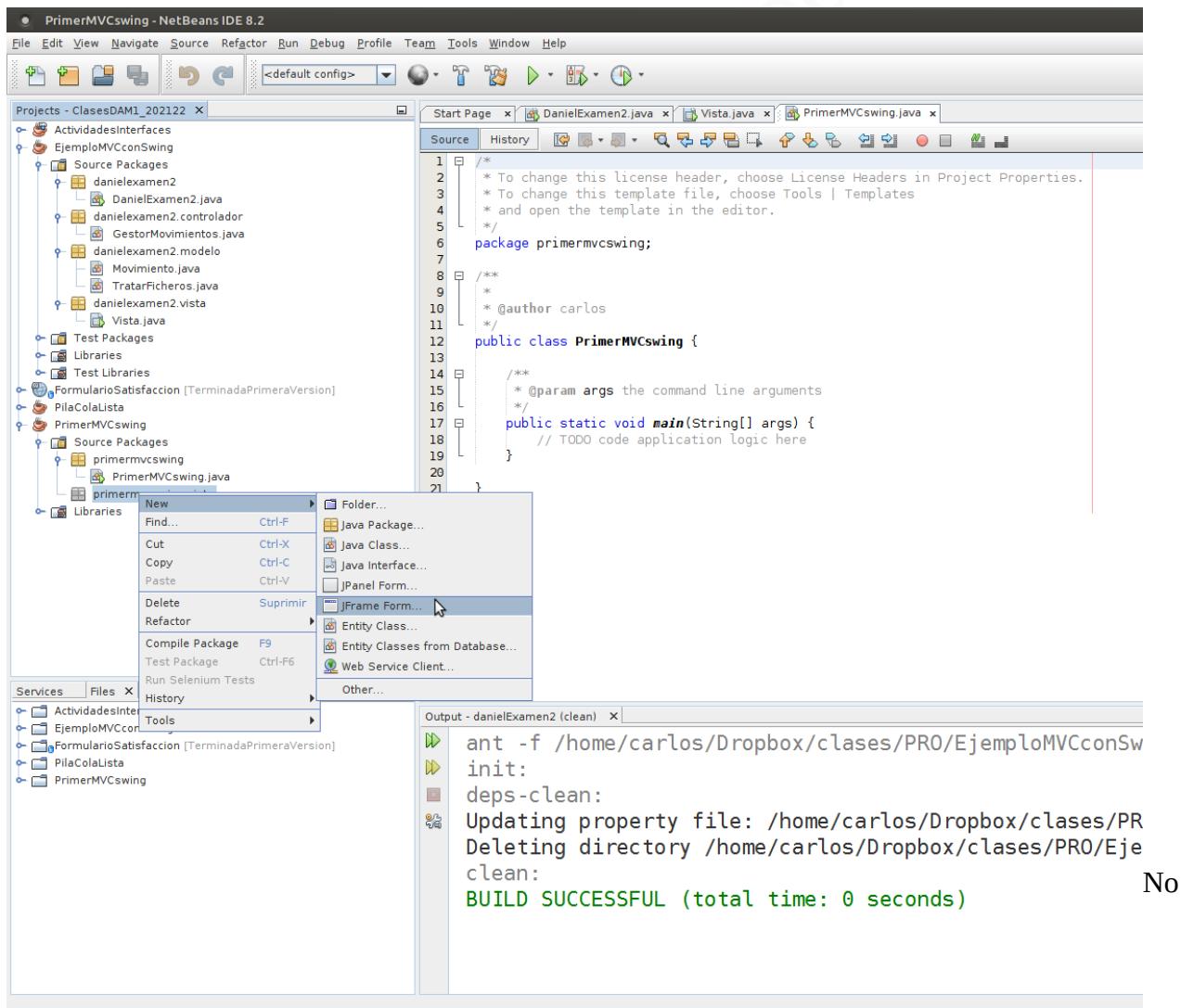
Práctica 2: Realizar la aplicación descrita mediante Netbeans con Swing

MVC con Swing

Creamos 3 paquetes:

- modelo
- vista
- controlador

Ahora en el paquete vista incorporamos un nuevo JFrame (el jframe lo vamos a llamar: Vista):



Sabemos que estamos en un modelo orientado a eventos. Podemos hacer que una clase (la que nosotros queramos) implemente el interfaz ActionListener, de esa forma cuando se desencadene un evento se ejecutará el método actionPerformed() (ese va a ser el punto central de nuestro controlador)

Vamos a comprobar que cada vez que se genere un evento se lanza el método y obtenemos el objeto que lo lanzó (evento.getSource())

```
public class Controlador implements ActionListener{  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        System.out.println(  
            e.getSource()  
        );  
  
    }  
  
    Vista vista;  
    public Controlador( Vista vista){  
        this.vista = vista;  
    }  
  
    public void iniciar(){  
        this.vista.setVisible(true);  
    }  
}
```

Para que esta clase Controlador gestione la vista tenemos que hacer varias cosas. Primero tenemos que saber desde el controlador de la vista. Por eso tenemos un constructor que recibe la vista y la ponemos como atributo: **this.vista = vista;**

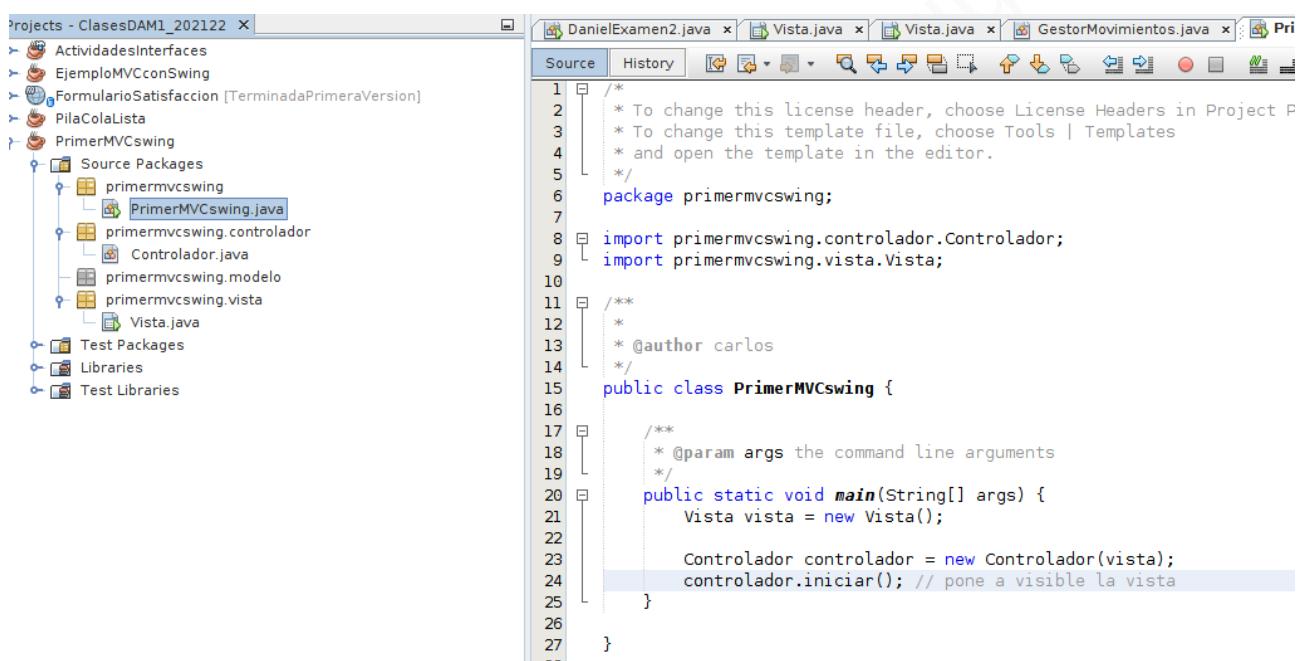
Como dijimos antes hay que implementar ActionListener lo cual nos general el método actionPerformed. Como dijimos ese será el punto central de recepción de información del controlador

```
@Override  
public void actionPerformed(ActionEvent e) {  
    System.out.println(  
        e.getSource()  
    );  
}
```

Finalmente se observa que generamos un método para hacer visible la vista. Lo hemos llamado: iniciar()

```
public void iniciar(){
    this.vista.setVisible(true);
}
```

Ese método se debe ejecutar en el main de nuestro proyecto. Vamos a ver ese código:
(método main en la clase inicial de nuestro proyecto)



The screenshot shows an IDE interface with a project tree on the left and a code editor on the right. The project tree contains several Java packages and files. The code editor displays the main method of the `PrimerMVCswing` class:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primermvcswing;

import primermvcswing.controlador.Controlador;
import primermvcswing.vista.Vista;

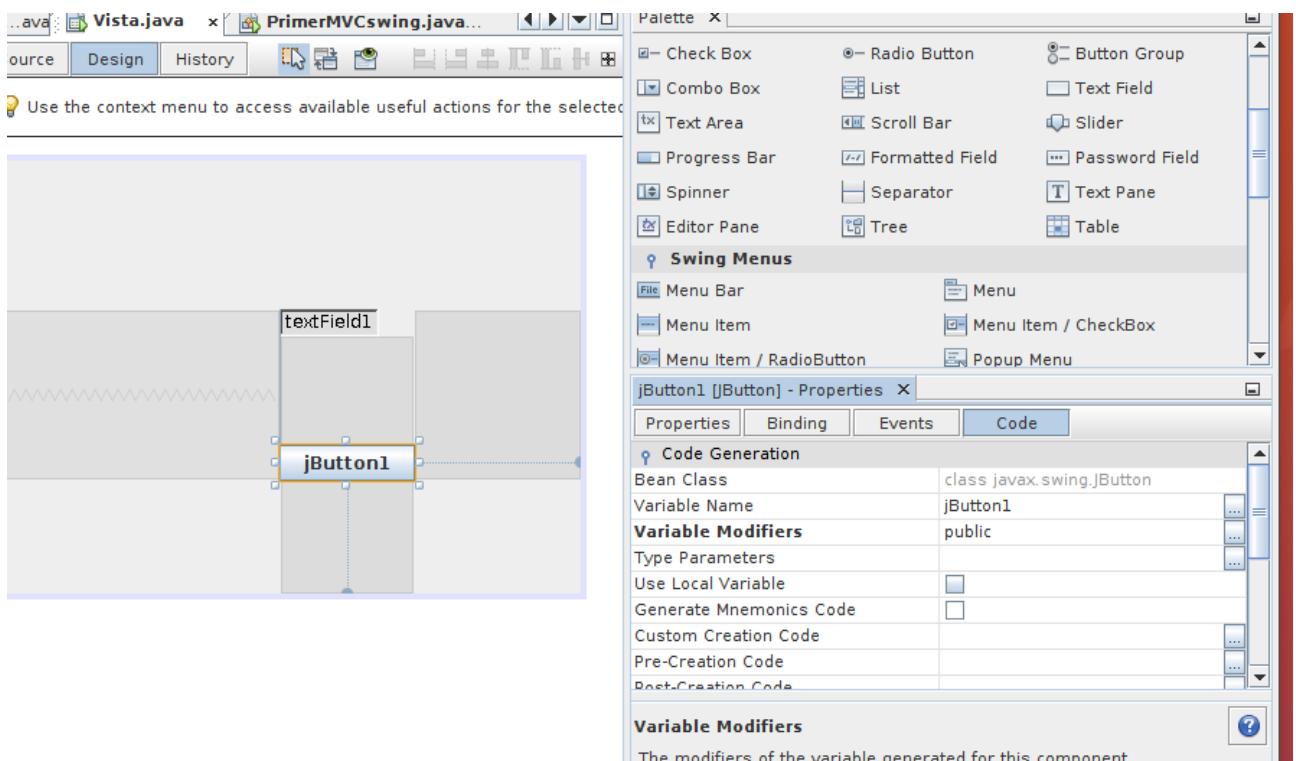
/**
 * @author carlos
 */
public class PrimerMVCswing {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Vista vista = new Vista();

        Controlador controlador = new Controlador(vista);
        controlador.iniciar(); // pone a visible la vista
    }
}
```

Vemos que se genera primero el objeto Vista, luego creamos un nuevo Controlador que le hemos pasado la Vista y finalmente hacemos visible la vista gracias al método: iniciar()

Con lo anterior ya tenemos casi toda la separación. Ahora lo que falta es agregar los listener a cada objeto que queramos que lo tenga. Supongamos que hemos creado un botón en nuestra vista (por defecto el IDE le pone el nombre: `jButton1`) Vamos asociarle un listener. Para ello primero debemos permitir que sea accesible desde nuestro controlador:



Observar que desde la vista diseño del jframe tenemos seleccionado el jButton y en la pestaña de Code hemos puesto en: Variable Modifiers: public

De esa forma nos genera el código (en gris en la vista Source) del botón como public y podemos acceder al botón desde el controlador

Ahora hagamos la modificación en el controlador para que ponga el listener:

```
public class Controlador implements ActionListener{  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        System.out.println(  
            e.getSource()  
        );  
  
    }  
  
    Vista vista;  
    public Controlador( Vista vista){  
        this.vista = vista;  
        this.vista.JButton1.addActionListener(this);  
    }  
  
    public void iniciar(){  
        this.vista.setVisible(true);  
    }  
}
```

Si ahora ejecutamos el proyecto vemos que nos muestra en consola el botón cuando pulsamos con el ratón

```
@Override  
public void actionPerformed(ActionEvent e) {  
  
    if( e.getSource() == this.vista.JButton1){  
        botonPulsado();  
    }  
}
```

En el código anterior vemos que cuando el evento lo haya disparado el botón JButton1 entonces se ejecutará el método: botonPulsado() de esa forma hemos conseguido separar la vista del controlador

No vamos a detenernos más en Swing Vamos a trabajar en JavaFX que es más moderno y nos permite ver todavía mejor la separación en el patrón MVC

JavaFX

JavaFX proporciona a los desarrolladores de Java una nueva plataforma gráfica. JavaFX 2.0 se publicó en octubre del 2011 con la intención de reemplazar a Swing en la creación de nuevos interfaces gráficos de usuario (IGU).

Para realizar una aplicación en JavaFX utilizaremos el paquete netbeans que nos da oracle para descargar ya con todo incluido

Te podría interesar mantener los siguientes enlaces:

- [Java 8 API](#) - Documentación (JavaDoc) de las clases estándar de Java
- [JavaFX 8 API](#) - Documentación de las clases JavaFX
- [ControlsFX API](#) - Documentación para el proyecto [ControlsFX](#), el cual ofrece controles JavaFX adicionales
- [Oracle's JavaFX Tutorials](#) - Tutoriales oficiales de Oracle sobre JavaFX

JavaFX es un conjunto de gráficos y paquetes de comunicación que permite a los desarrolladores para diseñar, crear, probar, depurar y desplegar aplicaciones cliente enriquecidas que operan constantemente a través de diversas plataformas

La biblioteca de JavaFX está escrita como una API de Java, las aplicaciones JavaFX puede hacer referencia a APIs de código de cualquier biblioteca Java. Por ejemplo, las aplicaciones JavaFX pueden utilizar las bibliotecas de API de Java para acceder a las capacidades del sistema nativas y conectarse a aplicaciones de middleware basadas en servidor.

La apariencia de las aplicaciones JavaFX se pueden personalizar. Las Hojas de Estilo en Cascada (CSS) separan la apariencia y estilo de la lógica de la aplicación para que los desarrolladores puedan concentrarse en el código. Los diseñadores gráficos pueden personalizar fácilmente el aspecto y el estilo de la aplicación através de CSS.

Se uede desarrollar los aspectos de la presentación de la interfaz de usuario en el lenguaje de scripting FXML y usar el código de Java para la aplicación lógica.

Si se prefiere diseñar interfaces de usuario sin necesidad de escribir código, entonces, se puede utilizar JavaFX Scene Builder. Al diseñar la interfaz de usuario con javaFX Scene Builder el crea código de marcado FXML que pueden ser portado a un entorno de desarrollo integrado (IDE) de forma que los desarrolladores pueden añadir la lógica de negocio.

Así pues por un lado tendremos:

- Un fichero fxml que representará la vista
- Un posible fichero CSS para los estilos de esa vista
- Ficheros java para el modelo y para la vista-controlador

Pero ¿ qué es fxml ?

FXML es un lenguaje de marcado declarativo basado en XML para la construcción de una interfaz de usuario de aplicaciones JavaFX. Un diseñador puede codificar en FXML o utilizar JavaFX Scene Builder para diseñar de forma interactiva la interfaz gráfica de usuario (GUI). Scene Builder genera marcado FXML que pueden ser portado a un IDE para que un desarrollador pueda añadir la lógica de negocio.

Hojas de estilo en cascada CSS

Ofrece la posibilidad de aplicar un estilo personalizado a la interfaz de usuario de una aplicación JavaFX sin cambiar ningún de código fuente de la aplicación. CSS se puede aplicar a cualquier nodo en el gráfico de la escena JavaFX y se aplica a los nodos de forma asincrónica.

JavaFX también puede aplicar estilos CSS fácilmente asignados a la escena en tiempo de ejecución, haciendo que una aplicación para cambiar dinámicamente.

Controles de interfaz de usuario(UI Controls)

Veamos algunos ejemplos de controles del interfaz de usuario:



La mejor forma de entender seguramente sea mediante ejemplo y seguiremos ese procedimiento. De cualquier forma, si se prefiere tener acceso a toda la documentación, se puede ver en los enlaces:

https://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm

<https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>

y un interesante tutorial (en inglés) que ya no es de oracle:

<http://tutorials.jenkov.com/javafx/index.html>

Primer proyecto de Ejemplo de JavaFX

Vamos a precisar que el IDE tenga soporte para JavaFX, posiblemente una opción rápida sea instalar el paquete unificado que da Oracle de jdk y netbeans Ya trae instalado y configurado JavaFX: <http://www.oracle.com/technetwork/java/javase/downloads/jdk-netbeans-jsp-142931.html>

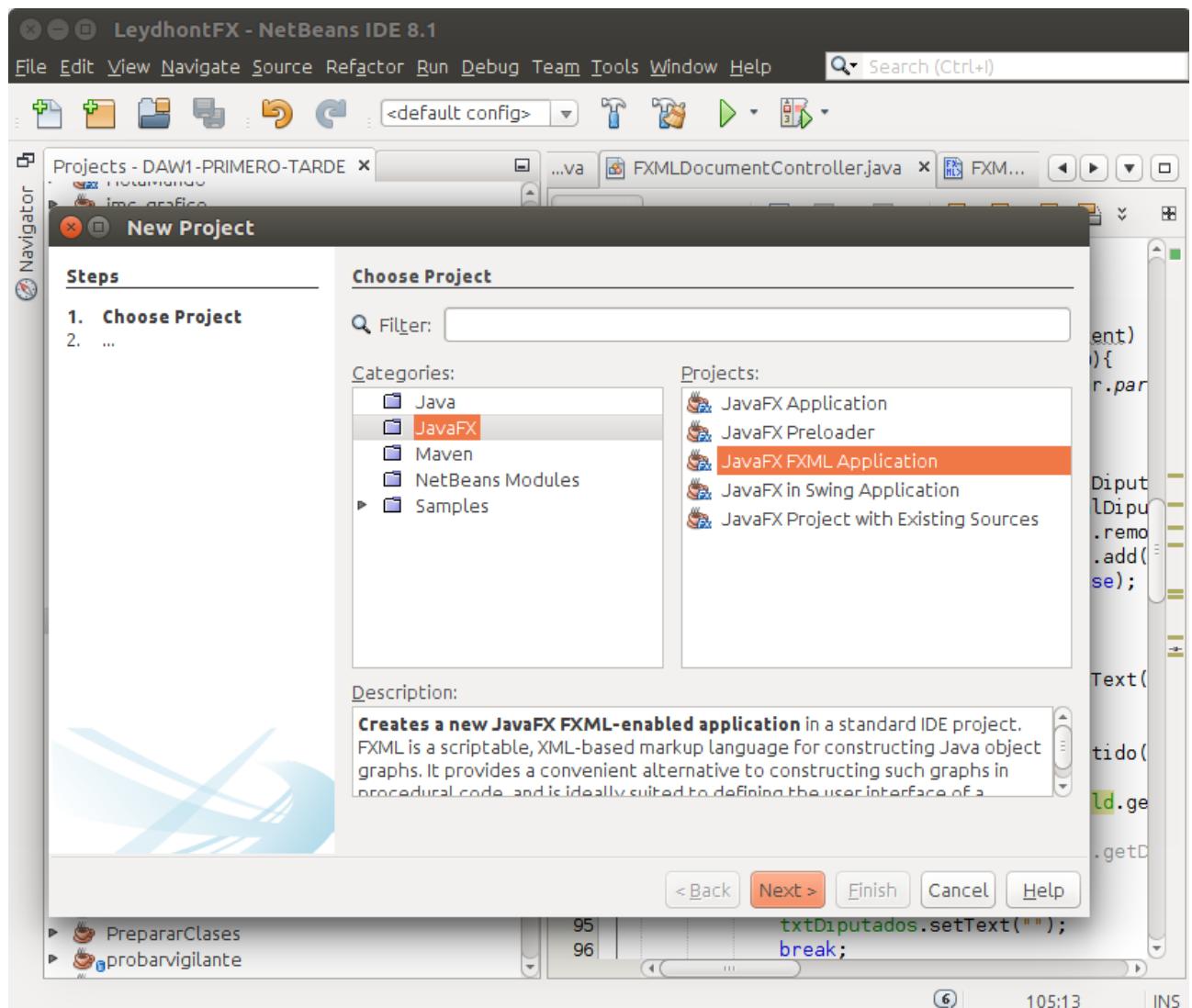
Adicionalmente se precisará el Scene Builder. En este caso se propone el de Gluon:

<http://gluonhq.com/products/scene-builder/#download>

Se propone elegir la opción de “Executable Jar” por ser independiente de la plataforma (windows-linux-mac) y no ser necesarios permisos de administrador

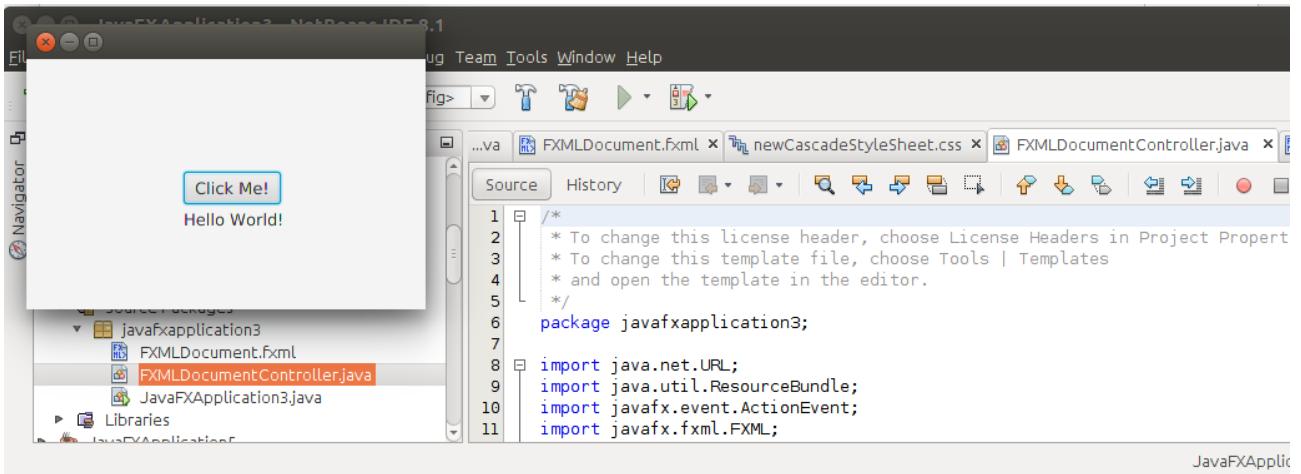
Una vez instalado todo el procedimiento es desde Netbeans:

File → New Project → JavaFX → JavaFX FXML Application



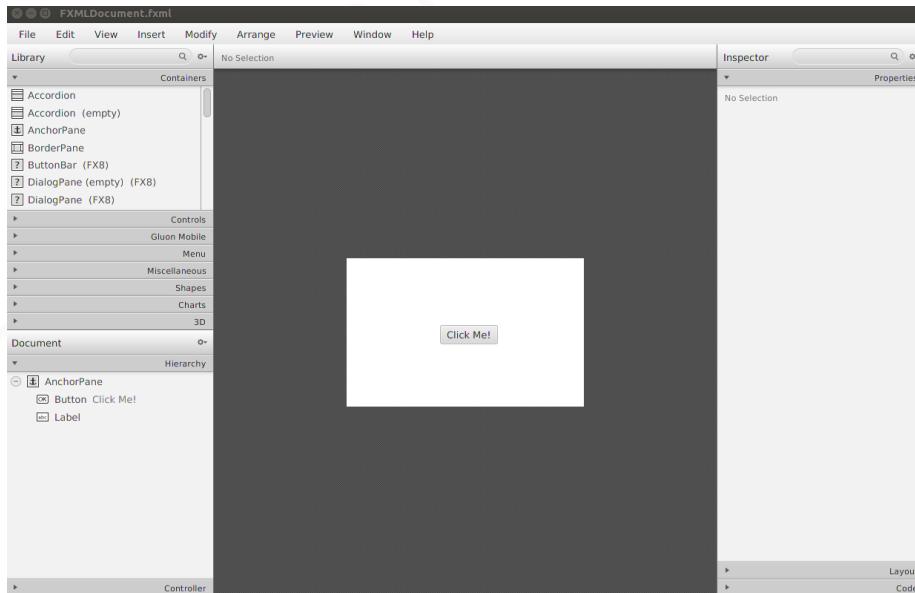
Al finalizar la creación del proyecto ya nos ha creado una miniaPLICACIÓN funcional. Compuesto por tres ficheros: el fichero con el main, el fichero para la vista: (extension .fxml) y el fichero para el controlador (FXMLDocumentController.java)

Si ejecutamos el proyecto nos muestra una ventana con un botón “click me”



Paramos la ejecución.

Ahora, para la personalización de la vista pulsamos doble click sobre el fichero con extensión: .fxml y nos abrirá el Scene Builder (también: botón derecho->open)



Observamos que hay un AnchorPane, un button y un label

Desde aquí con una interfaz gráfica podemos eliminar, agregar, editar todos los contenedores y controles que queramos en nuestra vista

Si en lugar de haber elegido open hubiéramos hecho: botón derecho sobre el fichero .fxml → Edit

nos habría abierto el fichero .fxml directamente, sin editores gráficos:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320" xmlns:fx="http://javafx.com/fxml/1" fx:controller="javafxapplication3.FXMLDocumentController">
    <children>
        <Button layoutX="126" layoutY="90" text="Click Me!" onAction="#handleButtonAction" fx:id="button" />
        <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69" fx:id="label" />
    </children>
</AnchorPane>
```

Como se puede observar, no difiere de otros XML que pudiéramos haber visto anteriormente.

Interesante observar en el código xml es el atributo: **onAction**

```
<Button layoutX="126" layoutY="90" text="Click Me!" onAction="#handleButtonAction" fx:id="button" />
```

De tal forma se está estableciendo en la vista que el método: handleButtonAction() del FXMLDocumentController.java se encarga de manejar el evento del botón. Eso significa que podemos desarrollar nuestro controlador por un lado y nuestra vista por otro. Y simplemente agregando el atributo a la línea xml de la vista que nos corresponda ya quedarían enlazados. Independizando mucho la vista del controlador. Observar que para referenciarlo utilizamos el símbolo: “#” en el nombre del método: **onAction="#handleButtonAction"**

También es interesante que observemos como se le dice a la vista cuál es el fichero controlador que le corresponde:

```
<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320" xmlns:fx="http://javafx.com/fxml/1" fx:controller="javafxapplication3.FXMLDocumentController">
```

Mediante el atributo: **fx:controller** se establece el nombre del fichero .java que será el controlador para esta vista

Veamos ahora el fichero controlador que nos ha generado:

```
import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

/**
 *
 * @author carlos
 */
public class FXMLDocumentController implements Initializable {

    @FXML
    private Label label;

    @FXML
    private void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
        label.setText("Hello World!");
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        // TODO
    }
}
```

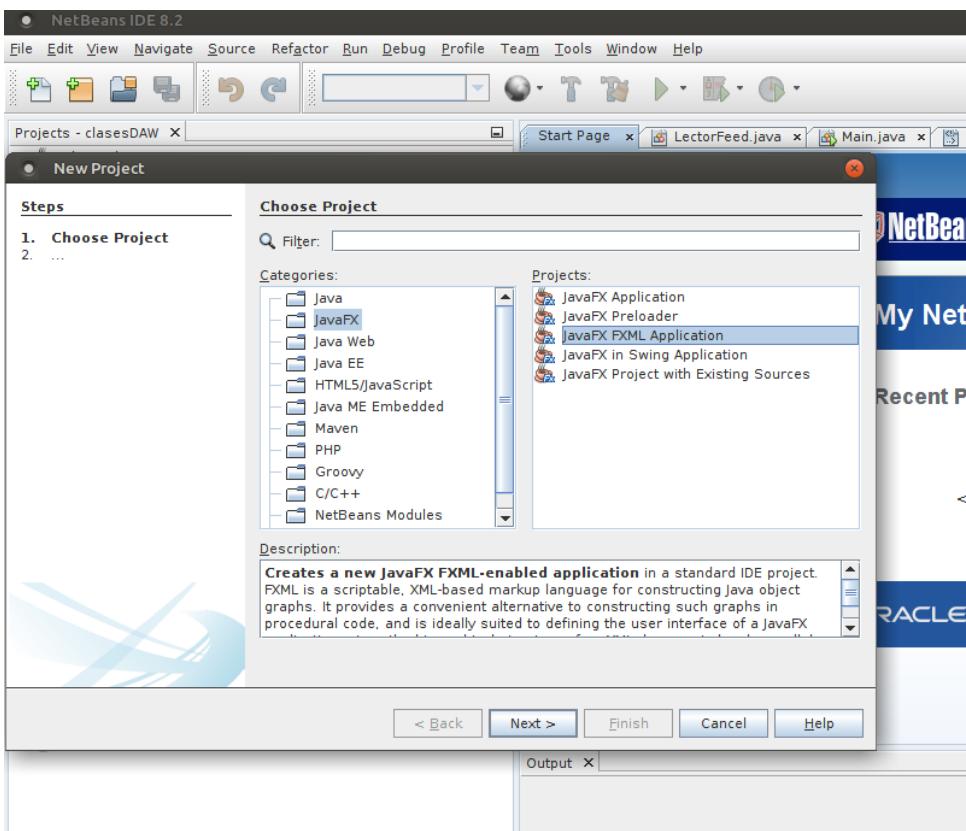
Mediante la anotación: `@FXML` establecemos que estamos con un objeto FXML y es en este fichero donde vemos el método que gestiona los eventos del botón: `handleButtonAction()`

● **Práctica 3:** Realizar la aplicación anterior, modificándola para que Muestre nuestro nombre completo cuando hacemos click en el ratón. Tomar captura de pantalla de la aplicación mostrando cuando pulsamos el botón

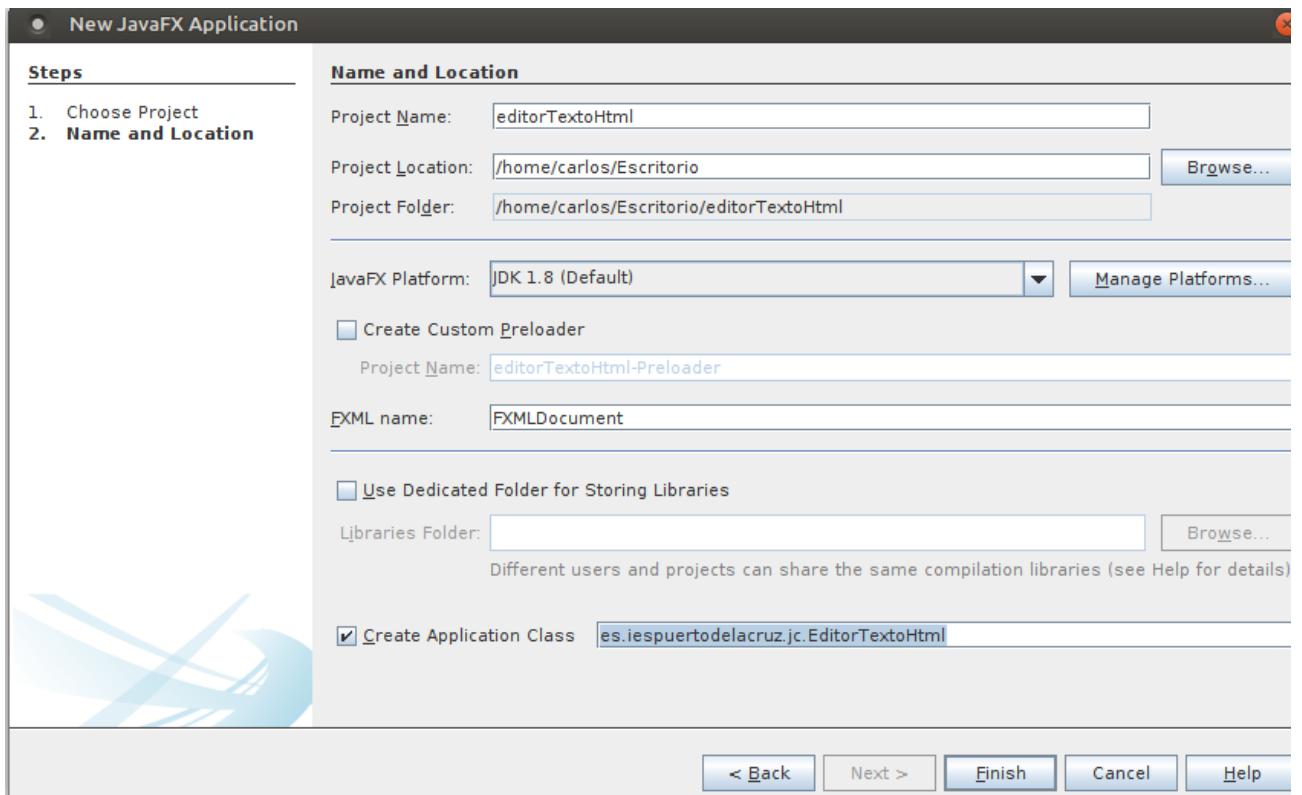
Separación de paquetes MVC en Netbeans JavaFX

Pongamos que estamos haciendo una aplicación que es un editor de texto y queremos hacer una separación apropiada por paquetes

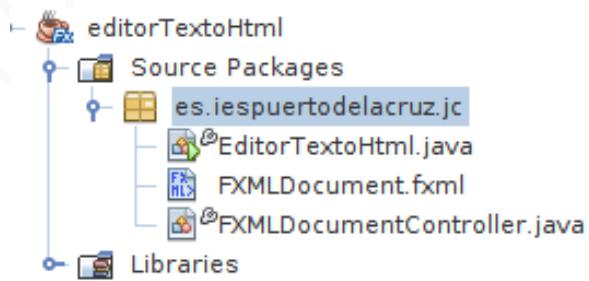
File → New Project → JavaFX → FXML Application



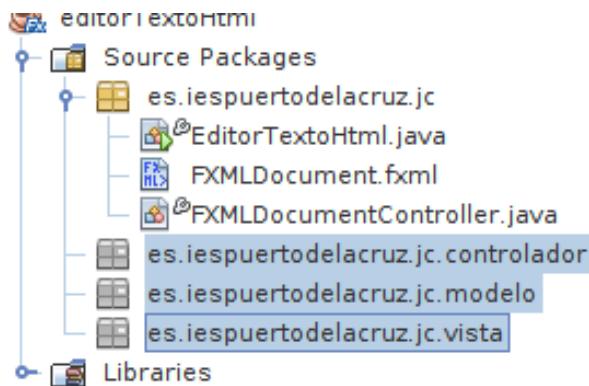
Observar en la siguiente imagen del wizard que aprovechamos para poner un nombre de paquete base apropiado (en terminología dns nombrando nuestro dominio) En el ejemplo: es.iespuertodelacruz.jc



La estructura que nos habrá generado es:



Vamos a crear subpaquetes para modelo, vista, controlador



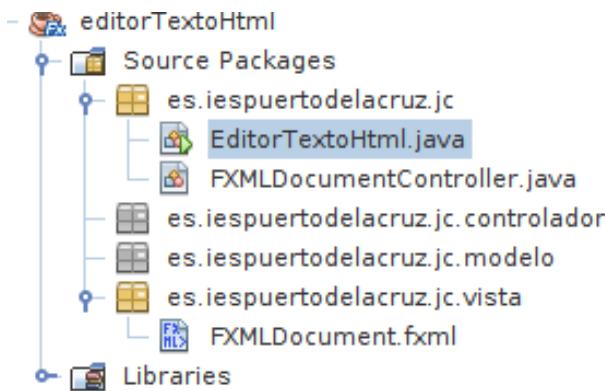
Ahora llevamos el fichero xml a la vista. El problema es que eso hará que no arranque nuestra aplicación (no refactoriza correctamente) tenemos que editar el fichero principal (el que tiene el main) y poner la nueva ruta donde ahora está ubicado el fichero XML

Observar que todos los puntos los convertimos en barras: /

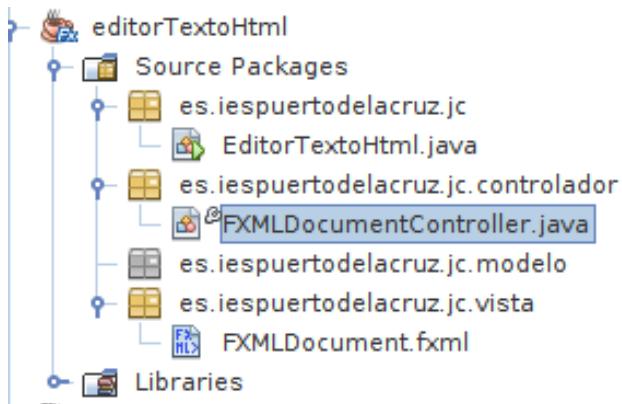
Ahora la ruta en este caso queda: "/es/iespuertodelacruz/jc/vista/FXMLDocument.fxml"

```
/*
public class EditorTextoHtml extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("/es/iespuertodelacruz/jc/vista/FXMLDocument.fxml"));
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```

La estructura después de haber movido el XML nos queda:



Para mover el fichero controlador al paquete que le corresponde también tendremos un problema de refactorización. Primero movemos el fichero controller al paquete que le corresponde



Ahora hay que editar el fichero XML para que apunte al controlador en su nueva ubicación.

Observar que ponemos la ruta completa con los puntos correspondientes al paquete. La línea a modificar sería ésta:

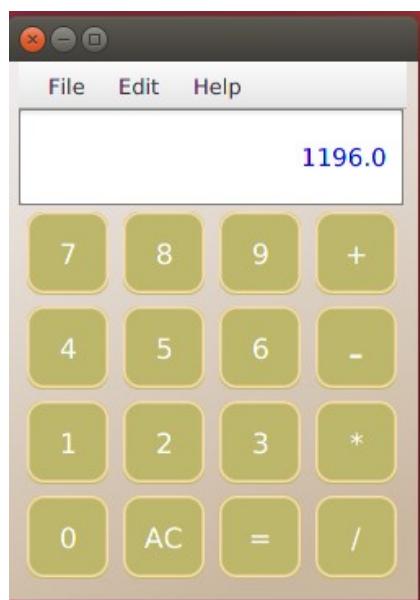
```
<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
    xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="es.iespuertodelacruz.jc.controlador.FXMLDocumentController">
```

Para verla en contexto tenemos una captura de pantalla:

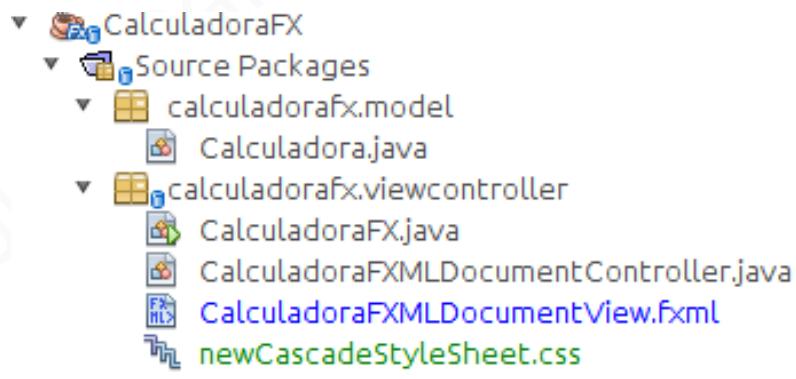
```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
3 <AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320" xmlns:fx="http://javafx.com/fxml/1" fx:controller="es.iespuertodelacruz.jc.controlador.FXMLDocumentController">
3   <children>
3     <Button layoutX="126" layoutY="90" text="Click Me!" onAction="#handleButtonAction" fx:id="button" />
3     <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69" fx:id="label" />
3   </children>
3 </AnchorPane>
```

Segundo proyecto: Calculadora

Vamos a realizar la siguiente calculadora:



En esta aplicación tendremos una idea clara de la separación del modelo: MVC



Se ha elegido crear un paquete para poner los ficheros java del modelo
y otro paquete para guardar la vista y controlador

Calculadora.java contiene la clase publica Calculadora que funciona como una calculadora antigua. Esto es, recibe un número y lo reserva, luego un operador y lo reserva, el segundo número lo reserva. Acto seguido al pulsar el símbolo “=” obtiene el resultado de la operación. De igual manera si en lugar de pulsar el “=” pulsa otro operador obtiene el cálculo anterior y lo convierte en el primer operando para la siguiente operación.

Ej. si se introduce la siguiente secuencia de comandos/instrucciones:

2 + 3 * 7 =

en el momento en que se pulsa el símbolo: “*” ejecuta la operación: $2+3 \rightarrow 5$

y luego toma ese número para que sea el primero operando:

5 * 7 =

mostrando en pantalla 35

Una posible implementación de los métodos del controlador (CalculadoraFXMLDocumentController.java) podría ser:

```
@FXML  
private void operando(MouseEvent event) {  
    Button btn = (Button)event.getSource();  
    int operando = Integer.parseInt(btn.getText());  
    calc.cargarNumero(operando);  
    txtResultado.setText(calc.getResultado());  
}  
  
@FXML  
private void operador(MouseEvent event) {  
    Button btn = (Button)event.getSource();  
    String op = btn.getText();  
    calc.operar(op);  
    txtResultado.setText(calc.getResultado());  
}  
  
@FXML  
private void igual(MouseEvent event) {  
    Button btn = (Button)event.getSource();  
    String op = btn.getText();  
    calc.operar(op);  
    txtResultado.setText(calc.getResultado());  
}  
  
@FXML  
private void limpiar(MouseEvent event) {  
    calc.limpiar();  
    txtResultado.setText(calc.getResultado());  
}
```

Donde calc es un objeto de tipo Calculadora creado en la inicialización:

```

    public void initialize(URL url, ResourceBundle rb) {
        calc = new Calculadora();
    }

```

Es fácil observar que este objeto Calculadora funciona perfectamente tanto para una aplicación de consola como para una aplicación gráfica. Por ejemplo, en consola la ejecución de las instrucciones que nombramos antes: $2 + 3 * 7 =$

```

calc.limpiar();
calc.cargarNumero(2);
calc.operar("+");
calc.cargarNumero(3);
calc.operar("*");
calc.cargarNumero(7);
calc.operar("=");
calc.getResultado();

```

el método limpiar() elimina todo lo que pudiera haber en memoria cargado.

cargarNumero() introduce un operando

operar() realiza la operaciones pendientes y establece el nuevo operador.

Así pues Calculadora.java es el **modelo** de nuestra aplicación perfectamente separado y portable para cualesquier interfaz gráfica que quisieramos utilizar

Como se puede observar en: CalculadoraFXMLDocumentController.java lo que se hace en el controlador es enlazar el elemento gráfico vista FXML con el modelo. Por ejemplo:

```

@FXML
private void operando(MouseEvent event) {
    Button btn = (Button)event.getSource();
    int operando = Integer.parseInt(btn.getText());
    calc.cargarNumero(operando);
    txtResultado.setText(calc.getResultado());
}

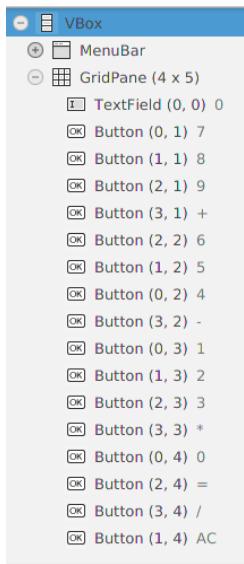
```

nos muestra que toma de la vista el número que aparece en el botón mediante parseInt lo convierte en entero y luego se lo pasa al modelo mediante: calc.cargarNúmero()

finalmente muestra lo que devuelve el modelo en pantalla en el txtResultado que ha creado la vista a tal efecto.

En la vista del navegador del proyecto que se ha mostrado antes también se observa un fichero .css que nos sirve para dar los estilos a nuestra aplicación

Pasos para crear la vista



Dentro de Scene Builder introducimos un contenedor VBox que a su vez va a tener un controlMenuBar y un contenedor GridPane

Vbox es un layout que posiciona todos sus hijos (los componentes) en una fila vertical.

Luego arrastramos un controlMenuBar hacia el VBox

de esta forma será el primer elemento que muestre la aplicación, ya que típicamente se suelen mostrar los menu en la parte alta de la interfaz de usuario (ui)

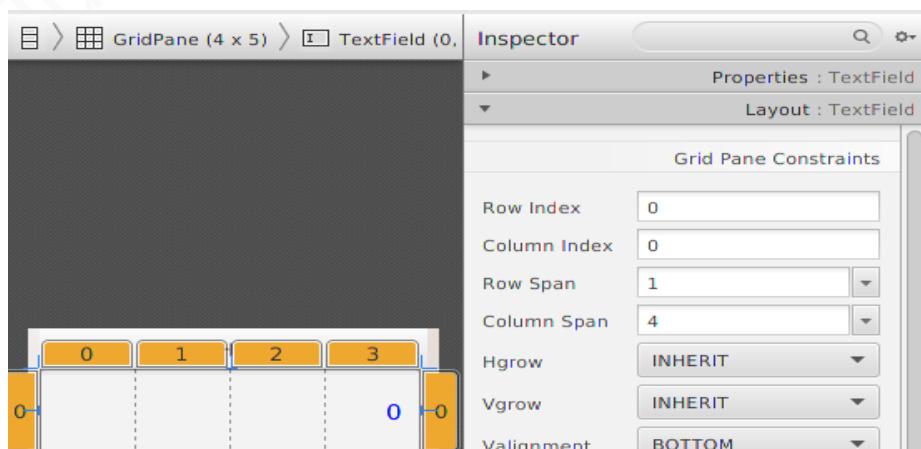
MenuBar se redimensiona automáticamente al ancho del VBox con los elementos menuitem que se vayan introduciendo /quitando

Posteriormente arrastramos un GridPane dentro del VBox

Este GridPane lo hacemos de 5filas y 4columnas

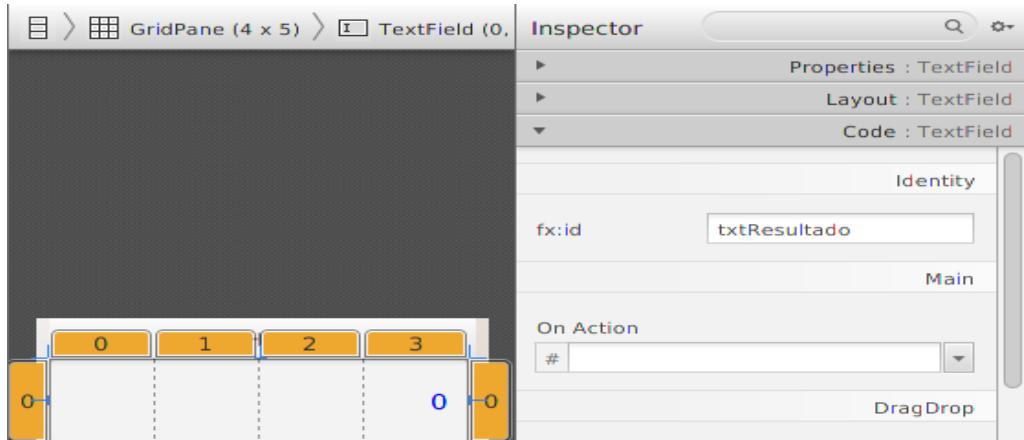


Como ejemplo de como establecer los elementos dentro del grid veamos la siguiente imagen:



En la imagen observamos que el elemento que hemos arrastrado un TextField a la posición 0,0 del grid. Y le hacemos un Column Span de 4 de esa forma la caja de texto para el resultado tomará toda la primera fila del grid

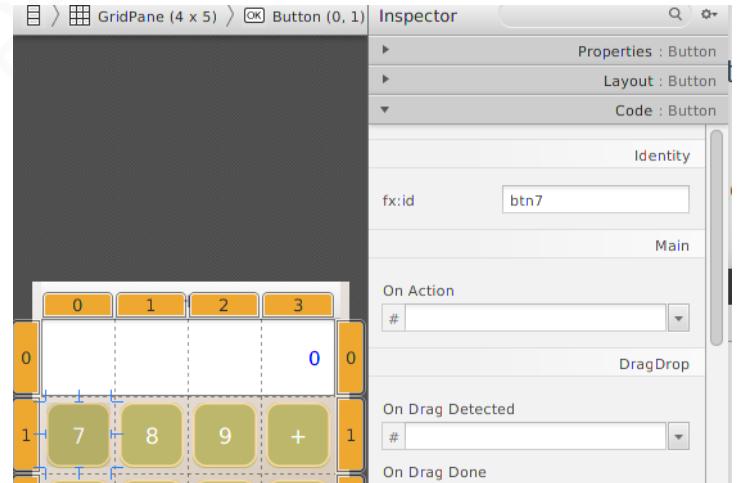
En la parte de code le ponemos el id que luego podremos utilizar tanto en el controlador como en el fichero css. En esta ocasión se ha elegido como id: txtResultado



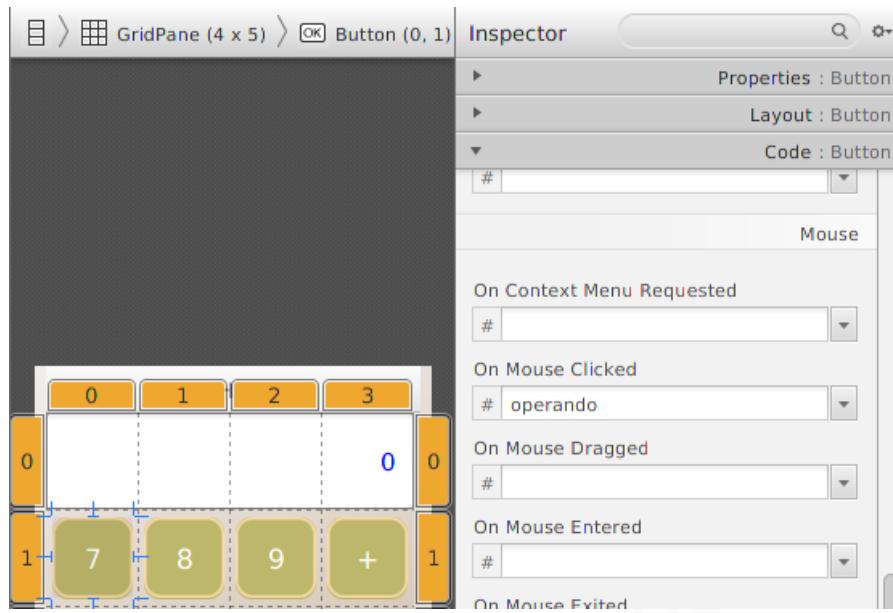
Los botones con números se han elegido como id: btnNum

así el botón para el número 7 nos queda:

id: btn7



Adicionalmente establecemos para el botón el método que va a lanzar cuando sea pulsado:



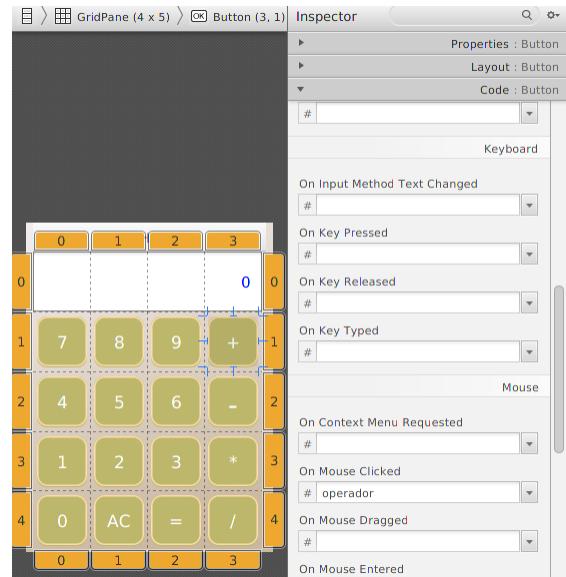
observar que no escribimos los paréntesis en el nombre del método y que lo hemos establecido para el evento: On Mouse Clicked

Para los operadores: “+”, “-”, “*”, “/”, “=” se ha elegido como identificador respectivamente:

btnMas, btnMenos, btnPor, btnDividir, btnIgual

y enlazarles el mismo método a todos ellos: operador()

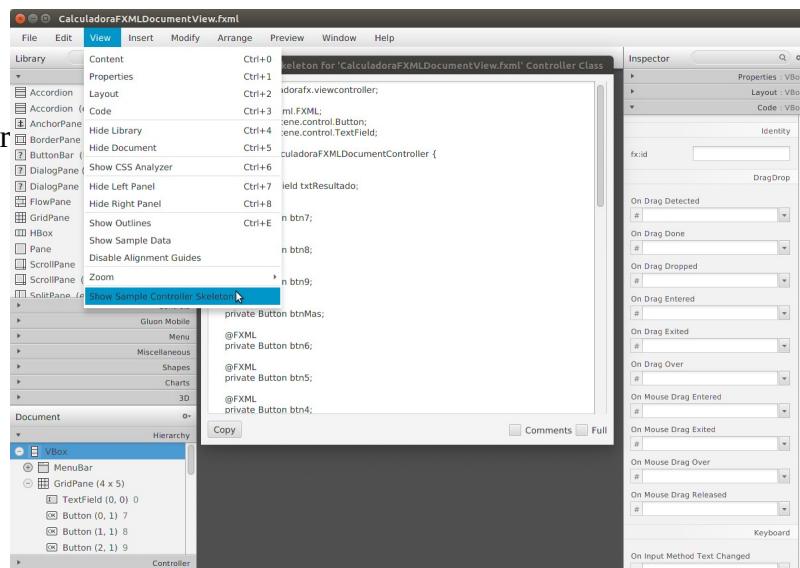
Como ejemplo vemos para el “+”:



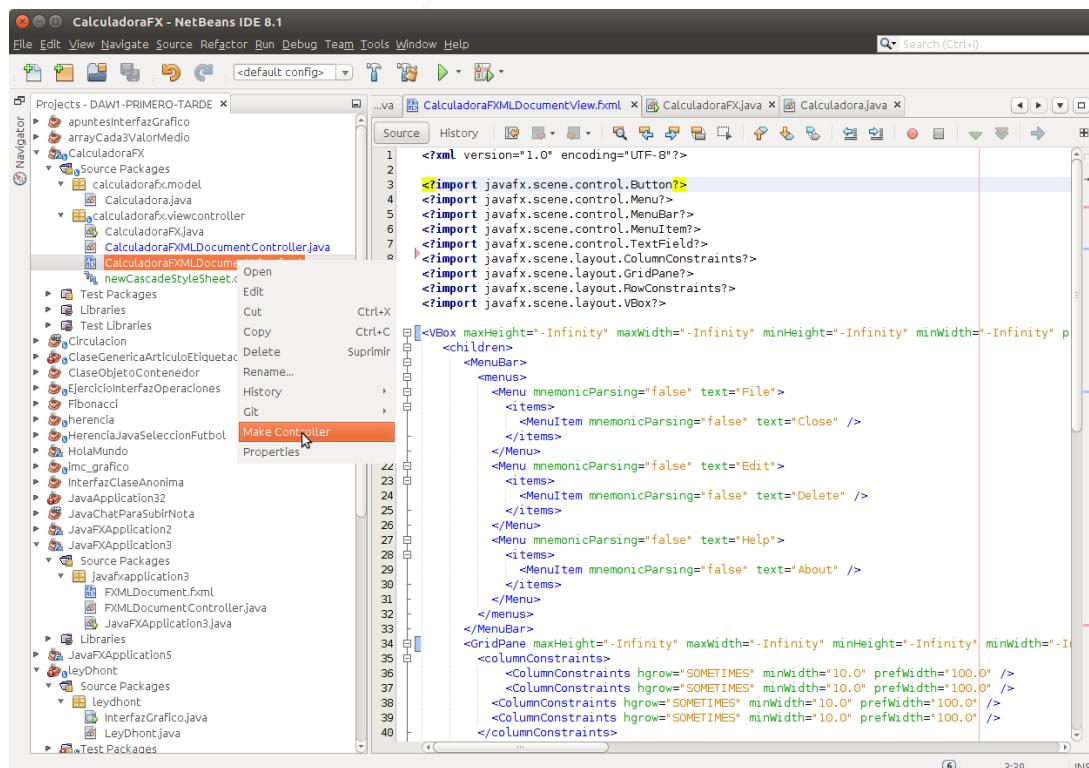
Se puede ver como nos quedaría la estructura de un fichero .java controlador

view → Show

sample controller
skelleton



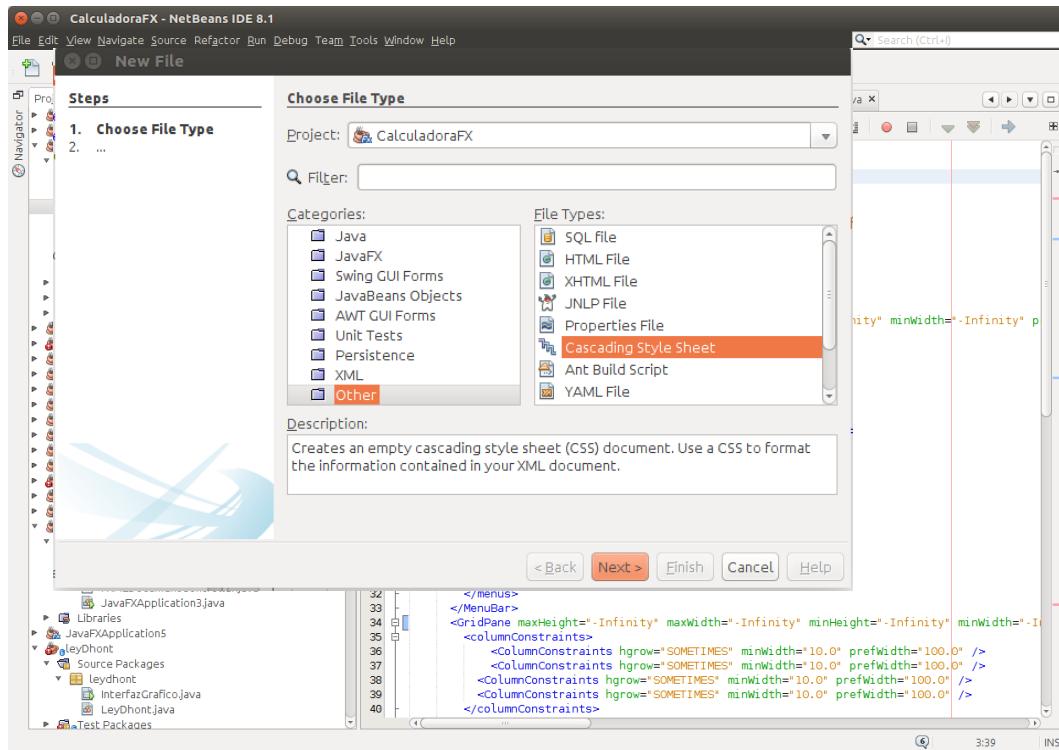
Seleccionar ese texto copiarlo y llevarlo al fichero controller puede sernos muy útil. De cualquier forma tenemos otra alternativa, posiblemente mejor. Nos ponemos en **netbeans sobre el fichero .fxml que nos ha sido generado en el Scene Builder → botón derecho → Make controller**



Nos falta ponerle estilos a la aplicación.

Para ello desde Netbeans botón derecho sobre el paquete donde queremos que quede el fichero:

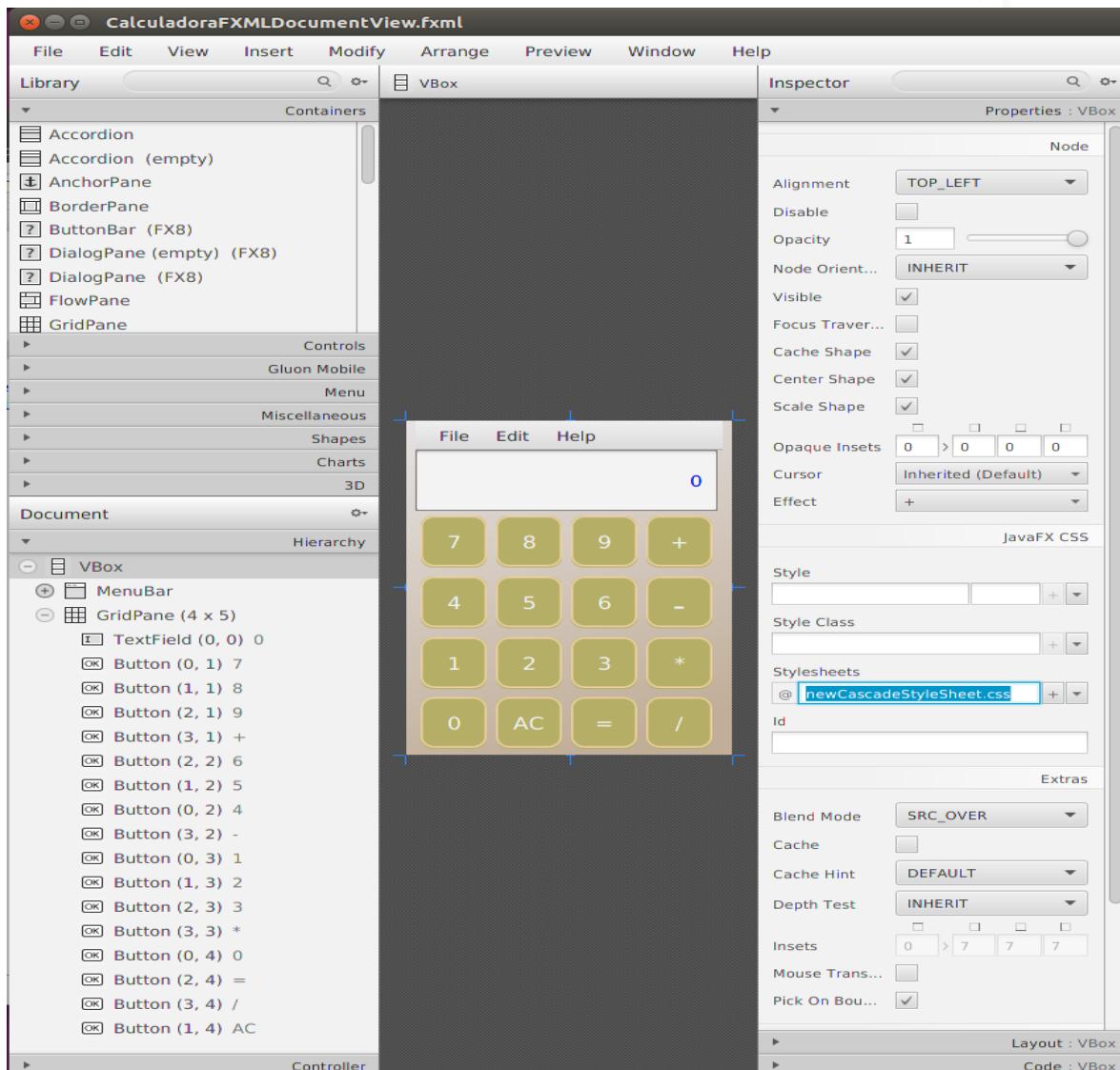
botón derecho → New → Other → Other → Cascading Style Sheet



Una vez creado lo enlazamos con la vista desde el Scene Builder:

Nos ubicamos en el elemento raíz (en el caso del ejemplo sería el VBox), en properties buscamos donde dice Stylesheets y allí usamos el botón “+” que nos abrirá una ventana para seleccionar el fichero.

**Nota: Para ver los efectos que nos queda en el CSS podemos pulsar CTRL+P
(Preview → Show Preview in Window)**



Ahora bien, este fichero CSS utiliza instrucciones específicas, detallaremos alguna, pero lo mejor es remitirse a la documentación CSS que se especificó antes

Podemos usar selectores genéricos:

.button{} → nos aplicará estilos a todos los botones

.button:hover{} → comportamiento cuando el puntero esté sobre un botón

.button:pressed{} → comportamiento cuando pulsamos sobre el botón

.root{} → raíz del documento. En nuestro caso coincidiría con el VBox

.text-field{} → estilos a todos los textfield (en nuestro caso a la caja de resultados)

También podemos acceder mediante el id que hemos generado para luego usar desde código. Así por ejemplo, podemos acceder específicamente al botón con el número 7 mediante:

#btn7{}

Respecto a las instrucciones que podemos utilizar son parecidas a cualesquier CSS. La diferencia es que suelen tener: “-fx” como prefijo. Así por ejemplo, para poner un fondo que tenga un gradiente y un padding como en la imagen de ejemplo. Podemos escribir:

```
.root{
    -fx-background-color: linear-gradient(from 0% 0% to 100% 200%, repeat, #F1EEEE 0%, #CAB7A0 50%);
    -fx-padding: 0 7px 7px 7px;
}
```

Otras instrucciones que se han usado en alguna parte del ejemplo son:

-fx-border-color → color del borde

-fx-background-radius → curvatura en los límites del fondo del elemento

-fx-border-radius → curvatura en el borde del elemento

-fx-text-fill → color de la letra

-fx-font → tamaño de la letra, negrita etc. Ej: -fx-font: 10px bold; → 10px y negrita

-fx-border-width → tamaño del border

-fx-scale-x → escalar el objeto en la coordenada x

-fx-scale-y → escalar el objeto en la coordenada y

-fx-font-size → tamaño de la letra

Para terminar vamos a fijarnos en un trozo de código que pusimos antes del controlador:

```
@FXML  
private void operando(MouseEvent event) {  
    Button btn = (Button)event.getSource();  
    int operando = Integer.parseInt(btn.getText());  
    calc.cargarNumero(operando);  
    txtResultado.setText(calc.getResultado());  
}
```

Fijarse que podemos saber que objeto ha desencadenado el evento mediante:

event.getSource()

de esa forma en el código mostrado se obtiene el número del botón y se lo podemos pasar al modelo (calc.cargarNumero(operando))

● **Práctica 4:** Realizar la aplicación de calculadora descrita. Tener en cuenta que cuando se pasa el ratón encima de un botón este cambia de color y se remarca el borde. Cuando se presiona el botón toma el color de fondo y escala ligeramente el botón en x, y

Tercer proyecto: Coches FX

Teniendo en cuenta la clase Coche que hemos creado en unidades precedentes vamos a realizar la siguiente aplicación:

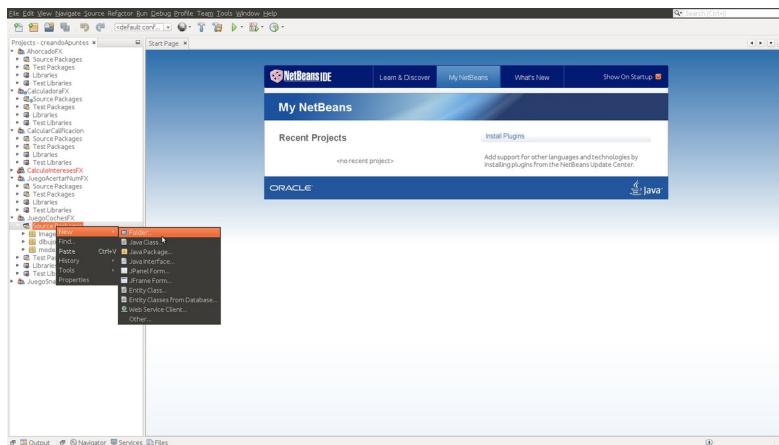


- Cuando se pulsa el botón instrucciones muestra el texto que se muestra en la primera pantalla
- Pulsando con el ratón encima del coche deseado lo seleccionamos e interactuamos con el mediante los comandos descritos en las instrucciones: e para encender, b para bajar el freno de mano, etc Observamos que los mensajes son coincidentes con los que hemos desarrollado para la clase Coche anteriormente
- Cuando en el desplazamiento de los coches llegan a coincidir se abrirá una ventana emergente que nos avise que ha habido un choque.
- Si un coche trata de desplazarse más allá de los límites del escenario no podrá

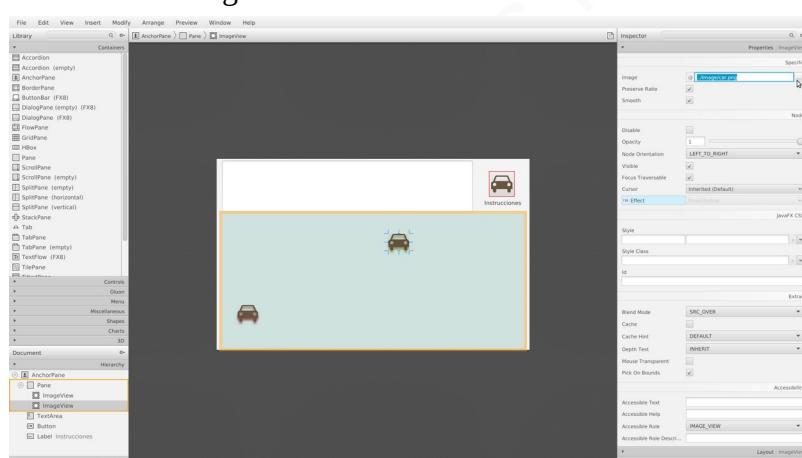
Con este proyecto aprenderemos a incluir imágenes en una aplicación trabajar con el objeto imageview, poner una imagen mediante CSS a un botón y a capturar eventos de teclado

Agregar imágenes a un proyecto

- Creamos una carpeta (por ejemplo images)
botón derecho sobre Source Packages → New → Folder



- Si tenemos la imagen en el portapapeles podemos pegarla directamente en netbeans encima de la carpeta la imagen
- desde scene builder al agregar el imageview pulsamos en image el botón de tres puntos y localizamos la imagen



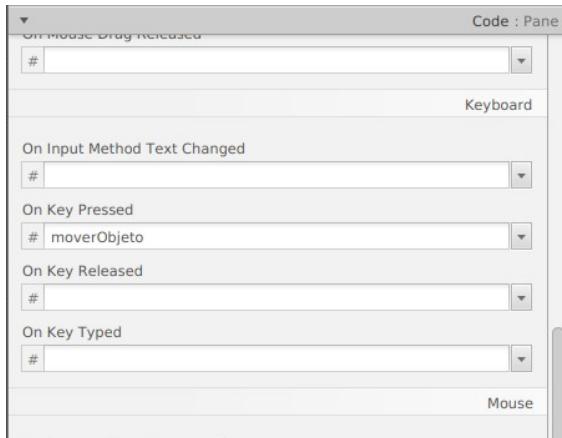
- Para agregar una imagen a un botón podemos hacerlo desde CSS.
Por ejemplo, imaginemos que quisiéramos que todos los botones fueran iguales, entonces usaríamos la clase .button:

```
.button{  
    -fx-background-image: url("../Image/car.png");  
    -fx-background-size: 50px 40px;  
    -fx-background-repeat: no-repeat;  
    -fx-background-position: center;  
}
```

En lo anterior observamos que Le hemos agregado la imagen que previamente habíamos guardado en la carpeta de netbeans. Ponemos un tamaño a la imagen de fondo, que no se repita y que quede centrada

Capturando un evento de teclado

Debemos pensar en que espacio queremos tomar el listener. Imaginemos que hemos determinado que sea en el Pane contenedor del escenario. Crearemos desde el scenebuilder en: “On key Pressed” el nombre del evento. Luego como en otras ocasiones desde netbeans con botón derecho sobre el fxml le pedimos que nos cree el controlador



Ahora en el método que nos ha generado en el controlador:

```
private void moverObjeto(KeyEvent event) {  
  
    KeyCode kc = event.getCode();  
    switch(kc){  
        case UP: //System.out.println("arriba");  
            mensaje=coche.moverArriba(paso);  
            break;  
        case DOWN: //System.out.println("abajo");  
            mensaje=coche.moverAbajo(paso);  
            break;  
        case LEFT: //System.out.println("izquierda");  
            mensaje=coche.moverIzquierda(paso);  
            break;  
        case RIGHT: //System.out.println("derecha");  
            mensaje=coche.moverDerecha(paso);  
            break;  
    }  
}
```

Vemos que tomamos un **KeyCode**. ¡¡Cuidado!! hay varios. El import correspondiente es:

```
import javafx.scene.input.KeyEvent;
```

Vemos que llamamos a diferentes métodos del modelo según que el usuario haya pulsado una tecla u otra de las ArrowKeys

Desplazando un objeto en un Pane

Los métodos **getLayoutX()**, **getLayoutY()**, **setLayoutX()**, **setLayoutY()** nos permiten tomar la posición (x,y) de un objeto y establecer una nueva posición (x,y) de esa forma conseguimos desplazar el objeto mediante coordenadas cartesianas. Hemos de tener en cuenta que los espacios comienzan siendo la posición (0,0) en la esquina superior izquierda, y terminan en la esquina inferior derecha coincidente con la posición: (**pane.getWidth()**, **pane.getHeight()**)

Creando ventana emergente

Los objetos del tipo: Alert permiten crear ventanas de Aviso, Error, Información, Confirmación

El siguiente ejemplo nos muestra una ventana con un mensaje que nos avisa que ha habido un choque:

```
Alert alert = new Alert(Alert.AlertType.INFORMATION , "Choque! ! !");  
alert.showAndWait();
```

Cuarto Proyecto: Ponderaciones



En la primera imagen observamos que el botón “agregar” nos permite ir poniendo diferentes ponderaciones. Así se van creando controles en tiempo de ejecución (runtime) . En concreto cada vez que pulsamos en el botón agregar se crea un button, un textfield y un label para esa ponderación. En la segunda imagen vemos que se han creado 3 ponderaciones y se han introducido 6 notas. Ese ejemplo refleja que se hubieran realizado 6 pruebas:

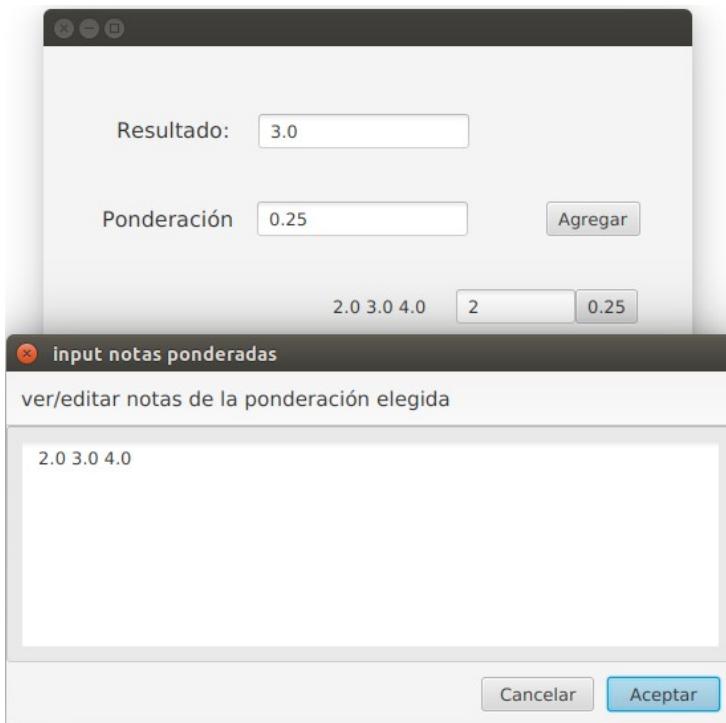
- con un peso de 0.25 las calificaciones: 2, 4
- con un peso de 1 las calificaciones: 5, 3, 9
- con un peso de 2 las calificaciones: 8

La fórmula para obtener el resultado en ese caso sería:

$$(0.25*2 + 0.25*4 + 1*5 + 1*3 + 1*9 + 2*8)/(0.25 + 0.25 + 1 + 1 + 1 + 2)$$

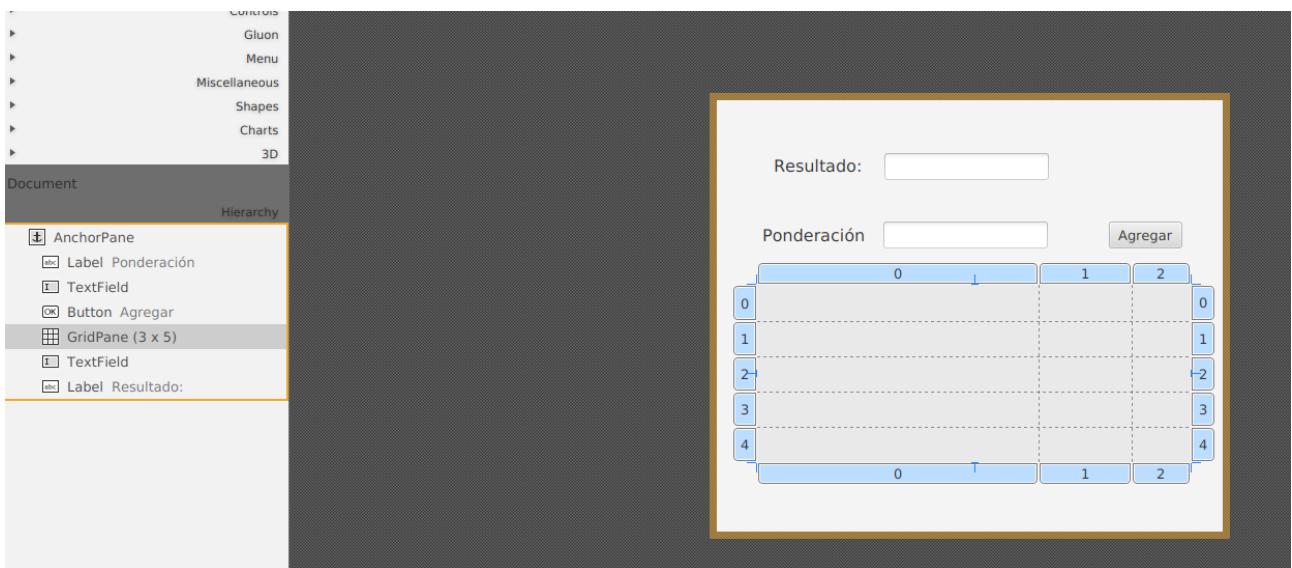
Como se ve se multiplica cada calificación por su peso correspondiente se suma todo y se divide entre la suma de todas las ponderaciones

En la siguiente pantalla vemos otro método de entrada/edición de datos de la aplicación:



Al pulsar con el ratón encima del label de la ponderación que queremos (en el ejemplo la ponderación de 0.25) se abre una ventana emergente que nos permite introducir/editar las puntuaciones correspondientes a ese peso de 0.25. Al pulsar en Aceptar esas puntuaciones son las que quedarán establecidas para ese peso y las veremos en el label

Veamos el scene builder correspondiente:



Vemos que hay un gridpane para poder agregar los controles en runtime. En la columna más a la izquierda iría el label, a continuación el textfield y finalmente el button

De esa forma cada fila conformarían los objetos para cada ponderación

Vamos a ver un ejemplo de un botón creado en runtime:

```
Button boton = new Button();  
  
boton.setText("0.25");  
//boton.setId("btn025");  
boton.addEventHandler(MouseEvent.MOUSE_CLICKED, evt->agregarDatos(evt));  
this.gridControlesRuntime.add(boton, 2, 1);
```

Vemos que establecemos el texto que va a mostrar el botón, entre comentarios que podemos si queremos establecerle un Id, y vemos como podemos agregarle un listener mediante addEventHandler(). En este caso en concreto le estamos agregando al botón el listener para cuando se hace click de ratón en el botón. Observar que: evt -> agregarDatos(evt) nos está diciendo que cuando se pulse el botón se ejecutará el código que hayamos puesto en el método: agregarDatos()

En la última línea vemos que estamos posicionando el botón dentro del grid mediante el método: grid.add(). Así this.gridControlesRuntime.add(boton, 2, 1) agrega el botón en la segunda columna de la primera fila

Otros métodos que puede ser que utilicemos en el código para los objetos runtime:

- setPrefWidth() // nos establece el ancho del objeto

- setAlignment(Pos.CENTER_RIGHT) // nos permite establecer el alineado del texto que contiene si lo queremos a la izquierda, derecha, etc

Creando cuadro de diálogo personalizado

El siguiente código es de un método que nos crea un cuadro de diálogo personalizado

```
/***
 * Creamos un nuevo objeto de Diálogo que incorporé un textarea y así se pueda
 * editar cómodamente las notas para la ponderación elegida
 * @param ponderacion es la ponderación que debemos buscar en nuestro modelo y tomar así las notas correspondientes
 * @return el nuevo cuadro de diálogo
 */
Dialog crearDialogoInputTextArea(double ponderacion){

    Dialog dialog = new Dialog<>();
    dialog.setTitle("input notas ponderadas");
    dialog.setHeaderText("ver/editar notas de la ponderación elegida");

    dialog.getDialogPane().getButtonTypes().addAll(ButtonType.OK,ButtonType.CANCEL);
    TextArea txaNotas = new TextArea();

    dialog.getDialogPane().setContent(txaNotas);
    dialog.setResultConverter( button ->{
        if( button == ButtonType.OK)
            return txaNotas.getText();
        else
            return null;
    });
    return dialog;
}
```

- setTitle() funciona igual que cuando lo hacemos a stage (el título de la ventana)
- setHeaderText() informa al usuario de lo que se pretende realizar en la ventana de diálogo
- Vemos que podemos agregar botones. Es bastante típico agregar un botón de OK y otro de Cancelar. Es lo que se reproduce en el código anterior

- Creamos un textarea y lo agregamos al cuadro de diálogo mediante: dialog.getDialogPane().setContent()
- setResultConverter() es un método que sirve para determinar que información va a regresar el cuadro de diálogo cuando se cierre. En nuestro caso es que si el usuario ha pulsado en OK le devuelva el contenido del textare

Vamos a ver ahora como usamos ese método y manejamos la información que nos regresa

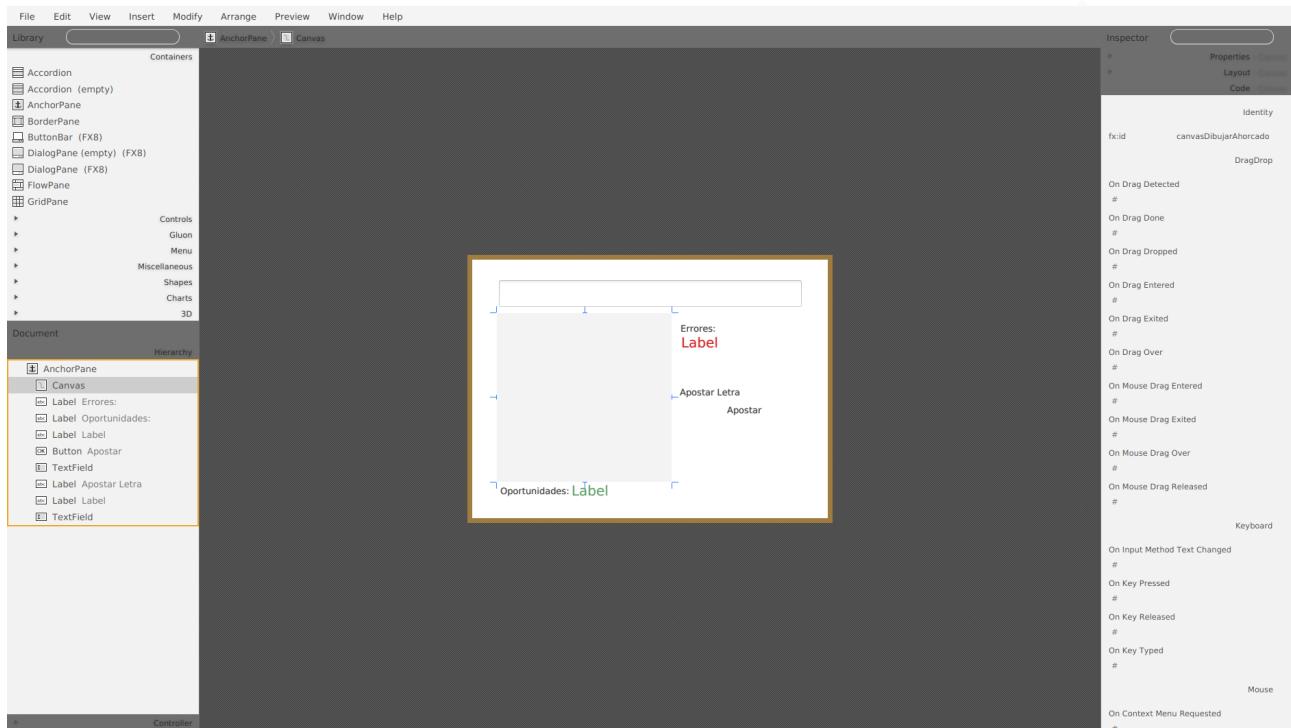
```
Dialog dialog = crearDialogoInputTextArea(ponderacion);
Optional<String> result = dialog.showAndWait();
if(result.isPresent()){
    // result.get() nos dará el contenido del textarea
}
```

- En la primera línea creamos el cuadro de diálogo
- En la segunda línea asignamos un Optional a lo que nos regrese el cuadro de diálogo cuando cierre (observar el showAndWait() que hace que se muestre la ventana hasta que el usuario actúe sobre ella cerrándola)
- En el if() estamos actuando como ya sabemos con los Optional. Ahí es donde manejamos el resultado que nos haya devuelto el cuadro de diálogo, que en este caso será el contenido del textarea

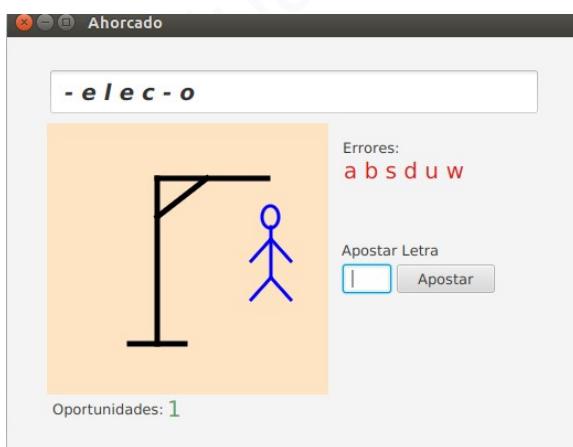
Quinto Proyecto: Ahorcado

El objetivo de este proyecto es iniciarnos en Canvas. En inglés canvas viene a ser “lienzo” así que es algo muy apropiado para dibujar-pintar. Es un objeto que nos encontramos en html-javascript en JavaFX, etc

En concreto para nuestro caso hay un objeto específico en scenebuilder simplemente lo arrastramos y le ponemos un id para poder acceder desde código



Vamos a ver como nos quedará la aplicación:



En la imagen se han producido 6 errores y nos queda un único intento equivocado.

Es fácil observar que el dibujo en el canvas está conformado por líneas y por un ovoide

Para usar el canvas supongamos que el id del objeto es: canvasDibujarAhorcado entonces procederíamos así:

```
//tomamos un graphicsContext
GraphicsContext gc = canvasDibujarAhorcado.getGraphicsContext2D();
//establecer el grosor de la linea de dibujo
gc.setLineWidth(5.0);
// establecer el color para dibujar lineas:
gc.setStroke(Color.BLUE );
//dibujar linea
gc.strokeLine(200, 50, 200, 75); // una linea desde el (200,50) hasta el punto
(200,75)
```

A tener en cuenta otras instrucciones que vamos a necesitar:

//limpiar una región (en concreto todo el lienzo)

```
gc.clearRect(0, 0, canvasDibujarAhorcado.getWidth(), canvasDibujarAhorcado.getHeight());

gc.setFill(Color.BLUE); // establecer el color de relleno, por ejemplo de un rectángulo
gc.fillRect(0,0,x,y) ; // dibujar un rectángulo lleno de color de la coordenada (0,0) a la
(x,y)

gc.strokeOval(195, 75, 15, 20); // dibujar un óvalo. (195,75) es el punto para ubicarnos 15
el ancho, 20 el largo
```

Para completar el juego tener en cuenta:

- se abrirá una alerta avisando de si se ha ganado o perdido informando de cuál era la palabra
- En total se dispone de 7 intentos. Repetir un mismo error descuenta entre los intentos pero NO se muestra en la parte de errores dos veces la misma letra
- Cuando se pierde el dibujo del ahorcado cambia a rojo:



Sexto Proyecto: Elecciones Dhont, Proporcional

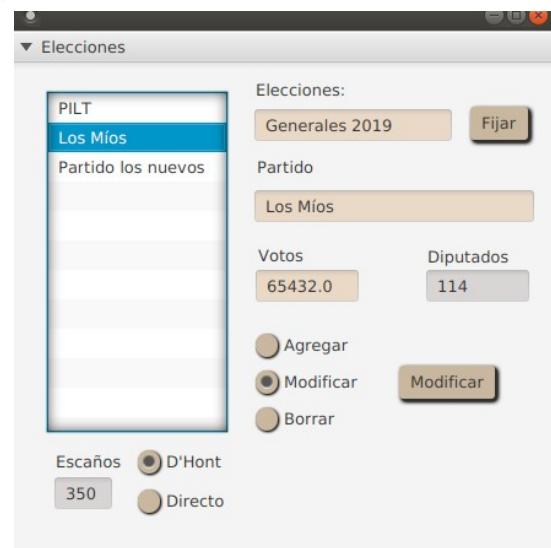
Pantalla inicio estableciendo elecciones:



Una vez establecido ahora datos de partidos:



Una vez hemos agregado los partidos hacemos click en cualquiera de la lista y podemos ver sus diputados asignados y modificar la información



Observar que si hacemos click en el radio button a Directo cambiará el método de elección y los diputados se calcularán de otra forma

Código significativo

Se está usando una ListView (es el único elemento diferenciador respecto a otros proyectos) para ir poniendo los partidos y seleccionarlos:

```
@FXML  
private ListView<Partido> lstPartidos;
```

Para agregar un partido a la vista podría valer lo siguiente

```
p = new Partido(txtPartido.getText(), Integer.parseInt(txtVotos.getText()));  
ld.agregarPartido(p); //← esta línea agrega partido al modelo ( clase Eleccion )  
lstPartidos.getItems().add(p);
```

Vemos que agregamos el partido con getItems().add()

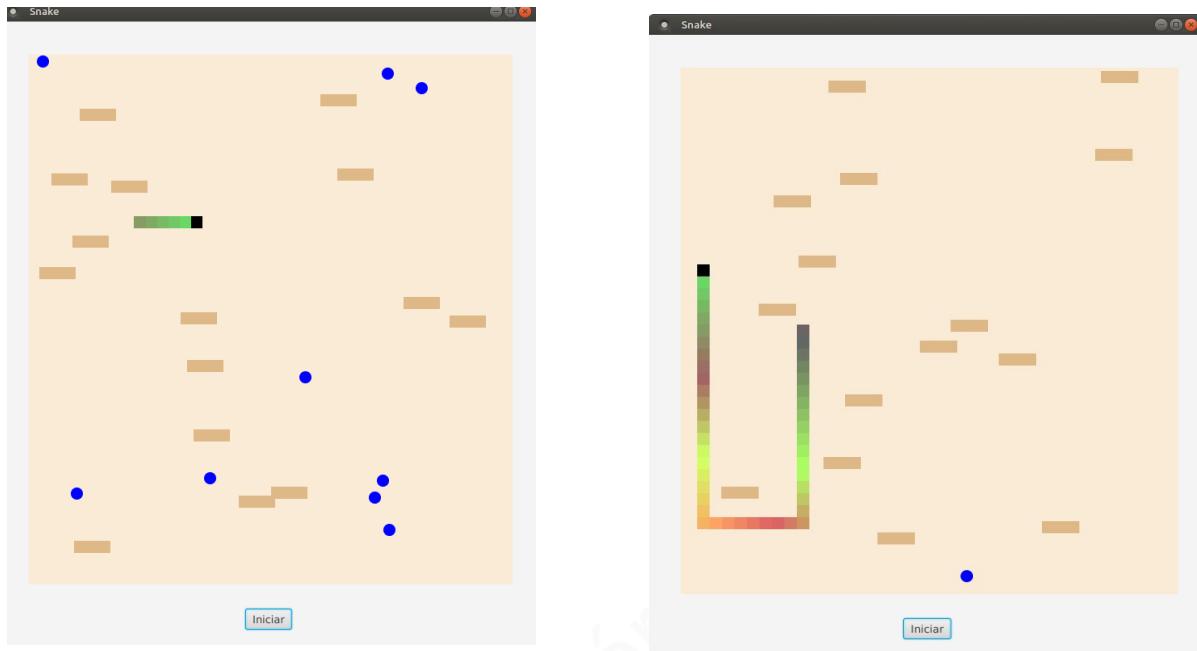
Obtener el partido que seleccionamos (click) en la lista:

```
Partido p = lstPartidos.getSelectionModel().getSelectedItem();
```

Limpiar la lista: ListView para rellenar de nuevo con otros partidos:

```
lstPartidos.getItems().clear(); //← limpia la vista: ListView  
lstPartidos.getItems().setAll(ld.getPartidos()) //← pone partidos del modelo
```

Sexto Proyecto: Snake



Este proyecto es similar a El Ahorcado (usa un canvas) pero difiere en que el juego “se mueve” sin necesidad de interacción del usuario.

Pensemos en fotogramas de una película. Nosotros vamos a dibujar el escenario completo nuevamente a una cantidad de tiempo establecida y así si la serpiente se desplaza u otros elementos aparecen/desplazan se redibujará de forma animada

¿ cómo aportamos esa “animación” ? Aquí ya aparece por primera vez el concepto de varios trabajos a la vez. Va a haber algo parecido a una tarea programada que se encargará de avanzar y redibujar el juego según el movimiento que esté establecido:

```
timeline = new Timeline(new KeyFrame(  
    Duration.millis(velocidadJuego), ae -> moverJuego()  
));  
timeline.setCycleCount(Animation.INDEFINITE); // ← no para nunca
```

Gracias a TimeLine establecemos que cada cuanto tiempo nosotros queramos (velocidadJuego) ejecutamos el método: moverJuego()

Ese método lo que hace (la mayor parte de las veces) es mover a la siguiente posición que le corresponde a la serpiente (a nivel del modelo) y redibujar el canvas (esto es, trasladar en la vista la situación actual en el modelo)

Pausar-parar/iniciar el juego:

Con el siguiente código cambiamos el estado previo (el juego solo puede estar parado o en movimiento) al contrario, simplemente usando las ordenes de timeline: stop() y play()

```
if(juegoIniciado){  
    timeline.stop();  
  
}else{  
    timeline.play();  
}  
juegoIniciado = !juegoIniciado;
```

Puede haber problemas de coherencia en el renderizado si se está ejecutando la “tarea programada” mediante el timeline y la interacción del usuario al pulsar una tecla. Se recomienda parar el juego en ese momento y después de hacer los cambios que implican en el modelo haber pulsado una tecla, volver a activarlo

Para conseguir la variación de colores de la serpiente nos basamos en rgb:

```
gc.setFill(Color.rgb(red,green,100));
```

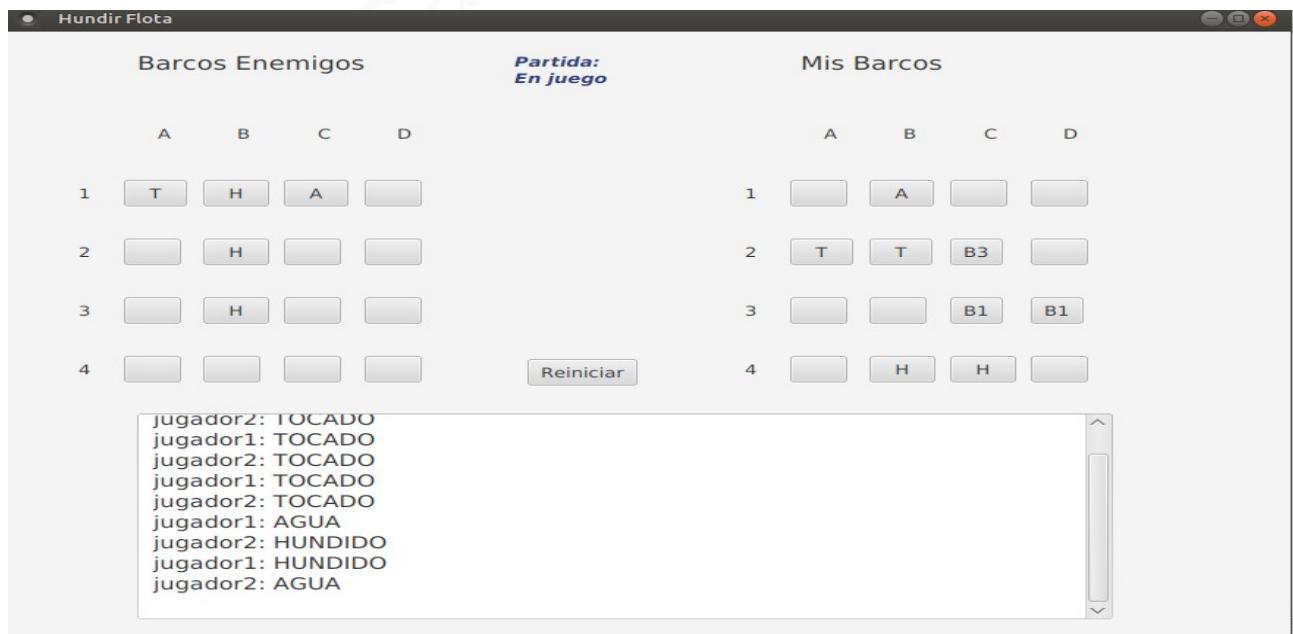
simplemente se varía progresivamente las variables que uno quiera (en el ejemplo la variable: red y green) y así los recuadros irán pasando de un tono a otro progresivamente

Séptimo proyecto: Hundir la flota

El juego será siempre contra la máquina y se visualizan dos tableros: el propio (se podrá ver los barcos que tenemos puestos y los impactos del enemigo) y el del contrario: únicamente si agua, tocado, hundido a cada acción nuestra:



Veamos una pantalla después de haber lanzado bombas sobre diferentes casillas del enemigo:



Vemos que el enemigo nos ha hundido un barco (el enemigo siempre juega justo después que nosotros hagamos click lanzando un ataque en el tablero contrario) De tal forma que nosotros observamos lo que ocurre al haberse dado dos acciones: nuestro ataque y el del contrario

Este juego lo que tiene de diferente respecto a otros proyectos es el rellenado automático de las casillas de la vista

Veamos como “inicializamos” el juego:

```
@Override  
public void initialize(URL url, ResourceBundle rb) {  
  
    grids = new GridPane[]{gridJugador,gridEnemigo};  
    for (GridPane grid : grids) {  
        for (int i = 1; i <= Tablero.size; i++) {  
            for (int j = 1; j <= Tablero.size; j++) {  
                CasillaFX casillaFX = new CasillaFX();  
                casillaFX.setOnAction(evt->apostar(evt));  
                casillaFX.x = i-1;  
                casillaFX.y = j-1;  
                grid.add(casillaFX, i, j);  
            }  
        }  
    }  
  
    reiniciando();  
}
```

Observar que tenemos un elemento gráfico propio que hemos creado (pertenece a la Vista) llamado CasillaFX:

```
public class CasillaFX extends Button {  
    private Casilla casilla;  
    public Integer x;  
    public Integer y;  
    public boolean revincularModelo(Tablero t){  
        if(x != null && y != null){  
            casilla = t.casillas[x][y];  
            return true;  
        }else{  
            return false;  
        }  
    }  
    ... // más código aquí  
}
```

¿ por qué hacemos eso ? Porque con un simple botón no tenemos bien enlazado nuestro modelo (las casillas de nuestro modelo: Casilla) con la representación gráfica. Más adelante hablamos del método: revincularModelo()

Si generamos este objeto, cada vez que una Casilla del modelo sea modificada tendremos esa información localizable fácilmente en nuestra vista

Ahora para establecer el vínculo veamos un método:

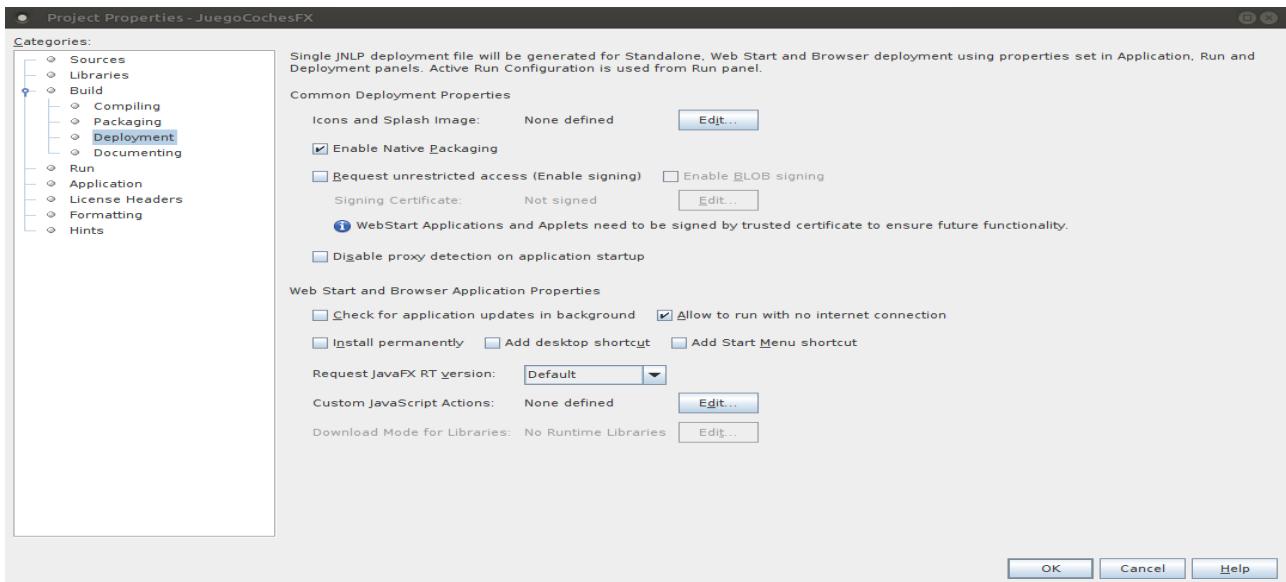
```
private void gridAsociarCasillaFX(GridPane grid, Tablero tablero){  
    ObservableList<Node> children = grid.getChildren();  
    for (Node node : children) {  
        try{  
            CasillaFX cFX = (CasillaFX)node;  
            cFX.revincularModelo(tablero);  
  
        }catch(Exception ex){}  
    }  
}
```

Aquí lo que se ve es como aprovechamos el modelo jerárquico (parecido al DOM de HTML) de JavaFX y recorremos todos los nodos hijos del GridPane (serán los botones del tablero) y hacemos uso del método que está más arriba: revincularModelo() para establecer el vínculo entre la CasillaFX y la Casilla del Modelo

El resto del código no tienen ninguna circunstancia especial a nivel gráfico.

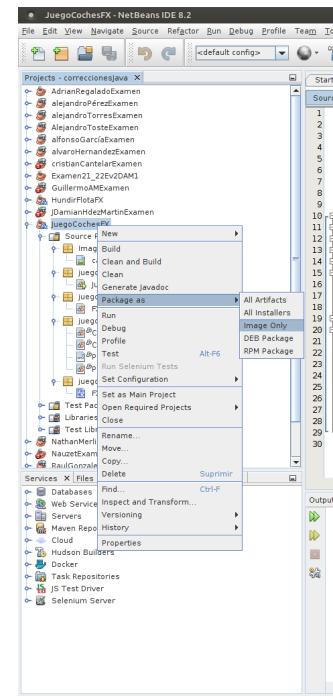
Anexo empaquetar una aplicación gráfica

Botón derecho sobre el proyecto → Properties → Build → Deployment → Enable Native Packaging



Ahora ya se puede generar el empaquetado.
Botón dcho sobre el proyecto → Package as → Image only

También hay opciones para generar un .exe o un .deb seg\xfano el sistema operativo donde estemos



Anexo: Crear un jar ejecutable en Netbeans 13 JavaFX

Después de Java 8 se separaron en módulos las diferentes partes de Java. Así JavaFX se convirtió en un módulo independiente y de hecho es un proyecto con su propio mantenimiento separado

Como Netbeans 13 no precisa de permisos de administrador para su instalación es cómoda su instalación en el propio home de usuario (buscar la que corresponda en un buscador web) Por ejemplo:

<https://netbeans.apache.org/download/nb13/nb13.html>

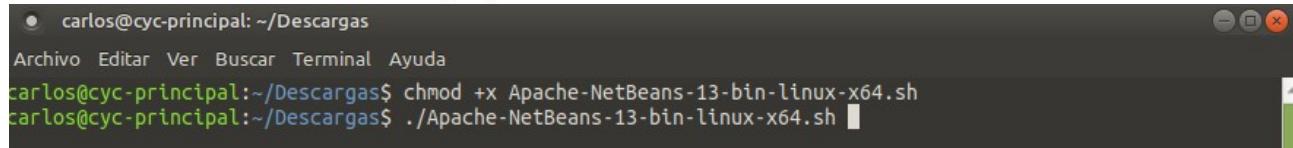
En este caso elegimos una instalación linux así que el enlace debe ser parecido a:

<https://www.apache.org/dyn/closer.cgi/netbeans/netbeans-installers/13/Apache-NetBeans-13-bin-linux-x64.sh>

Tendremos un script .sh que quizás tenemos que ponerle permisos de ejecución:

chmod +x nombredelscript.sh

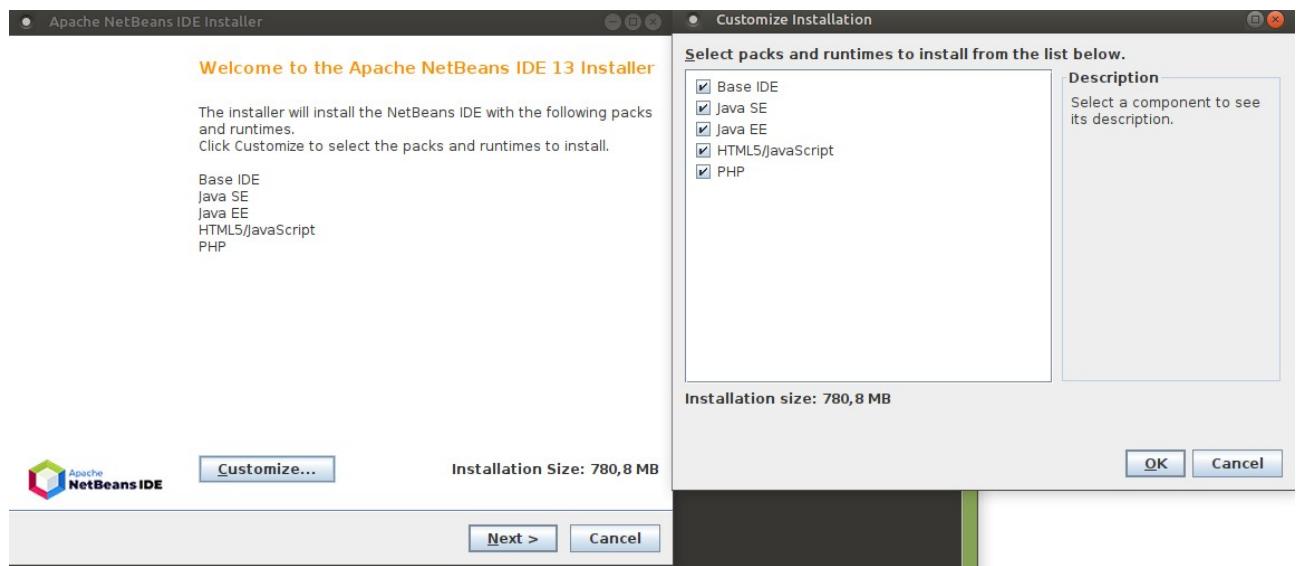
y finalmente lo lanzamos (se presupone que tenemos una terminal en la misma carpeta donde hemos descargado el script de instalación):



```
carlos@cyc-principal: ~/Descargas
Archivo Editar Ver Buscar Terminal Ayuda
carlos@cyc-principal:~/Descargas$ chmod +x Apache-NetBeans-13-bin-linux-x64.sh
carlos@cyc-principal:~/Descargas$ ./Apache-NetBeans-13-bin-linux-x64.sh
```

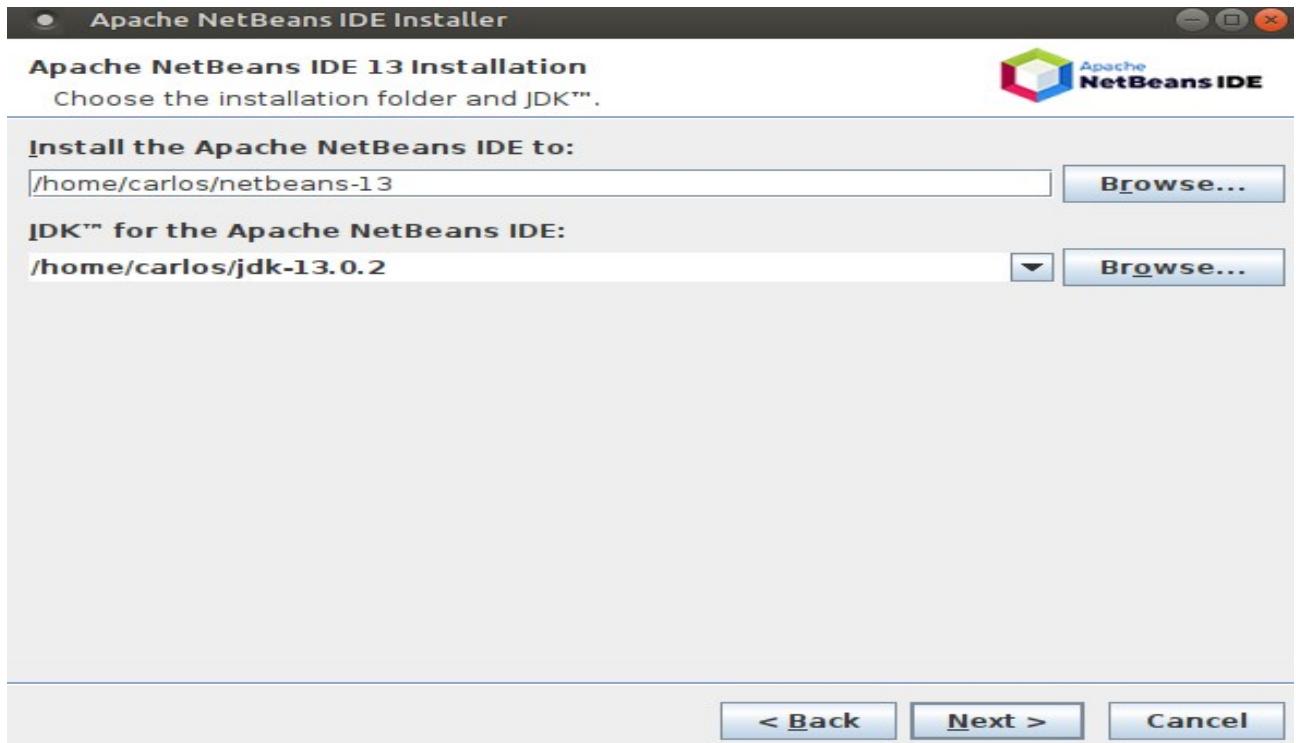
A screenshot of a Linux terminal window titled "carlos@cyc-principal: ~/Descargas". The window has a dark theme with white text. It shows the user's path, menu bar options (Archivo, Editar, Ver, Buscar, Terminal, Ayuda), and a command-line interface. The user has run two commands: "chmod +x Apache-NetBeans-13-bin-linux-x64.sh" and "./Apache-NetBeans-13-bin-linux-x64.sh". The second command is still executing, indicated by a small progress bar icon next to it.

Las opciones por defecto en “Customize” son buenas pero debemos revisarlo por si cualquier cambio del instalador:



Después nos pedirán que aceptemos los términos de licencia. Aceptamos y pulsamos: next

En la parte que seleccionamos la ubicación del IDE, es buena la opción por defecto (el home del usuario) y tenemos la opción de seleccionar si queremos el jdk que ya tiene instalado el sistema o que se descargue un nuevo jdk (también en el home del usuario)

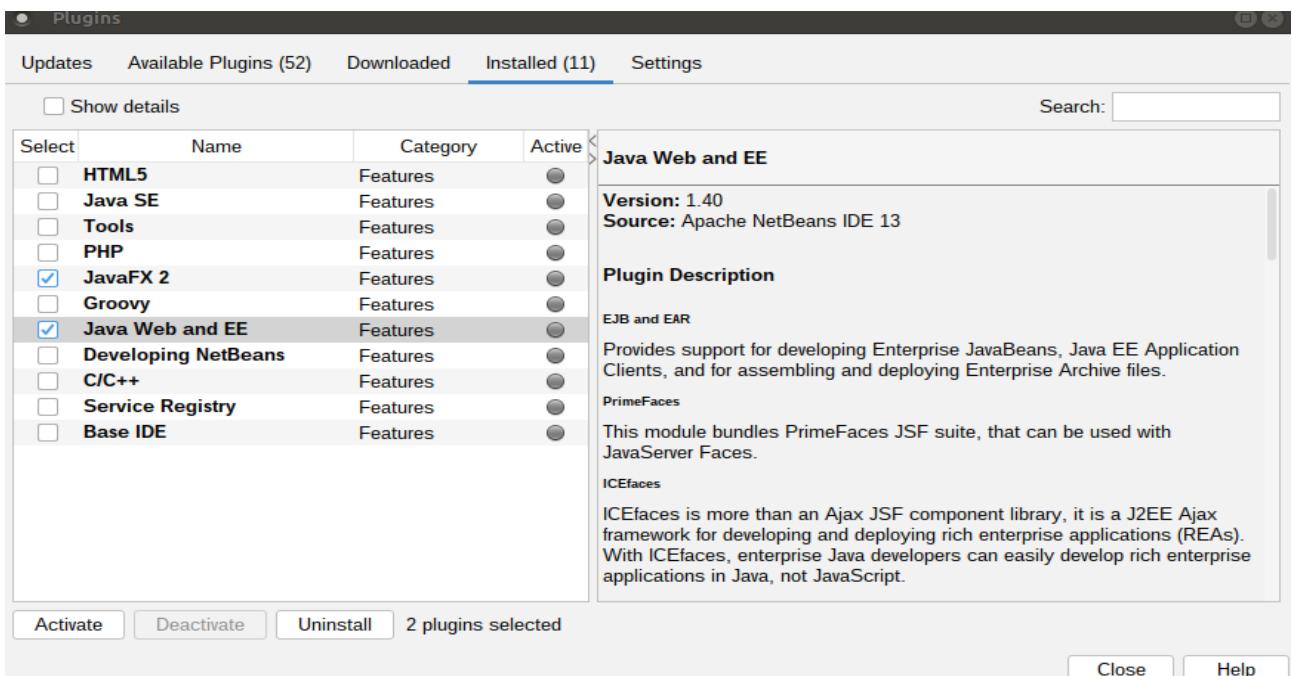


Le damos las veces que sean necesarias a: “next” y que nos haga la instalación

Al arrancar el IDE vamos a dejarlo ya preparado para que esté activo tanto para javaFX como para Web (todo con Maven)

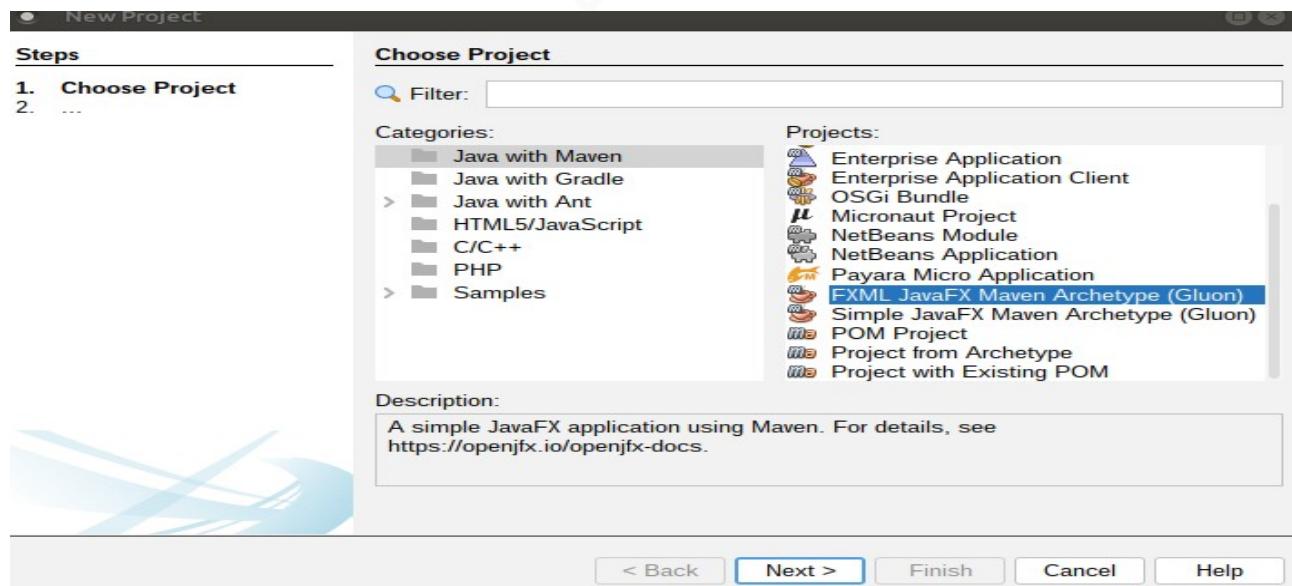
En tools → plugins – (en la pestaña installed) → hacemos check en JavaFX 2 y Java Web and EE

y pulsamos en: "Activate"



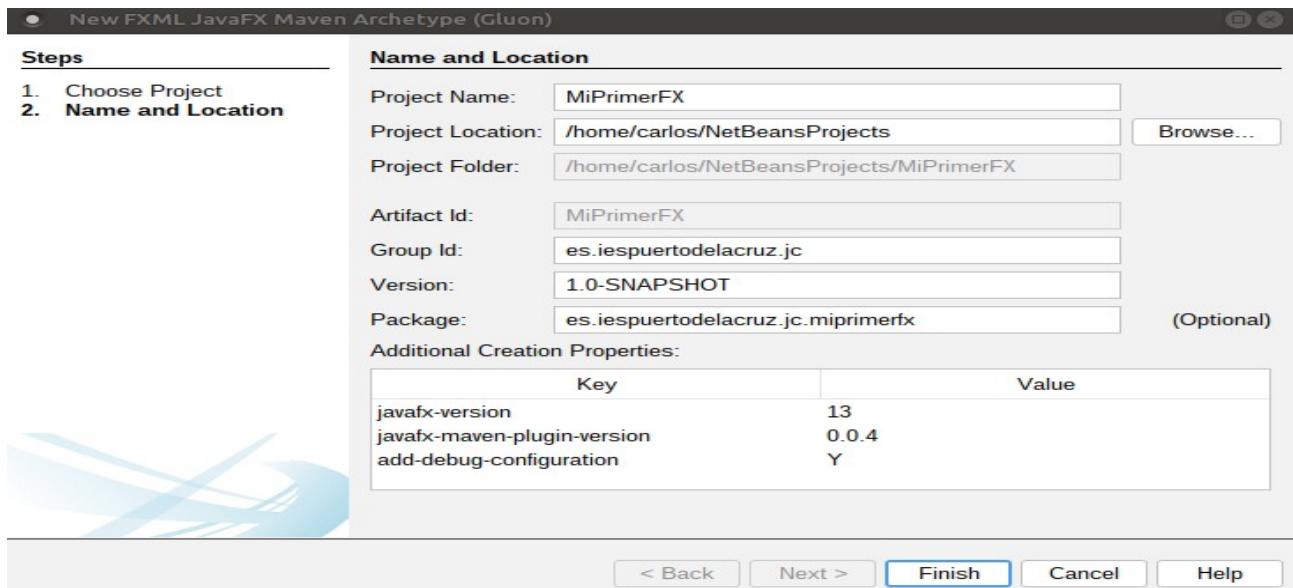
Vamos ahora a crear un proyecto Maven JavaFx:

File → New Project → Java with Maven → FXML JavaFX Maven Archetype (Gluon)



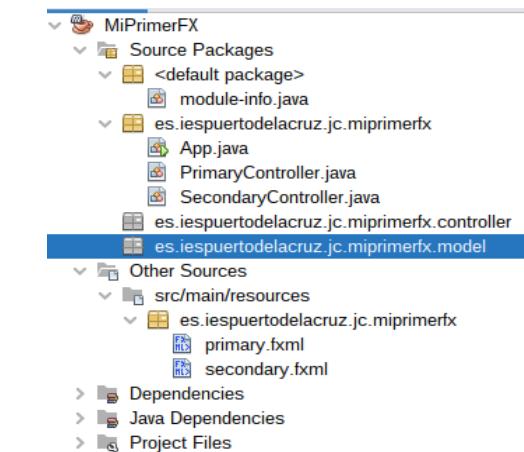
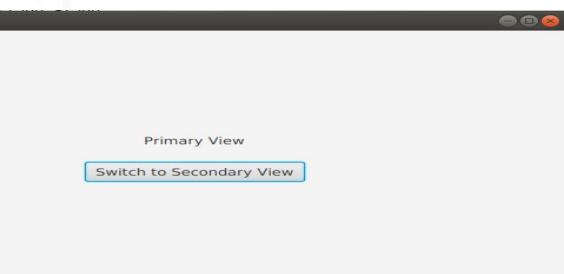
Nos acostumbraremos, en los proyectos Maven, a poner en Group id la referencia a todo nuestro grupo de proyectos: El nombre cualificado (estilo DNS) de nuestra empresa y luego el subpaquete que nos corresponde a nosotros (en este caso aparece jc por: Juan Carlos)

Y ponemos el nombre del proyecto. En este caso se ha llamado: MiPrimerFX:



El proyecto generado ya es ejecutable (tiene dos FXML para saltar de una ventana a otra)

Vamos a hacer la separación de paquetes. En este caso, la estructura de ficheros: .fxml y .css en “resources” es perfectamente válida y no la vamos a cambiar. Pero sí que crearemos el paquete controller y el model



Al llevarnos los “Controller” a su paquete nos cantará error el IDE. Tenemos que hacer un par de cambios:

- En el fxml (realmente solo necesitamos primary.fxml y PrimaryController.java si se quiere se puede borrar lo demás) Tenemos que apuntar a la nueva ubicación: **“es.iespuertodelacruz.jc.miprimerfx.controller.PrimaryController”**

```
xmlns:fx="http://javafx.com/fxml/1" fx:controller="es.iespuertodelacruz.jc.miprimerfx.controller.PrimaryController">
```

Veamos ese trozo de código en contexto:



```
<?xml version='1.0' encoding='UTF-8'?>
<VBox alignment='CENTER' spacing='20.0' xmlns='http://javafx.com/javafx/8.0.171' xmlns:fx='http://javafx.com/fxml/1' fx:controller='es.iespuertodelacruz.jc.controller.miprimerfx.PrimaryController'>
    <children>
        <Label text='Primary View' />
        <button fx:id='primaryButton' text='Switch to Secondary View' onAction='#switchToSecondary' />
    </children>
    <padding>
        <Insets bottom='20.0' left='20.0' right='20.0' top='20.0' />
    </padding>
</VBox>
```

La queja que aparece en el IDE en el fichero PrimaryController.java es porque el método: App.setRoot() no es público. Lo ponemos a público en el fichero: App.java:

```
public class App extends Application {

    private static Scene scene;

    @Override
    public void start(Stage stage) throws IOException {
        scene = new Scene(loadFXML("primary"), 640, 480);
        stage.setScene(scene);
        stage.show();
    }

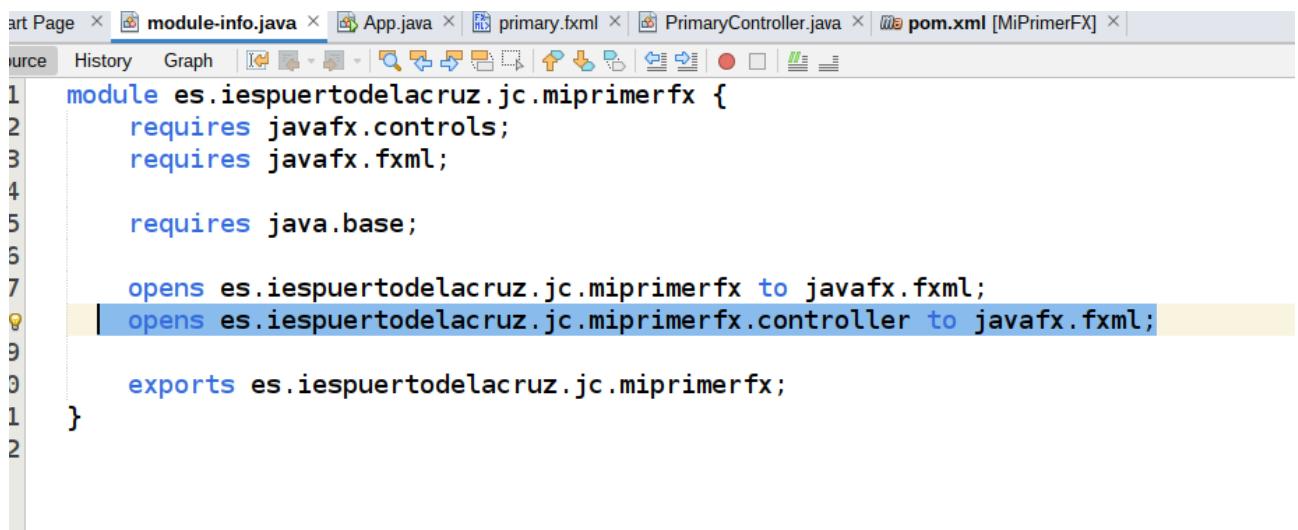
    public static void setRoot(String fxml) throws IOException {
        scene.setRoot(loadFXML(fxml));
    }

    private static Parent loadFXML(String fxml) throws IOException {
        FXMLLoader fxmlLoader = new FXMLLoader(App.class.getResource(fxml + ".fxml"));
        return fxmlLoader.load();
    }

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

Para que la aplicación no tenga problemas con la separación de paquetes tenemos que retocar el fichero: module-info.java

Ese fichero es el que se encarga de cargar módulos y darnos “seguridad” respecto a que módulos tienen acceso a qué cosas. De esa forma nos queda una aplicación mejor particionada y no estamos cargando módulos innecesarios (permitiendo hacer aplicaciones más ligeras)



```
1 module es.iespuertodelacruz.jc.miprimerfx {
2     requires javafx.controls;
3     requires javafx.fxml;
4
5     requires java.base;
6
7     opens es.iespuertodelacruz.jc.miprimerfx to javafx.fxml;
8     opens es.iespuertodelacruz.jc.miprimerfx.controller to javafx.fxml;
9
10    exports es.iespuertodelacruz.jc.miprimerfx;
11 }
12
```

Observar que se le indica que el módulo javafx.fxml tiene acceso al paquete controller

Lo siguiente es innecesario (únicamente para usar controles que están en otro lado ahora mismo como es el htmleditor) La dependencia Maven para javafx-web es:

```
<!-- https://mvnrepository.com/artifact/org.openjfx/javafx-web -->
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-web</artifactId>
    <version>15</version>
</dependency>
```

y luego nos quedaría el module-info.java así:

```
module es.iespuertodelacruz.jc.miprimerfx {
    requires javafx.controls;
    requires javafx.fxml;
    requires javafx.web;
    requires java.base;

    opens es.iespuertodelacruz.jc.miprimerfx to javafx.fxml;
    opens es.iespuertodelacruz.jc.miprimerfx.controller to javafx.fxml;

    exports es.iespuertodelacruz.jc.miprimerfx;
}
```

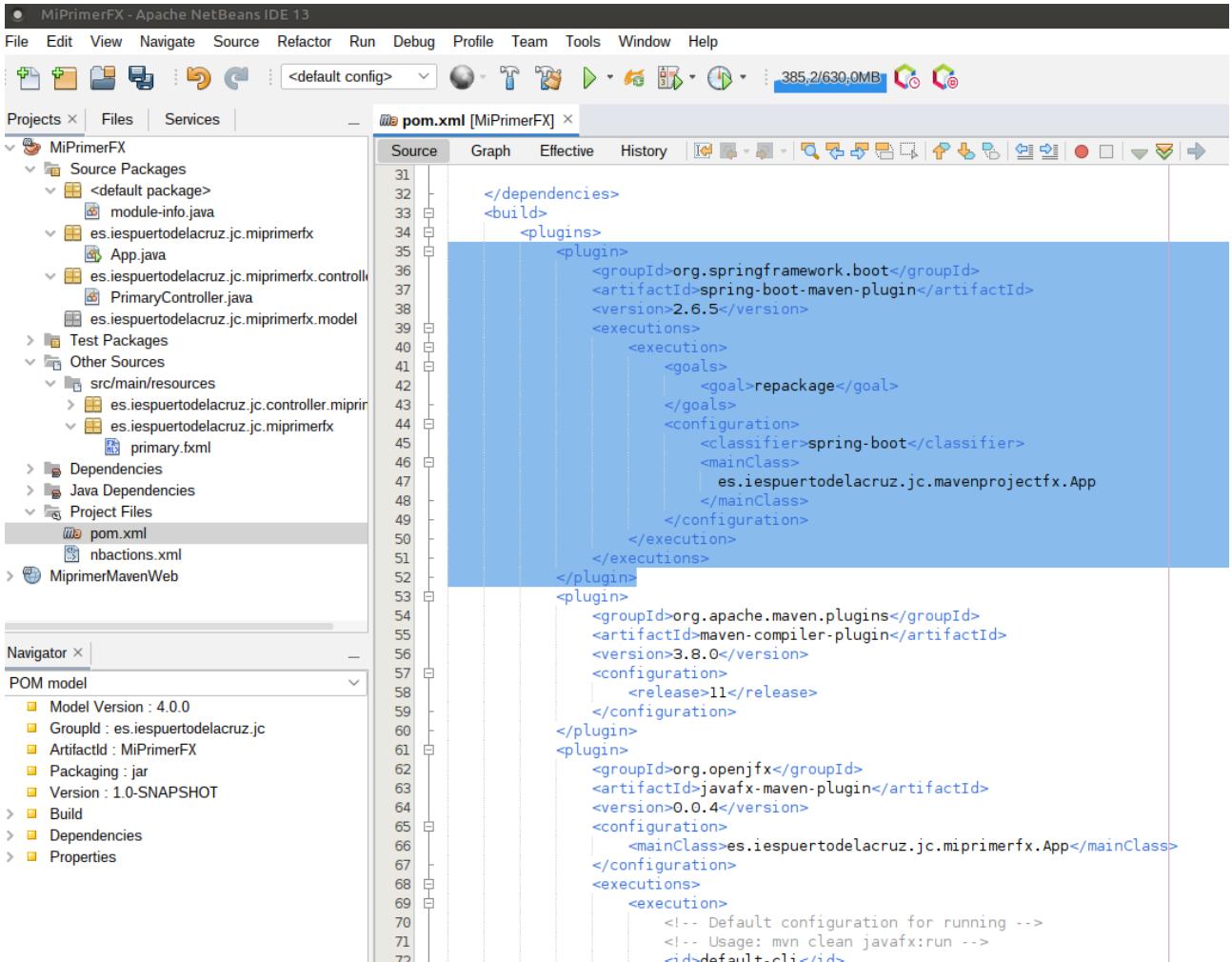
Crear el jar ejecutable

Agregamos el siguiente plugin al pom.xml:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>2.6.5</version>
    <executions>
        <execution>
            <goals>
                <goal>repackage</goal>
            </goals>
            <configuration>
                <classifier>spring-boot</classifier>
                <mainClass>
                    es.iespuertodelacruz.jc.mavenprojectfx.App
                </mainClass>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Se ha marcado en amarillo la línea que tenemos que personalizar (poner el nombre de la clase que comienza/arranca el proyecto JavaFX)

Veamos en contexto el plugin en el pom.xml:



```
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>2.6.5</version>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                    <configuration>
                        <classifier>spring-boot</classifier>
                        <mainClass>es.iespuertodelacruz.jc.mavenprojectfx.App</mainClass>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.0</version>
            <configuration>
                <release>11</release>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-maven-plugin</artifactId>
            <version>0.0.4</version>
            <configuration>
                <mainClass>es.iespuertodelacruz.jc.miprimerfx.App</mainClass>
            </configuration>
            <executions>
                <execution>
                    <!-- Default configuration for running -->
                    <!-- Usage: mvn clean javafx:run -->
                    <id>default-cli</id>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Ahora al hacer el clean and build (shift + f11) vemos en consola la ruta donde ha alojado el jar. En este caso ha sido:

```
Installing          /home/carlos/NetBeansProjects/MiPrimerFX/target/MiPrimerFX-1.0-SNAPSHOT-spring-boot.jar      to
/home/carlos/.m2/repository/es/iespuertodelacruz/jc/MiPrimerFX/1.0-SNAPSHOT/MiPrimerFX-1.0-SNAPSHOT-spring-boot.jar
```

Vamos a esa ubicación y cogemos el jar. Para ejecutar desde cualquier lado realizar un lanzador que ejecute:

```
java -jar NombreDelJarConRutaCompleta.jar
```

Anexo Nomenclatura de controles Swing

Nomenclatura de Swing Controls

| Control | Prefijo |
|----------------|----------------|
| JButton | btn |
| JButtonGroup | btg |
| JCheckBox | cbx |
| JComboBox | cmb |
| JLabel | lbl |
| JList | lst |
| JPasswordField | pwd |
| JProgressBar | pgb |
| JScrollBar | scb |
| JTable | tbl |
| JTextArea | txa |
| JTextField | txt |
| JTextPane | txp |
| JTree | jt |
| JDateChooser | jdc |
| JCalendar | jcl |
| JRadioButton | jrb |

3. Nomenclatura de Swing Menus

| Menu | Prefijo |
|-------------|----------------|
|-------------|----------------|

JMenu mnu

JMenuBar mnb

JMenuItem mni

4. Nomenclatura de Swing Windows

| Window | Prefijo |
|---------------|----------------|
|---------------|----------------|

| | |
|---------------|-----|
| JColorChooser | cch |
|---------------|-----|

| | |
|---------|-----|
| JDialog | dlg |
|---------|-----|

| | |
|--------------|-----|
| JFileChooser | jfc |
|--------------|-----|

| | |
|--------|-----|
| JFrame | frm |
|--------|-----|

| | |
|-------------|-----|
| JOptionPane | opt |
|-------------|-----|

5. Otros

| Window | Prefijo |
|---------------|----------------|
|---------------|----------------|

| | |
|-------------------|-----|
| DefaultTableModel | dtm |
|-------------------|-----|

| | |
|---------|-----|
| JDialog | dlg |
|---------|-----|

| | |
|--------------|-----|
| JFileChooser | jfc |
|--------------|-----|

| | |
|--------|-----|
| JFrame | frm |
|--------|-----|

| | |
|-------------|-----|
| JOptionPane | opt |
|-------------|-----|