

Taller JavaFX

CEP Almería

Índice

1. ¿Qué es JavaFX?.....	2
2. Instalación.....	3
2.1. Instalación de JDK.....	3
2.2. Instalación de Librerías de JavaFX.....	3
2.3. Instalación de Maven.....	4
2.4. Instalamos Scene Builder.....	5
2.5. Configuración de variables de entorno.....	5
2.6. Instalación de Eclipse.....	8
2.6.1. Plugin para Java FX en eclipse: E(fx)clipse.....	8
2.7. Configuración de Eclipse.....	10
3. Nuestro primer programa.....	11
3.1. Agregar Librerías de Java FX.....	13
3.2. Crear un módulo.....	17
3.3. Clases Principales.....	19
3.3.1. Árbol de una aplicación.....	20
3.4. Ciclo de vida de una aplicación.....	22
4. Layouts.....	26
4.1. Hbox.....	26
4.2. FXML y separación vista/controlador.....	28
4.3. VBox.....	33
4.4. BorderPane.....	36
4.5. FlowPane.....	39
4.6. GridPane.....	42
4.5.1. De manera programática.....	43
4.5.2. Separando la vista en FXML.....	45
4.6. Más Layouts.....	46
5. Eventos y Listeners.....	46
5.1. Creando eventos SIN controlador.....	47
5.2. Creando eventos CON controlador (MVC Friendly).....	49
6. Threads.....	51
7. Aplicando estilos con CSS.....	56
7.1. Programáticamente con estilos en línea.....	57
7.2. Programáticamente con una hoja de Estilos.....	59
7.3. Estilos con FXML con SceneBuilder.....	62
8. Crear un proyecto Maven.....	64
9. Demo de aplicación – CalculadoraJavaFX.....	65

1. ¿Qué es JavaFX?

JavaFX es un framework para la creación de GUIs (Graphical User Interfaces) en Java, y es considerado el sucesor de Swing. Se trata de una API para el diseño de interfaces capaces de correr en casi cualquier dispositivo con soporte Java.

En Java 8 JavaFX pasó a formar parte de JDK, con lo cual no necesitábamos usar ninguna librería externa. Sin embargo, desde Java 11, se sacó del JDK, ofreciéndose como un módulo independiente, entre otras razones por la tendencia de Oracle a dejar en el JDK sólo los componentes core. Ahora, los módulos de JavaFX están disponibles bien como artefactos maven para ser usados por Maven/Gradle o como un SDK standalone que debemos incluir en nuestro proyecto.

La web oficial del proyecto es <https://openjfx.io/>. Aquí podemos descargarnos el SDK, encontrar los [Javadocs](#), [Getting Starteds](#), ... así como gran cantidad de ejemplos.

Entre las mejoras que incorpora respecto a Swing y AWT tenemos:

- Posibilidad de diseñar la vista usando ficheros XML → XML. Por tanto se produce una **separación efectiva de la vista y el resto de la aplicación**.
- Posibilidad de crear aplicaciones siguiendo un patrón MVC.
- Manejo eficiente de hilos de ejecución.
- Facilidad para la creación de gráficos (Charts)
- Facilidad para la creación de gráficos y formas.
- ...

¿Qué cubrimos en este manual?

Este manual pretende ser una introducción básica a JavaFX. El objetivo es que el lector se familiarice con lo básico y adquiera los fundamentos de JavaFX que le permite poder seguir por su cuenta.

En concreto veremos:

1. **Cómo instalar el entorno de desarrollo completo:** Eclipse, JDK, Maven, SceneBuilder, SDK de JavaFX, ...
2. Comprensión de las clases básicas en una aplicación **JavaFX**, así como del ciclo de vida de una aplicación
3. Principales layouts para organizar los contenidos.
4. Cómo asociar eventos a controles: distintos tipos de eventos, varias formas de implementarlos, ...

5. Como manejar Threads con JavaFX para evitar ocupar el hilo de ejecución principal con tareas pesadas.
6. Como aplicar estilo CSS a una aplicación

2. Instalación

Instalaremos todo el software necesario en [C:/desarrollo](#).

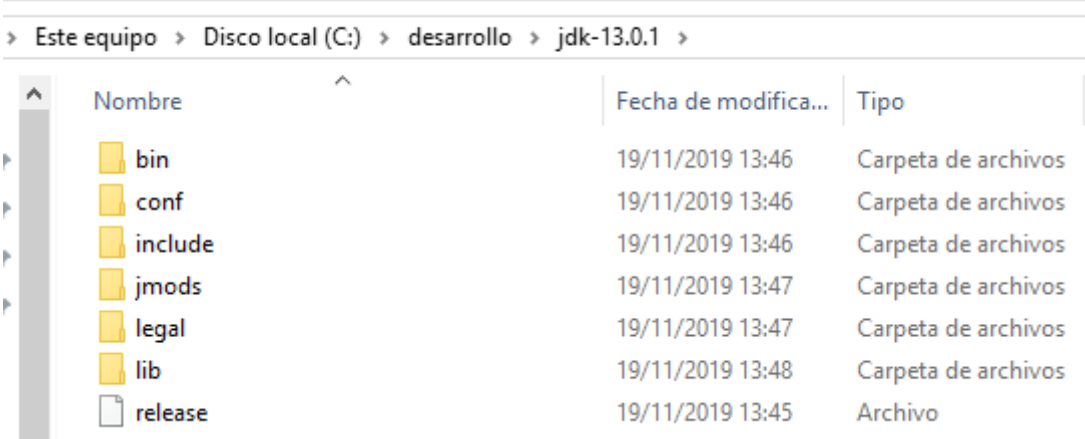
2.1. Instalación de JDK

En esta ocasión, vamos a usar **Open JDK**, disponible en la [Web de Open JDK](#). (<https://openjdk.java.net/install/index.html>). Descargaremos preferentemente la opción en .zip, ya que es más controlable.

En el momento de escribir este manual, la última versión disponible es la **13.0.1**.

Si quisiéramos, podríamos haber instalado el JDK de Oracle sin ningún problema.

Procuramos que quede en **C:/desarrollo\jdk-13.0.1**



Nombre	Fecha de modifica...	Tipo
bin	19/11/2019 13:46	Carpeta de archivos
conf	19/11/2019 13:46	Carpeta de archivos
include	19/11/2019 13:46	Carpeta de archivos
jmods	19/11/2019 13:47	Carpeta de archivos
legal	19/11/2019 13:47	Carpeta de archivos
lib	19/11/2019 13:48	Carpeta de archivos
release	19/11/2019 13:45	Archivo

Figura 1: Instalación de Open JDK

2.2. Instalación de Librerías de JavaFX

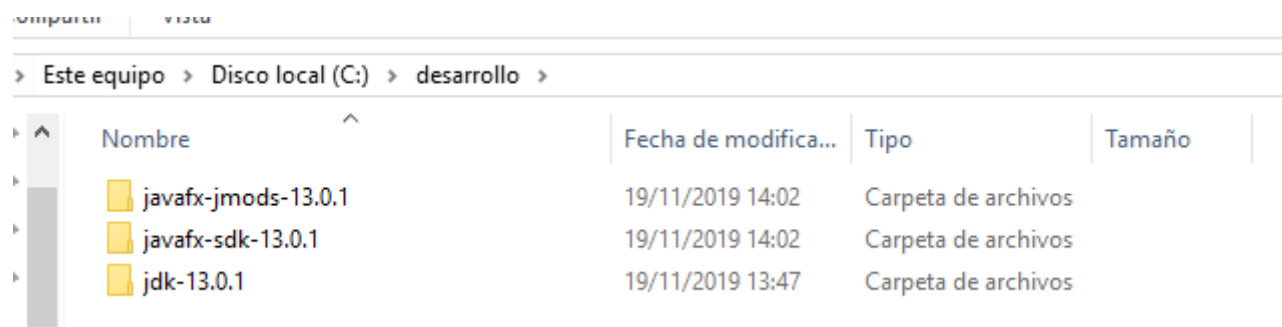
Como sabes, Java FX ya no forma parte del SDK de Java. Así que tenemos 2 opciones: bien nos descargamos las librerías necesarias y las incluimos manualmente en nuestros proyectos javafx, o creamos un proyecto Maven e incluimos las dependencias. Probaremos las 2 opciones, pero por lo pronto, descargamos las librerías necesarias de <https://gluonhq.com/products/javafx/>

En el momento de realizar este tutorial, la versión disponible es la 13:

Product	Version	Platform	Download
JavaFX Windows SDK	13.0.1	Windows	Download [SHA256]
JavaFX Windows jmods	13.0.1	Windows	Download [SHA256]

Figura 2: SDK de Java FX

Descargamos tanto el SDK como los jmods. Recuerda descomprimir ambos en [C:/Desarrollo](#):



Nombre	Fecha de modifica...	Tipo	Tamaño
javafx-jmods-13.0.1	19/11/2019 14:02	Carpeta de archivos	
javafx-sdk-13.0.1	19/11/2019 14:02	Carpeta de archivos	
jdk-13.0.1	19/11/2019 13:47	Carpeta de archivos	

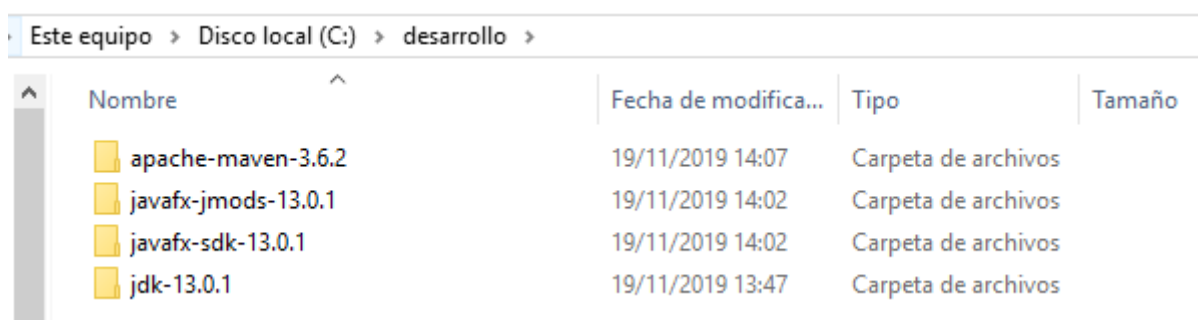
Figura 3: SDK de JavaFX y jmods ubicados en C:/desarrollo

2.3. Instalación de Maven

Vamos a hacer algún ejemplo usando Maven como gestor de dependencias. En lugar de usar el integrado en eclipse, vamos a usar uno descargado de su Web oficial:

<https://maven.apache.org/download.cgi>

Igualmente, nos descargamos el .zip (a fecha de escribir este manual es la versión 3.6.2), lo descomprimos y lo movemos a [C:/desarrollo](#):



Nombre	Fecha de modifica...	Tipo	Tamaño
apache-maven-3.6.2	19/11/2019 14:07	Carpeta de archivos	
javafx-jmods-13.0.1	19/11/2019 14:02	Carpeta de archivos	
javafx-sdk-13.0.1	19/11/2019 14:02	Carpeta de archivos	
jdk-13.0.1	19/11/2019 13:47	Carpeta de archivos	

Figura 4: Ubicación de Apache Maven

2.4. Instalamos Scene Builder

Recordemos que una ventaja de **JavaFX** respecto a **Swing** es que permite una separación efectiva entre la vista y el resto de la aplicación a través de los **ficheros FXML** que permiten, por medio de una gramática XML, describir el aspecto y los componentes de una ventana.

Editar estos ficheros en modo texto no es fácil: hay que conocer en profundidad el espacio de nombres de fxml. **SceneBuilder** nos permite evitar esa complejidad, permitiéndonos diseñar las interfaces visualmente, y generando el código XML necesario por nosotros.

Podemos descargar SceneBuilder de <https://gluonhq.com/products/scene-builder/>. Por desgracia, el instalador (de la versión 11.0.0) no te pregunta donde lo quieres instalar, y lo hace directamente en:

C:\Program Files\SceneBuilder

2.5. Configuración de variables de entorno

Las usaremos para indicar las rutas de nuestros runtimes.

JAVA_HOME

Para indicar a nivel global, nuestro directorio de instalación de Java. Recuerda que lo descomprimos en [C:/desarrollo](#).

Nos vamos al *Panel de Control* → *Sistema y Seguridad* → *Sistema*. Y aquí, hacemos clic derecho en ‘Configuración Avanzada del Sistema’:

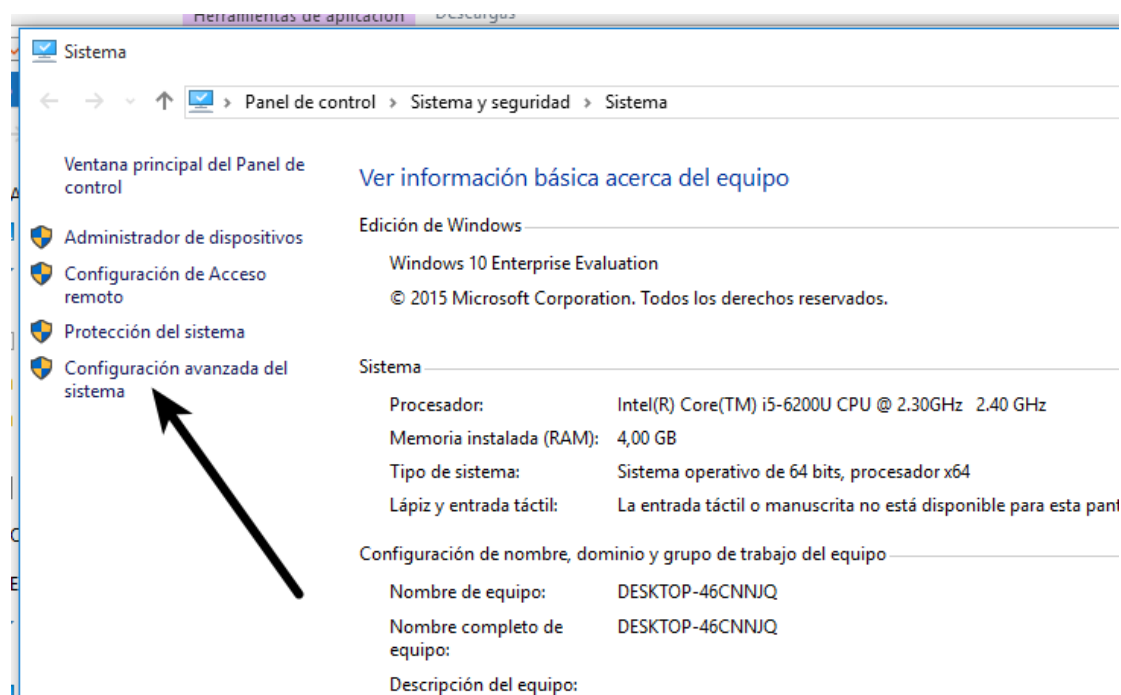


Figura 5: Configuración avanzada del sistema

Tras esto, haz clic en ‘Variables de entorno...’ y aparecerá el diálogo de variables de entorno:

En el apartado ‘Variables de sistema’, pulsa en ‘Nueva...’ y añade la variable de nombre `JAVA_HOME` y de valor, la ruta donde tengas instalado el jdk. En nuestro caso:

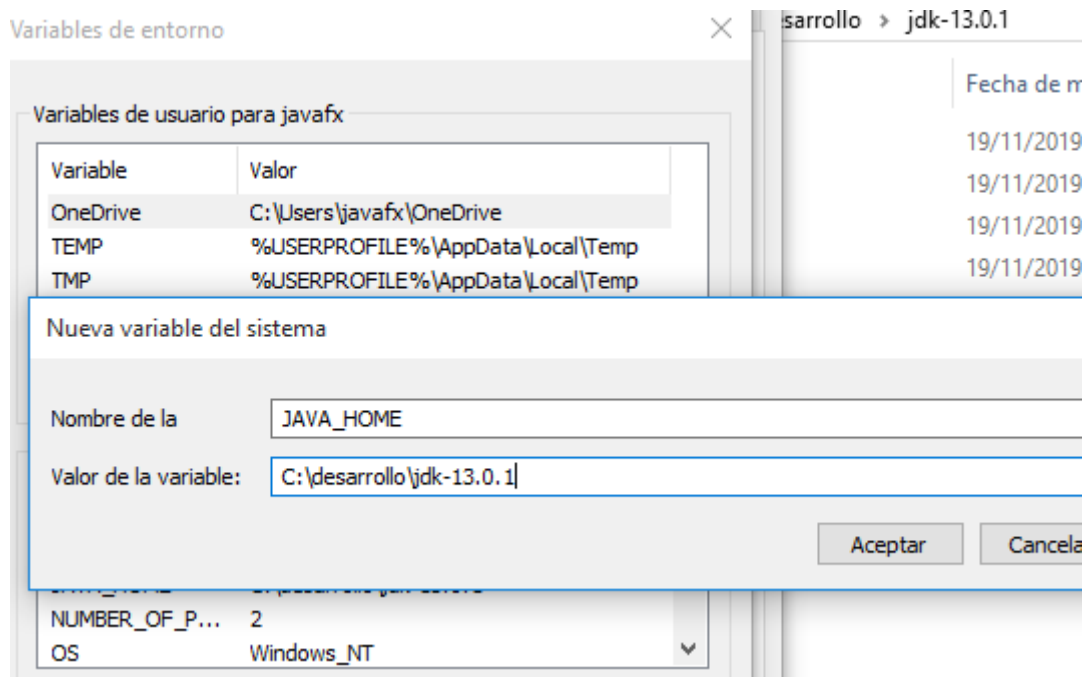


Figura 6: Variable de sistema `JAVA_HOME`

Variable del sistema **Path**

La variable **Path** ya existe, pero vamos a añadirle un nuevo valor. Esta variable contiene rutas del sistema que contienen ejecutables para que puedan ser invocados sin especificar su ruta completa, tal y como haces con comandos como `dir`, `regedit`, `msconfig`, ...

Selecciona la variable **Path** en la lista de variables del sistema, y añade al final del valor existente un punto y coma (;) y la ruta hacia el directorio bin de tu instalación de jdk. En nuestro caso, sería:

`;C:\desarrollo\jdk-13.0.1\bin`

Por tanto, nuestra variable **Path** tiene el valor que tenía, más `;C:\desarrollo\jdk-13.0.1\bin`

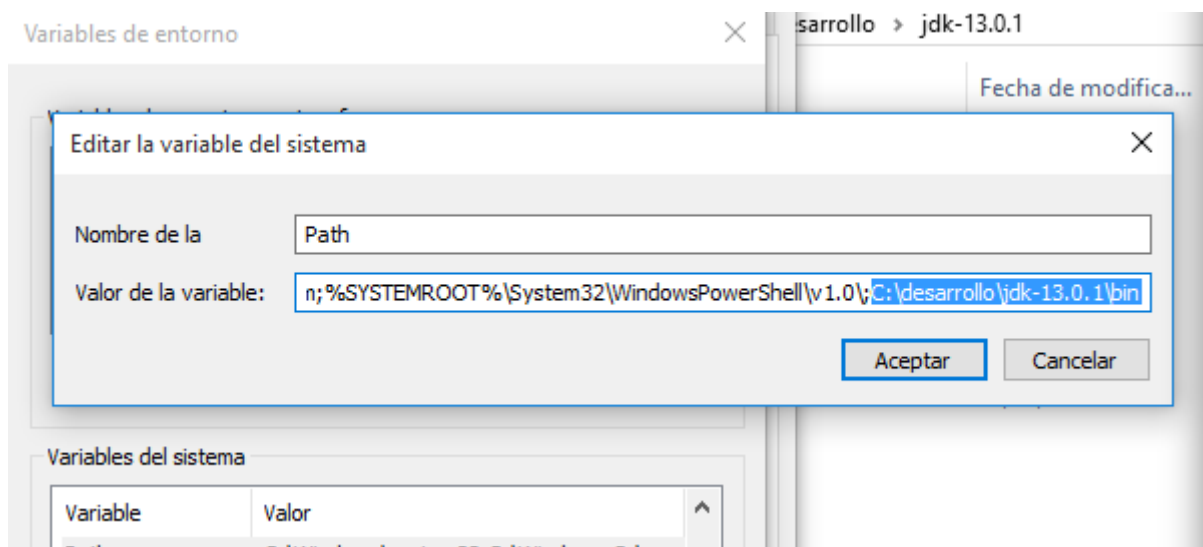


Figura 7: Modificación de la variable Path

M2_HOME

Para invocar maven desde la terminal, y que otros programas puedan invocarlo automáticamente sin necesidad de saber su ruta, debemos crear esta variable de sistema.

Seguimos el mismo procedimiento que para **JAVA_HOME**, pero creamos una con los siguientes datos:

- **Nombre:** M2_HOME
- **Valor:** C:\desarrollo\apache-maven-3.6.2

Sustituye el valor por la ruta de tu instalación de maven.

Tienes que añadir también el directorio **bin** de maven al Path. Para ello, sigue los mismos pasos que el apartado anterior (para añadir el directorio bin de java a la variable Path), pero esta vez, añadiendo a Path:

;C:\desarrollo\apache-maven-3.6.2\bin

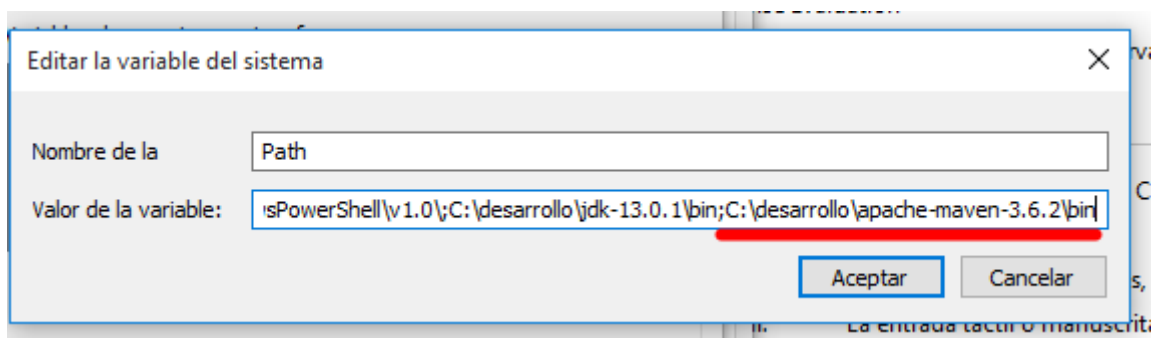
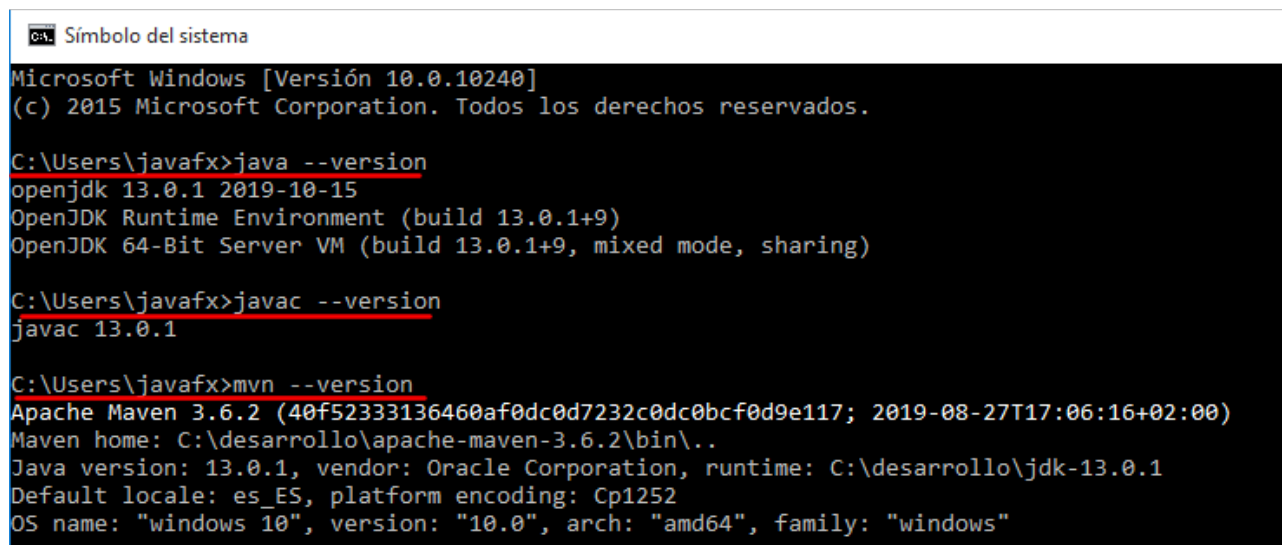


Figura 8: Directorio bin de maven añadido al Path

Para comprobar que todo ha ido bien, abre una consola y ejecuta los siguientes comandos:



```
Microsoft Windows [Versión 10.0.10240]
(c) 2015 Microsoft Corporation. Todos los derechos reservados.

C:\Users\javafx>java --version
openjdk 13.0.1 2019-10-15
OpenJDK Runtime Environment (build 13.0.1+9)
OpenJDK 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)

C:\Users\javafx>javac --version
javac 13.0.1

C:\Users\javafx>mvn --version
Apache Maven 3.6.2 (40f52333136460af0dc0d7232c0dc0bcf0d9e117; 2019-08-27T17:06:16+02:00)
Maven home: C:\desarrollo\apache-maven-3.6.2\bin\..
Java version: 13.0.1, vendor: Oracle Corporation, runtime: C:\desarrollo\jdk-13.0.1
Default locale: es_ES, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Figura 9: ejecución de java, javac y mvn

2.6. Instalación de Eclipse

Por último, instalamos eclipse. La versión escogida en este tutorial es la 2019-09. Puedes descargar eclipse de <https://www.eclipse.org/downloads/>.

Procura instalarlo en <C:\Desarrollo>

En nuestro caso, hemos decidido también crear el workspace de eclipse en <c:/desarrollo/workspace>

2.6.1. Plugin para Java FX en eclipse: E(fx)clipse

Puedes consultar las instrucciones aquí:

[https://wiki.eclipse.org/Efxclipse/Tutorials/AddingE\(fx\)clipse_to_eclipse](https://wiki.eclipse.org/Efxclipse/Tutorials/AddingE(fx)clipse_to_eclipse)

Para instalarlo, haz clic en la barra de herramientas en ‘**Help**’ → ‘**Eclipse Market Place**’:

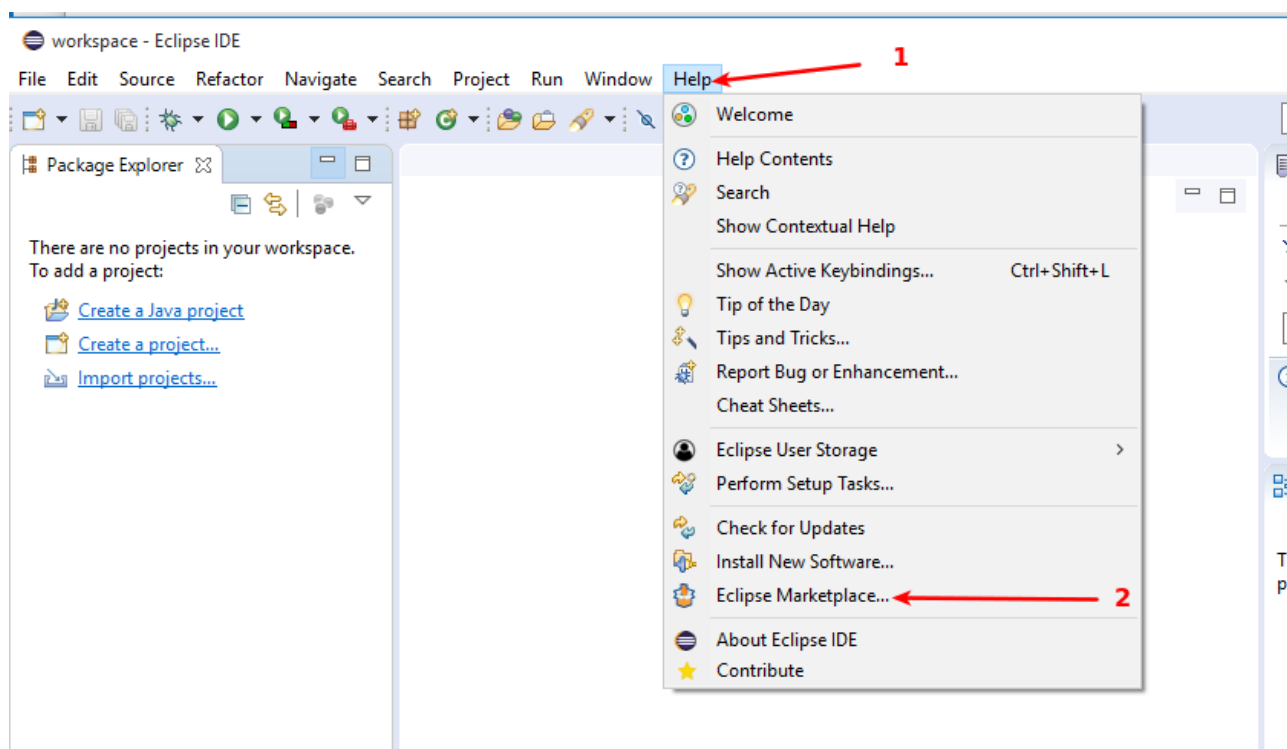


Figura 10: Abrir eclipse Marketplace

Una vez se abra el diálogo del **Marketplace**, escribe en el filtro de búsqueda 'javafx' y pulsa buscar. Cuando salgan los resultados, haz clic en el mostrado a continuación:

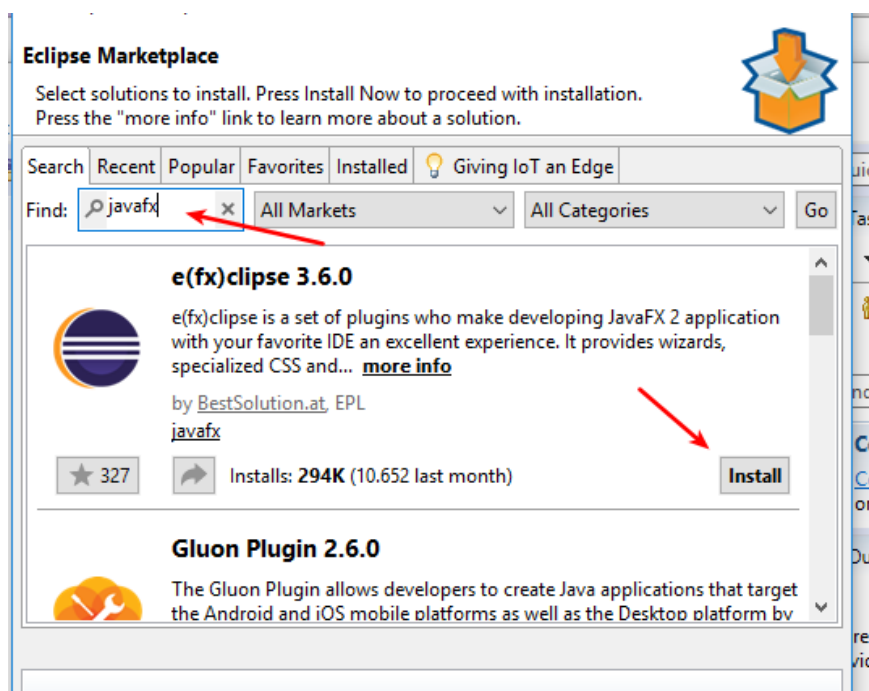


Figura 11: Eclipse Marketplace

2.7. Configuración de Eclipse

Una vez instalado eclipse y el plugin de java fx, nos aseguramos de que nuestros JDK (en el caso de que tengamos más de uno) están bien.

Nos vamos a la barra de herramientas, a **Windows** → **Preferences**, y en el filtro de opciones, escribimos *installed*:

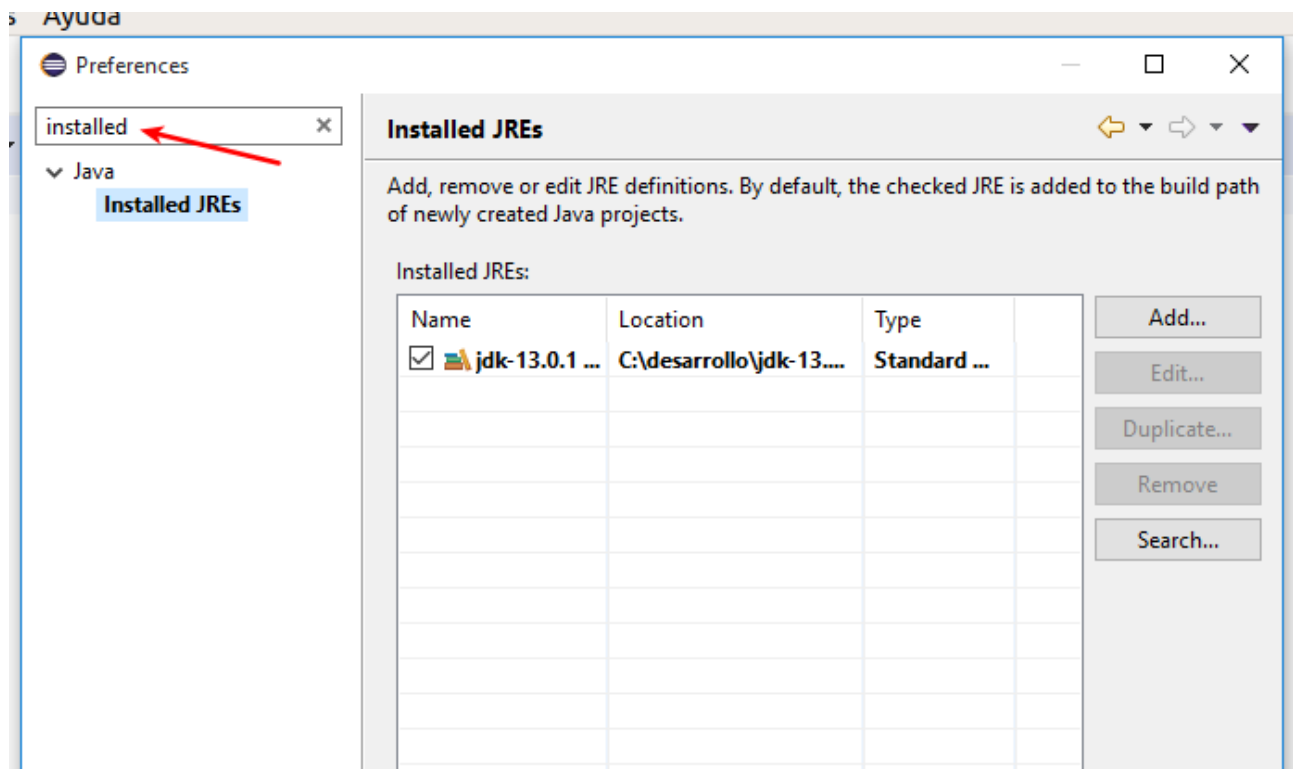


Figura 12: JREs instalados

Nos aseguramos de que tenemos seleccionado el que queremos (Open JDK 13 en nuestro caso). Si queremos probar con otro, como por ejemplo el de oracle, aquí podemos añadirlo.

3. Nuestro primer programa

El código fuente para este programa está en: <https://github.com/pes130/01-HolaMundoFX>

Pulsamos en **File** → **New** → **Other....** Y a continuación, en el diálogo para seleccionar lo que deseamos hacer, escribimos en el filtro javafx:

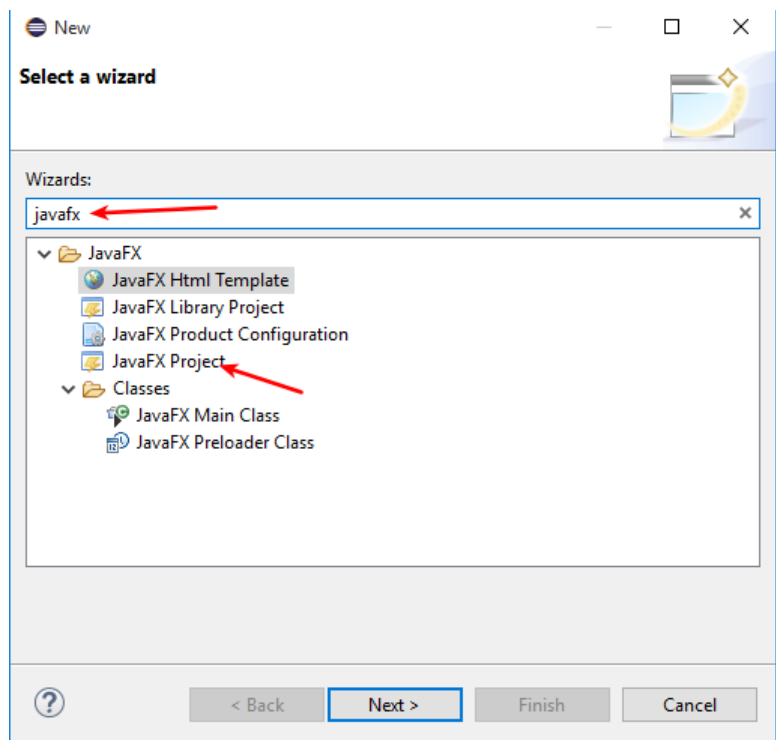


Figura 13: Proyecto JavaFX

Tras esto, selecciona '**JavaFX Project**' y haz clic en **Next**. En el siguiente paso, ponle de nombre al proyecto 01-HolaMundoFX, y selecciona que deseas usar el jdk por defecto:

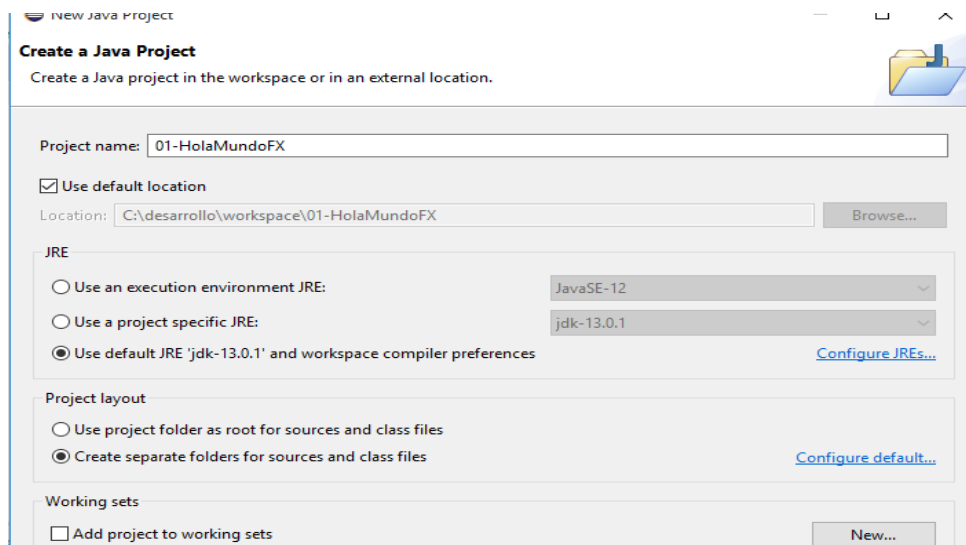
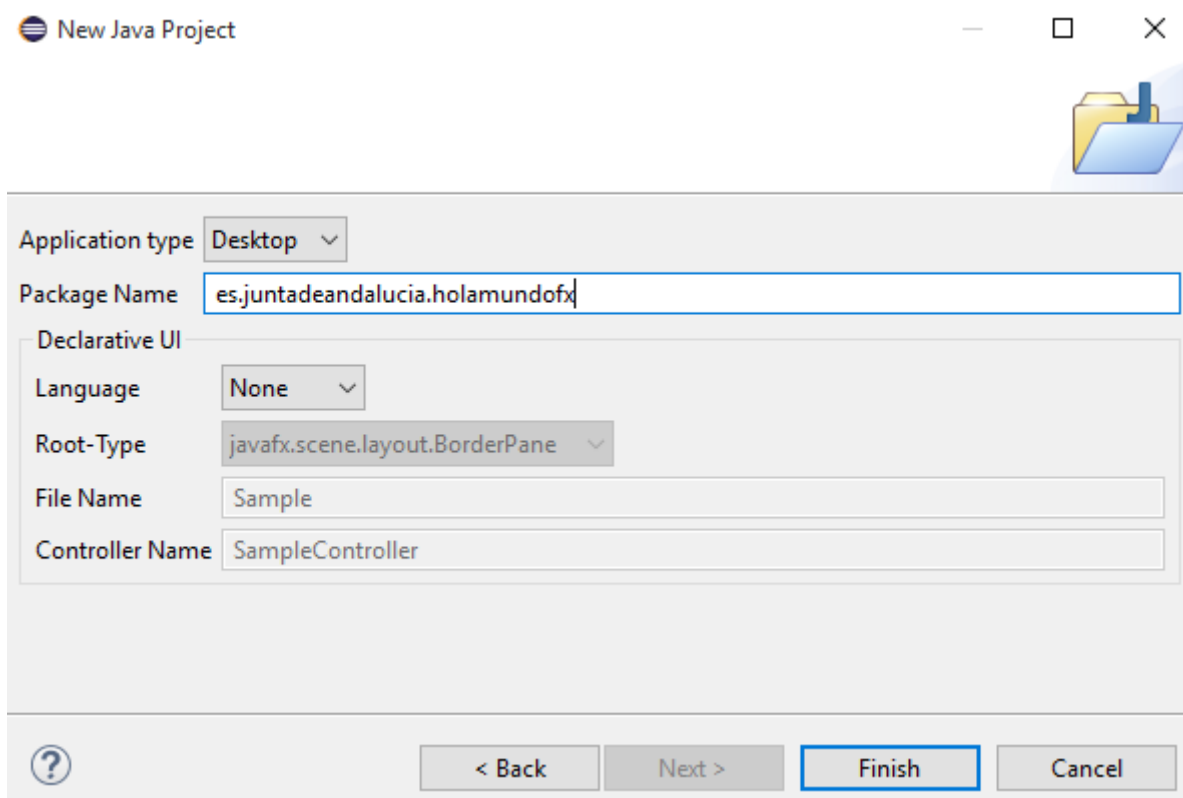


Figura 14: Opciones de nuevo proyecto

Pulsa **Next** dos veces hasta llegar al paso siguiente donde tienes que elegir un nombre para tus paquetes, en nuestro caso **es.juntadeandalucia.holamundofx**:



Pulsa en **Finish** y espera a que se construya el proyecto.

3.1. Agregar Librerías de Java FX

Si te fijas, el proyecto recién creado da muchos errores:

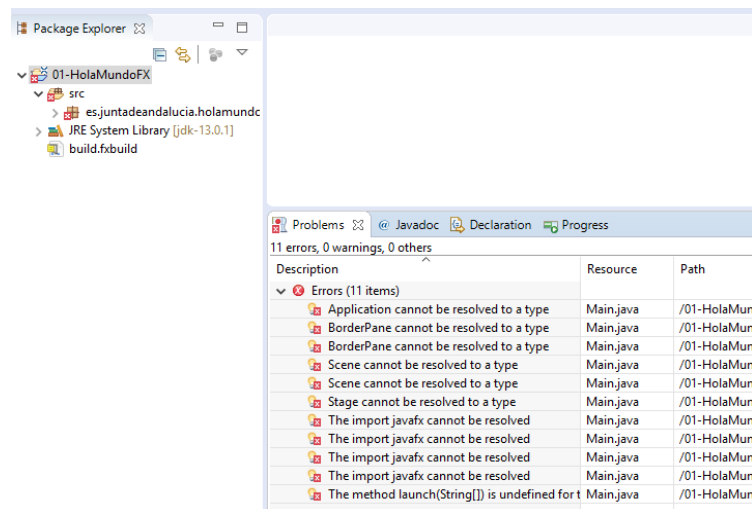


Figura 15: Errores nada más crear proyecto

Es debido a que las clases de java FX no son reconocidas por el compilador. Tenemos que incluirlas en nuestro **build path**. Para ello, hacemos clic derecho sobre nuestro proyecto en el árbol de proyectos de la izquierda, y seleccionamos **'Build Path' → 'Configure Build Path ...'**:

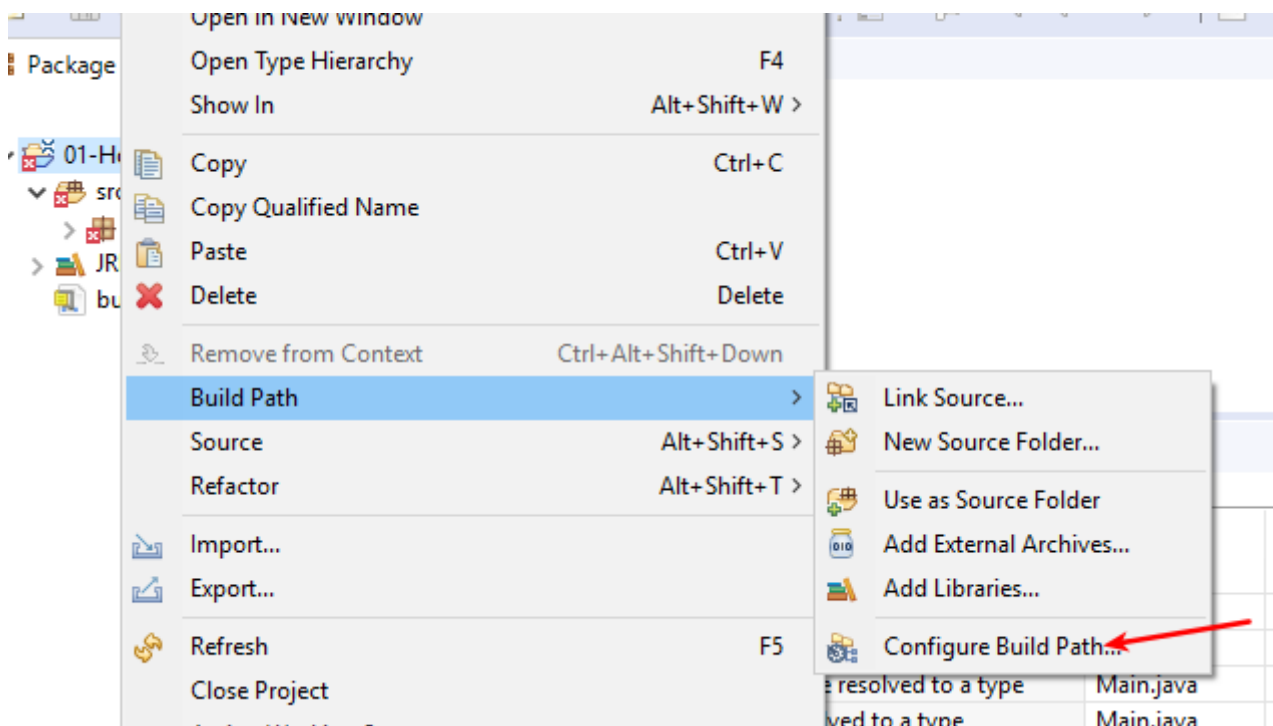


Figura 16: Configuramos el Build Path

Hacemos clic en la pestaña ‘**Libraries**’ y luego en ‘**Modulepath**’:

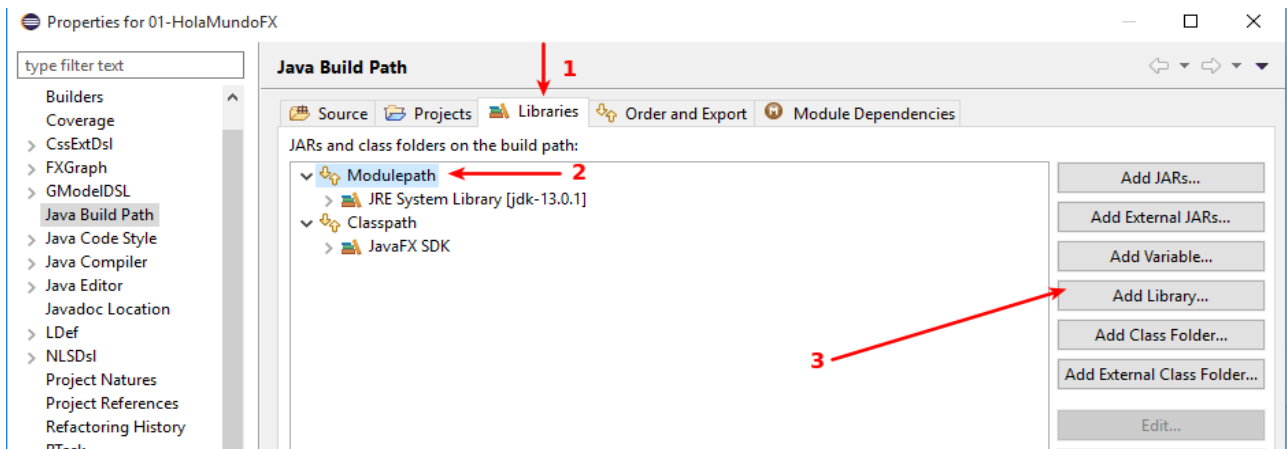


Figura 17: Añadimos una librería I

Seleccionamos ‘User Library’ y le damos a siguiente:

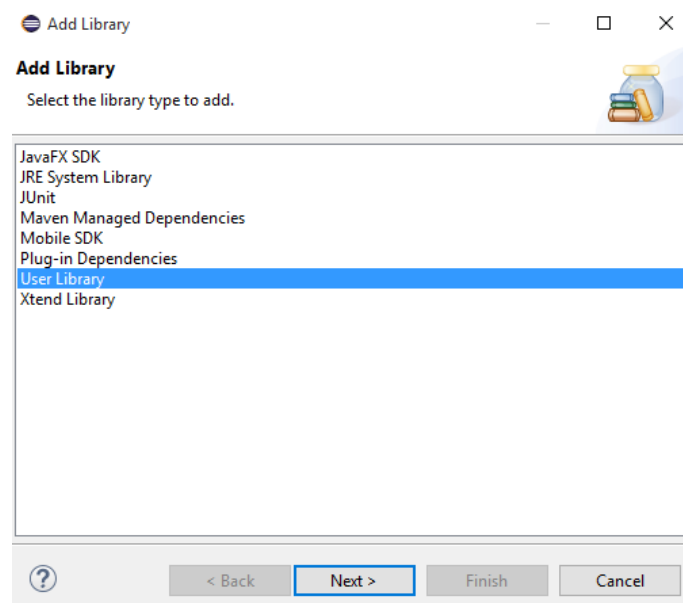


Figura 18: Añadir una Librería II

Pulsamos en el Botón '**User Libraries**', y luego en **Next**:

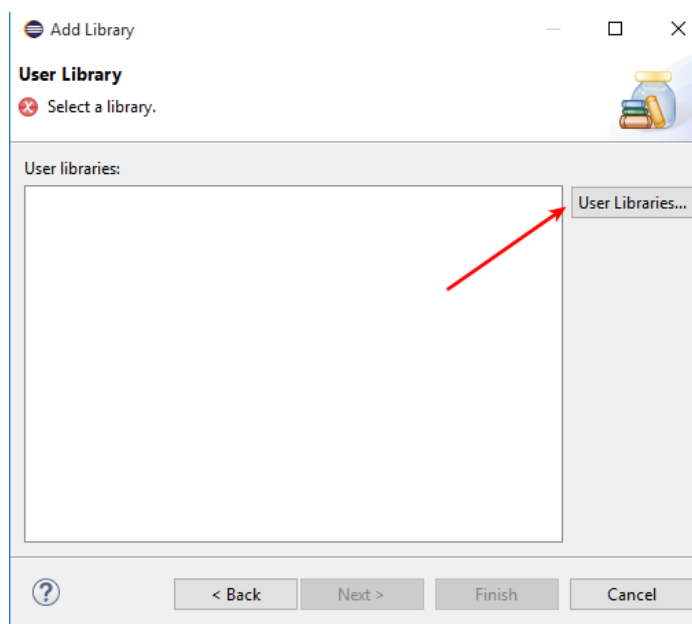


Figura 19: Añadir una librería III

Le damos a **New**, y en el diálogo le decimos de nombre '**javafx13**':

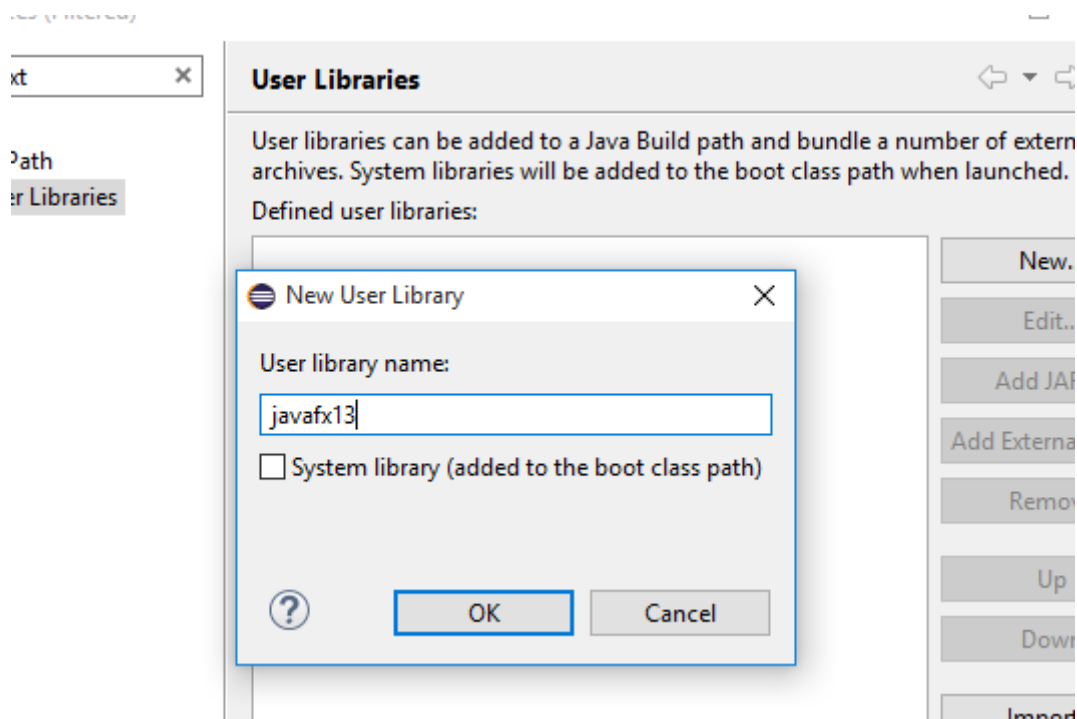


Figura 20: Añadir una librería IV

Pulsa Ok y se cerrará el diálogo. Tras esto, vamos a añadir los jars de javafx a nuestra recién creada librería. Pulsa en el botón 'Add External JARS' y localiza los jars de javafx que deberían estar en C:\desarrollo\javafx-sdk-13.0.1\lib. Añade todos los jars:

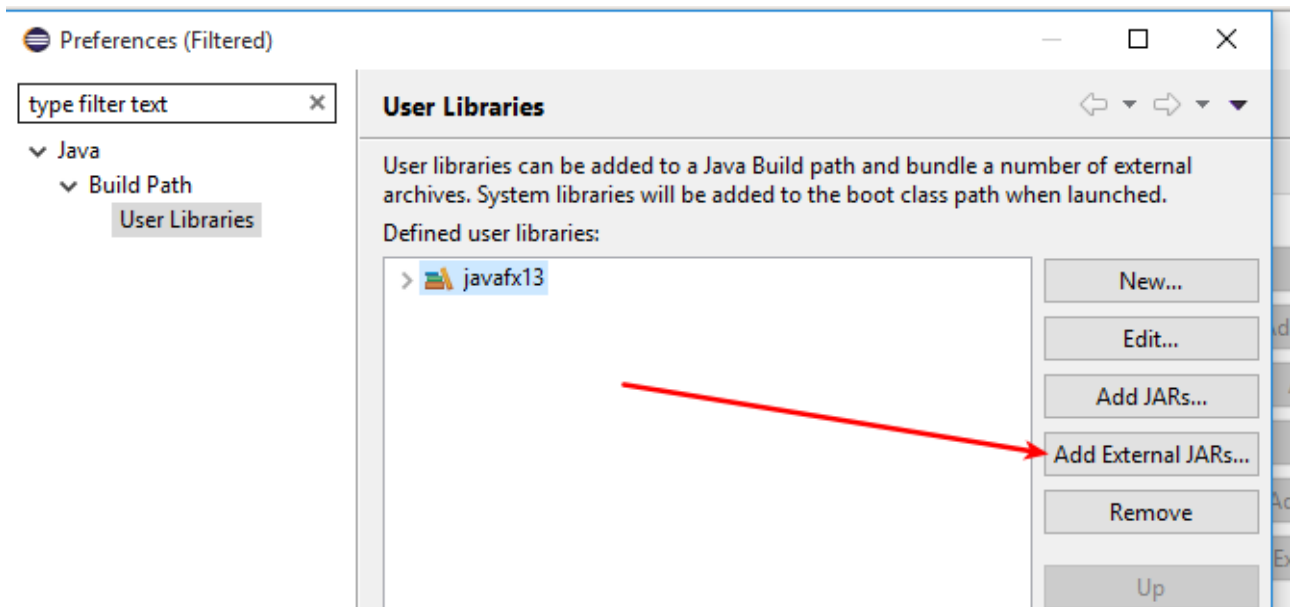


Figura 21: Añadimos los jars de javafx

Tras hacerlo, tu librería debería verse así:

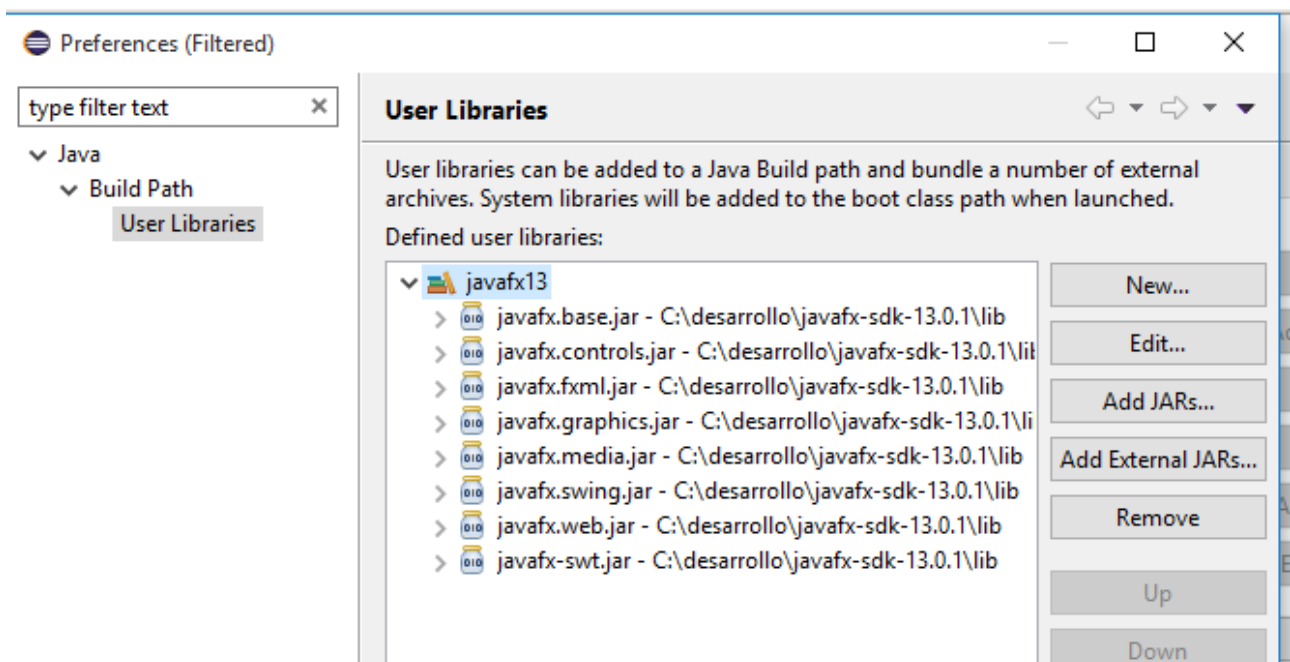


Figura 22: JARS añadidos a nuestra librería

Tras esto, vamos aplicando y cerrando diálogos abiertos, hasta que cerramos todos y volvemos al editor de eclipse. Los errores deberían desaparecer.

3.2. Crear un módulo

Los módulos existen desde Java 9. En ellos indicamos qué paquetes vamos a necesitar y qué vamos a exportar de nuestra propia aplicación para poder ser usado desde fuera. Vamos a crear uno. Para ello, crea un fichero llamado **module-info.java**.

Haz clic derecho en **src** → **New** → **File**:

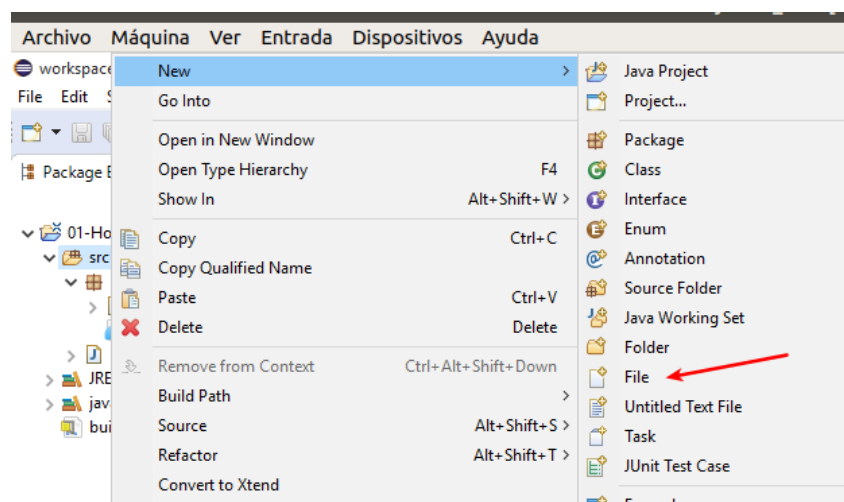


Figura 23: Crear un nuevo fichero

Tras esto, pone de nombre: **module-info.java**:

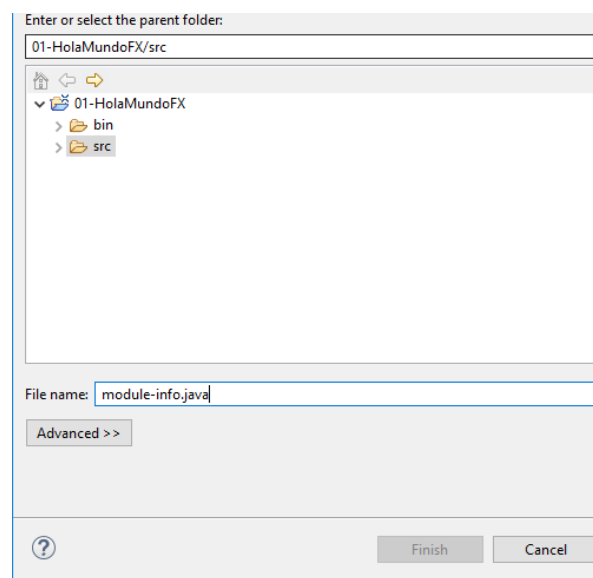


Figura 24: module-info.java

Escribe en él el siguiente contenido:

```
module HolaMundoFX {  
    requires javafx.fxml;  
    requires javafx.controls;  
    requires javafx.graphics;  
    requires javafx.web;  
    requires javafx.base;  
  
    exports es.juntadeandalucia.holamundofx;  
}
```

En el archivo **Main.java**, escribe el siguiente contenido:

```
public class Main extends Application {  
    private Button btn;  
  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        btn = new Button();  
        btn.setText("Click me!");  
        BorderPane pane = new BorderPane();  
        pane.setCenter(btn);  
  
        Scene scene = new Scene(pane, 300, 200);  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

En el siguiente apartado identificaremos las clases principales usadas en el ejemplo.

3.3. Clases Principales

`javafx.application.Application`

La clase principal de nuestra aplicación debe extender la clase `javafx.application.Application`. Además, esta será nuestra clase ejecutable y el punto de partida de nuestra aplicación. Para ello, debemos escribir el método estático **main**, y dentro, invocar al método **launch** de nuestra clase (que es heredado de **Application**):

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

Esta clase `Application` tiene una serie de métodos que podemos sobrescribir y que definen el ciclo de vida de nuestra aplicación, como veremos en el siguiente apartado.

`javafx.stage.Stage`

Una **Stage** es, básicamente, una ventana. Representa el marco donde se cargarán luego los layouts y los controles de una ventana concreta. La clase **Application** por defecto nos crea una, y nos la proporciona como parámetro de entrada en su método **start**.

`javafx.scene.Scene`

Una **escena** o **Scene**, por otro lado, es el conjunto de objetos (controles, layouts, ...) que se pintan en una **Stage** o Ventana. En resumen, una **Stage** es el marco de la ventana, que contiene: una barra de título, botones de maximizar, cerrar y restaurar. Un **Scene** es un objeto asociado a una **Stage**, y cuya función es contener los **Nodos**. Como nodos en terminología JavaFX se definen los **controles** (botones, labels, campos de texto, ...) y **layouts** (elementos para organizar los controles (en tabla, en vertical, en horizontal, ...)).

Podemos hacer el símil con un teatro: el `Stage` es un escenario, donde se desarrollan Escenas (`Scenes`)

3.3.1. Árbol de una aplicación

Una vez vista la aplicación de ejemplo, podemos ver una aplicación, desde el punto de vista jerárquico, del siguiente modo:

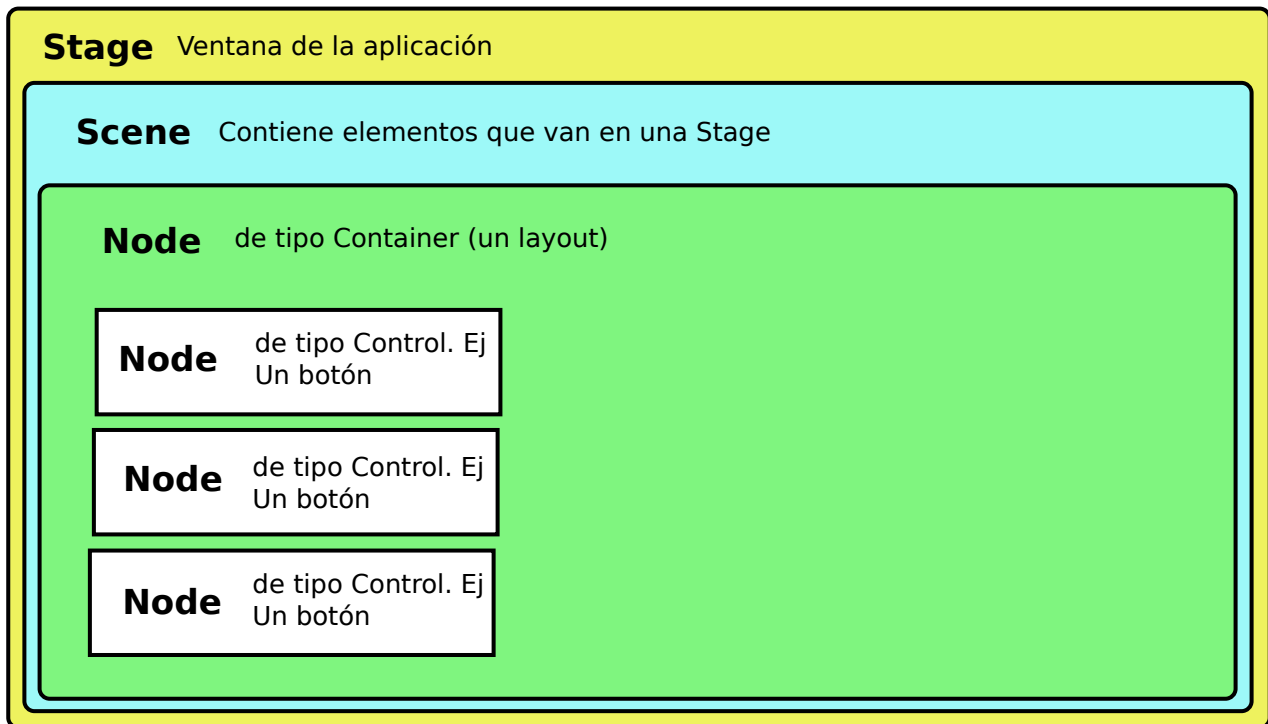


Figura 25: Jerarquía de clases en una aplicación

Veamos esto con un poco más de detalle. A continuación se muestra lo que se conoce como SceneGraph o gráfico de escena. Básicamente, toda aplicación JavaFX puede representarse como un árbol de la siguiente manera:

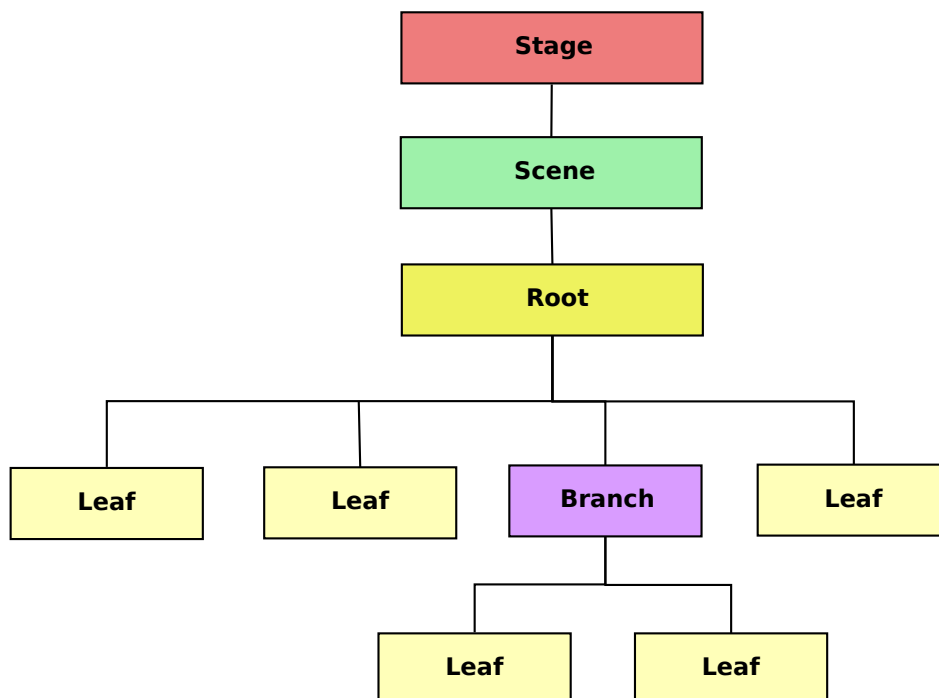


Figura 26: SceneGraph o árbol de una escena con JavaFX

El **Stage**, es la ventana (marco, con sus botones de cerrar, minimizar, ...) , el **Scene** es la escena (recuerda el símil **Stage** = Escenario de un teatro, **Scene** = una escena que se está representando en un teatro, con sus actores, su atrezzo, ...), y luego tenemos el nodo **Root**, que es el nodo padre de todos los nodos que cuelgan de una escena. Este nodo root puede tener hijos ‘hoja’ o ‘leaf’, que son finales y no tienen más hijos. Y nodos **Branch**, que pueden tener, a su vez, más hijos ‘Leaf’ y/o ‘Branch’.

Esto se consigue en JavaFX mediante una jerarquía de clases Abstractas.

- La clase base para todos los nodos (sea el root, leaf o branch) es **Node** (<https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/Node.html>).
- El nodo **Root**, extiende directamente a la clase abstracta **Parent**, que en realidad es extendida por todos los componentes con capacidad de albergar Nodos (que a su vez extiende a **Node**) <https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/Parent.html>
- A los nodos **Leaf** u hoja, normalmente se les conoce como controles, porque extienden la clase base Control. (<https://openjfx.io/javadoc/13/javafx.controls/javafx/scene/control/Control.html>)
- Los nodos **Branch**, también extienden la clase **Parent**.



Figura 27: Partes en nuestro primer programa

3.4. Ciclo de vida de una aplicación

Con ciclo de vida, nos referimos a las fases por las que pasa una aplicación JavaFX desde que es ejecutada. En el fondo, se trata de una serie de métodos de la clase **Application** (y que hereda nuestra clase **Main**, ya que está extendiendo a **Application**) y que se ejecutan en un orden cronológico concreto, y que podemos sobrescribir si deseamos para realizar alguna acción.

Dichos métodos, y el orden en que se ejecutan es:

1. `init()`
2. `start()`
3. `stop()`

1. Método Init

Se ejecuta justo después de ejecutarse el constructor de `Application`. Se usa para inicializaciones específicas de la aplicación (leer algún fichero de propiedades, iniciar alguna conexión, ...), **que no sean relativas a GUI (interfaz gráfica)!!!**. Esto es debido a que en estos momentos, el `Thread` que maneja la interfaz aún no está disponible.

2. Método Start

Se llama cuando el hilo de ejecución de la interfaz gráfica está por fin disponible (creado y listo). Como puedes ver en el código de nuestra primera aplicación, te proporciona un escenario o `Stage` que puedes usar como tu ventana principal.

En este método se cargará todo lo referente a interfaz gráfica: `Stages`, `Scenes`, nodos, `css`, ...

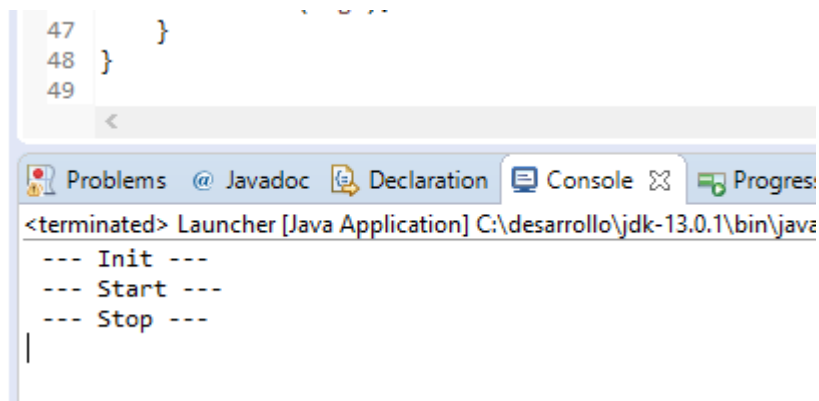
3. Método Stop

Cuando la aplicación está a punto de cerrar. Es el sitio ideal para hacer tareas de limpieza como cierre de conexiones, ficheros, ...

En la siguiente modificación de la clase Main.java, hemos sobrescrito los métodos que aún no teníamos, y hemos modificado el start. En cada uno hemos puesto un **`System.out.println("Estoy en el método X")`** para poder observar como evoluciona una aplicación.

```
public class Launcher extends Application {  
    private Button btn;  
  
    @Override  
    public void init() throws Exception {  
        System.out.println(" --- Init --- ");  
        super.init();  
    }  
  
    @Override  
    public void stop() throws Exception {  
        System.out.println(" --- Stop --- ");  
        super.stop();  
    }  
  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        System.out.println(" --- Start --- ");  
        btn = new Button();  
        btn.setText("Click me!");  
        BorderPane pane = new BorderPane();  
        pane.setCenter(btn);  
  
        Scene scene = new Scene(pane, 300, 200);  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```


El resultado de ejecutar esto, produce la siguiente salida de consola:



```
47     }  
48 }  
49  
<terminated> Launcher [Java Application] C:\desarrollo\jdk-13.0.1\bin\java  
--- Init ---  
--- Start ---  
--- Stop ---  
|
```

Figura 28: Salida de consola al ejecutar nuestra aplicación

4. Layouts

Los layouts son elementos que sirven para organizar Controles (botones, campos de texto, ... incluso otros layouts). Cada uno tiene distintas características que pueden hacerlos más convenientes según qué situación.

Todos se encuentran en el paquete `javafx.scene.layout`.

Vamos a darle un repaso rápido a los principales. Para esta sección, vamos a crear un proyecto llamado 02-DemoLayouts, donde vamos a ir añadiendo una clase java que extiende de Application para demostrar el uso de cada interfaz, en lugar de crear proyectos separados para cada una.

En el apartado 4.1, pondremos el enlace a un vídeo donde se describe, además de lo relativo al Hbox, cómo creamos el proyecto inicial.

Todo el código de este apartado, está disponible en:

<https://github.com/pes130/02-DemoLayouts>

4.1. Hbox

El vídeo guiado de este ejercicio, y de la creación del proyecto, puedes verlo en:

<https://youtu.be/89vFhkDAu3w>

El Hbox disponen los controles horizontalmente, uno a continuación de otro. En nuestro caso, vamos a construir esta aplicación:

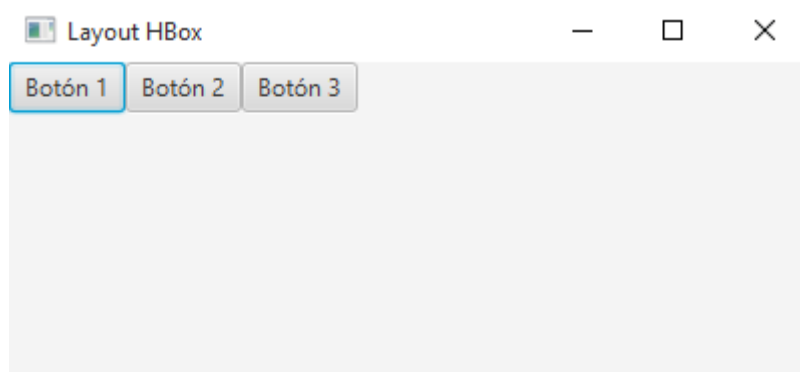


Figura 29: Layout HBox

Usando este layout, los controles se van añadiendo horizontalmente uno a continuación de otro. Sin intervención por parte del programador.

La parte principal de nuestra clase es la siguiente (recuerda que puedes consultar todo en Github):

```
@Override
public void start(Stage primaryStage) {
    Button btn1 = new Button("Botón 1");
    Button btn2 = new Button("Botón 2");
    Button btn3 = new Button("Botón 3");

    HBox hbox = new HBox();
    hbox.getChildren().addAll(btn1, btn2, btn3);

    // Cambiar espaciado entre componentes
    hbox.setSpacing(10);

    // Cambiar padding
    hbox.setPadding(new Insets(10, 20, 10, 20));

    // Cambiar margin
    HBox.setMargin(btn2, new Insets(30));

    Scene scene = new Scene(hbox, 400, 300);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Layout HBox");
    primaryStage.show();
}
```

Como aspecto destacable, decir que hemos usado 3 métodos de Hbox para definir aspectos gráficos:

- **hbox.setSpacing (valor)** → introduce un espacio entre los controles dentro del layout de **valor** píxeles.
- **hbox.setPadding(new Insets (arriba, derecha, abajo, izquierda))** → Introduce un espacio entre los bordes del layout y su contenido de la cantidad de píxeles que indiquemos **arriba, abajo, derecha** y a **izquierda**. Es necesario hacerlo a través de este objeto **Insets**. Hay 4 valores porque sirve para introducir margin respecto a: **1.** al borde superior, **2.** al borde derecho, **3.** al borde de abajo, **4.** al borde izquierdo. Es igual que el **padding** de CSS.

Para poner el mismo valor en las 4 partes, también podríamos haber usado **hbox.setPadding(new Insets (20))**

- **Hbox.setMargins(control, new Insets (arriba, derecha, abajo, izquierda))** → Sirve para aplicar margen, de manera independiente, a cada uno de los lados del control que le pasemos como primer parámetro. Funciona igual que el margin de html. Por ejemplo, para poner al btn2 un margen de 30 píxeles por los 4 lados, podemos hacer:

```
HBox.setMargin(btn2, new Insets(30));
```

o bien:

```
HBox.setMargin(btn2, new Insets(30, 30, 30, 30));
```

4.2. FXML y separación vista/controlador

Podemos ver aquí el paso a paso https://youtu.be/I4Vkb1b_t3M

FXML es una gramática de XML que nos permite declarar un contenido para una escena. Lo normal es que en el raíz tengamos un nodo de tipo container o layout (en nuestro caso, un Hbox o disposición horizontal) y de que de él vayan colgando el resto de controles u otros layouts.

Nuestro archivo **FXML** quedaría así:

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

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.HBox?>

<HBox maxHeight="-Infinity" maxWidth="-Infinity"
    minHeight="-Infinity" minWidth="-Infinity" prefHeight="225.0"
    prefWidth="399.0" spacing="10.0"
    xmlns="http://javafx.com/javafx/11.0.1"
    xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="es.cursojavafx.demolayouts.controller.HBoxFXML02Controller">
    <children>
        <Button fx:id="btn1" mnemonicParsing="false" text="Botón 1" />
        <Button fx:id="btn2" mnemonicParsing="false" text="Botón 2">
            <HBox.margin>
                <Insets bottom="30.0" left="30.0" right="30.0" top="30.0" />
            </HBox.margin>
        </Button>
        <Button fx:id="btn3" mnemonicParsing="false" text="Botón 3" />
    </children>
</HBox>
```

```
<Button fx:id="btn4" mnemonicParsing="false" text="Botón 4" />

</children>

<padding>

    <Insets bottom="10.0" left="20.0" right="20.0" top="10.0" />

</padding>

</HBox>
```

Conocer todas las posibilidades del espacio de nombres <http://javafx.com/fxml/1> es prácticamente inabordable, por la multitud de atributos y subelementos que hay que conocer. Para evitar esta complejidad, podemos usar la herramienta **SceneBuilder**, que nos permite hacer todo de manera visual:

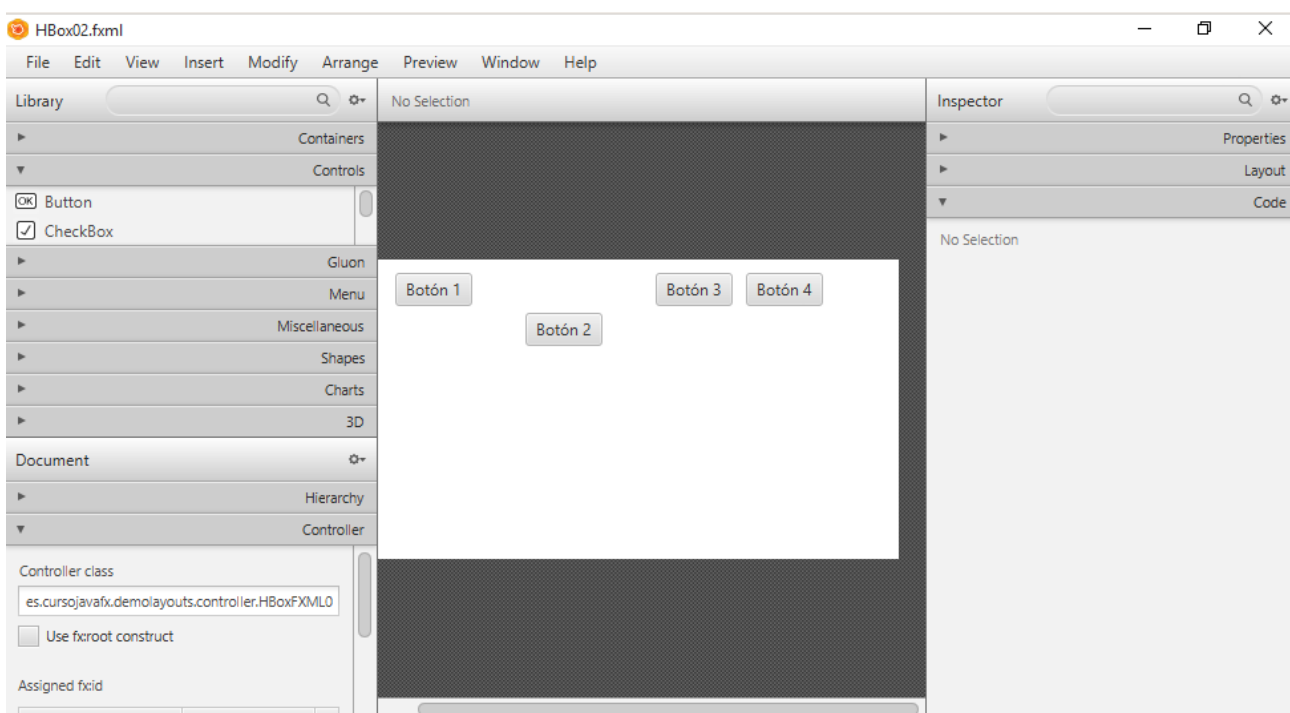


Figura 30: Scene Builder

En este apartado, hemos tratado de hacer la misma interfaz que en el apartado anterior, con la salvedad de que en lugar de componer el contenido de la escena en código (en java), hemos separado la vista de la lógica de la aplicación como primer paso para conseguir una visión MVC (Modelo Vista Controlador). Se trata de una mejora muy importante que JavaFX introduce respecto a Swing o AWT.

Para que nuestra aplicación pueda usar este fichero fxml, hemos hecho 2 cosas:

- Hemos creado una **carpeta de recursos** para alojar ficheros de utilidades, como por ejemplo, ficheros **fxml**.

- Hemos puesto, en el método **start ()** de nuestra clase principal, el siguiente código para cargar el FXML:

```
FXMLLoader fxml = new FXMLLoader(getClass().getResource("HBox02.fxml"));  
Parent hbox = fxml.load();
```

En la primera línea cargamos el archivo **HBox02.fxml** de nuestra carpeta de recursos, y en la segunda, parseamos su contenido a un objeto **Parent**. **Parent** es una clase abstracta extendida por nodos que albergan hijos (contenedores, layouts, ...): La documentación puedes verla aquí: <https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/Parent.html>

Por último, JavaFX nos permite **asociar un controlador a cada vista FXML**. Un **Controlador** es una clase especial que mapea los controles que pongamos en nuestro FXML, nos permite implementar eventos para los controles que tengamos en nuestra vista, etc. En definitiva, tras la vista separada en archivos fxml, es otra aportación más de JavaFX para conseguir aplicaciones siguiendo el patrón **MVC modelo vista controlador**:

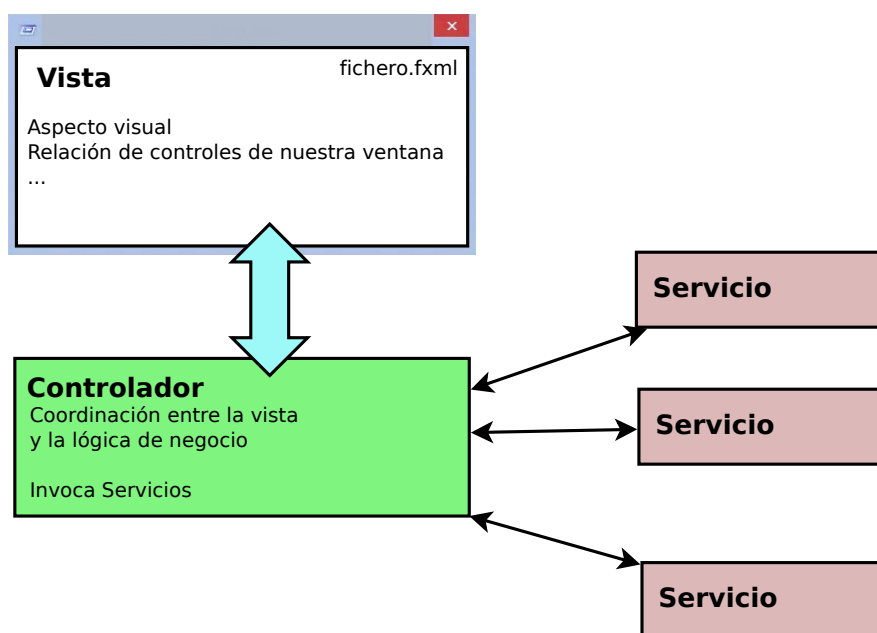


Figura 31: Modelo Vista Controlador en JavaFX

Podemos **asociar una vista a su controlador** de 2 formas:

- De **manera estática**, indicándolo en el fichero fxml:

```
<Hbox ...  
fx:controller="es.cursojavafx.demolayouts.controller.HBoxFXML02Controller">
```

O desde SceneBuilder:

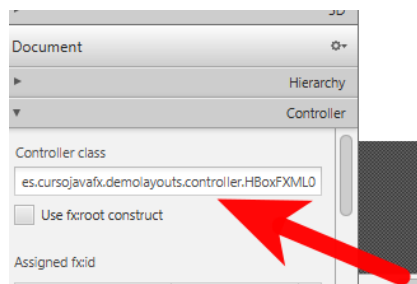


Figura 32: Poner controlador a vista con Scene Builder

- De **manera programática**, haciéndolo en Java:

```
Parent hbox = fxml.load();  
HBoxFXML02Controller hboxCtrl = new HBoxFXML02Controller();  
fxml.setController(hboxCtrl);
```

Usar en java controles declarados en FXML

A partir de aquí, surge una duda muy común: ¿Cómo podemos manipular en java, en nuestro controlador, controles (botones, campos de texto, ...) declarados en el FXML?

La respuesta es sencilla, con **ids** y **anotaciones**:

1. En FXML ponemos un id al control en cuestión. Por ejemplo, para nuestro botón btn2, en SceneBuilder hacemos:

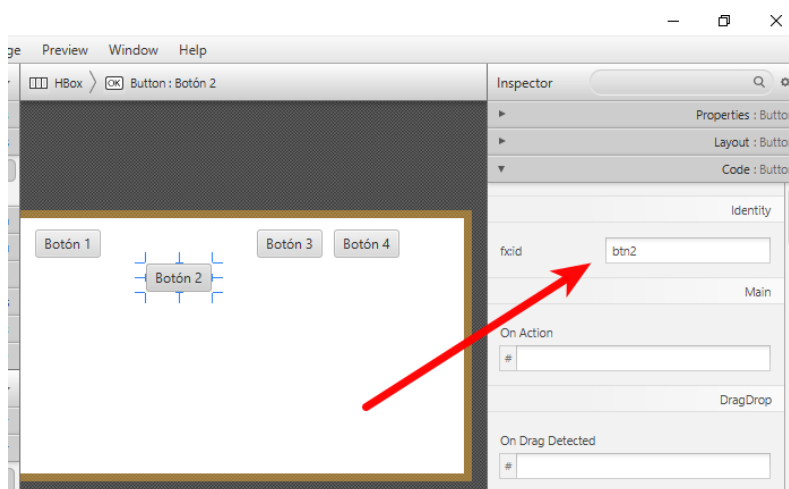


Figura 33: Poner id a un botón con SceneBuilder

También puede hacerse directamente en el FXML:

```
<Button fx:id="btn1" mnemonicParsing="false" text="Botón 1" />
<Button fx:id="btn2" mnemonicParsing="false" text="Botón 2">
  <HBox.margin>
    <Insets bottom="30.0" left="30.0" right="30.0" top="30.0" />
  </HBox.margin>
</Button>
```

Figura 34: Id a Botón 2 en fxml

2. En el **controlador** (HBoxFXML02Controller), añadimos una propiedad privada cuyo nombre sea el id del control en cuestión, y la anotamos con **@FXML**:

```
public class HBoxFXML02Controller implements Initializable {
    @FXML
    private Button btn1;

    @FXML
    private Button btn2;

    @FXML
    private Button btn3;
```

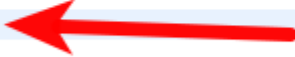


Figura 35: Anotación de controles en el controlador

4.3. VBox

El comportamiento de **Vbox** es muy similar al de **Hbox**, pero organiza los contenidos verticalmente de manera automática.

En este caso, vamos a construir lo siguiente:

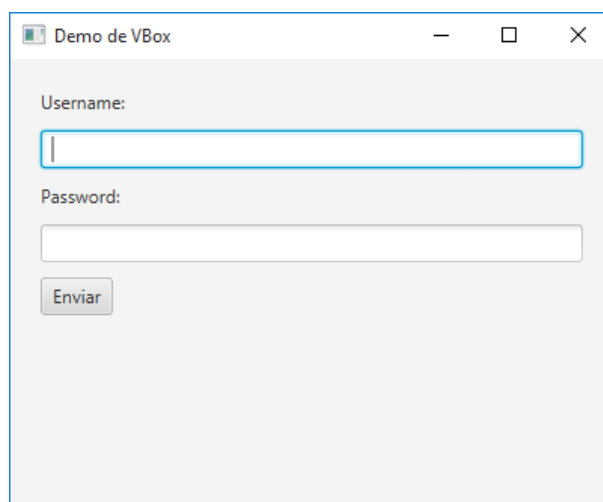


Figura 36: Ejemplo de VBox

El enlace de vídeo guiado es: <https://youtu.be/qdfC0jiFd6o>.

Recuerda que el código fuente está en github. Esta clase **VBoxEjemplo03.java** puedes verla en <https://github.com/pes130/02-DemoLayouts/blob/master/src/es/cursojavafx/demolayouts/VBoxEjemplo03.java>

Por no ser demasiado repetitivos, ya que este caso es muy parecido al anterior, vamos a introducir un par de cosas adicionales sobre JavaFX:

- Introducir 3 controles nuevos: **Label**, **TextField** y **PasswordField**.
- Introducir las colecciones **ObservableList<T>**.

El código del método `start` de nuestra clase **VboxEjemplo03.java**, contiene lo siguiente:

```
@Override
public void start(Stage primaryStage) {
    Label lbl_username = new Label("Username: ");
    TextField field_username = new TextField();
    Label lbl_password = new Label("Password: ");
    PasswordField field_password = new PasswordField();
    Button btn_send = new Button("Enviar");
```

```
// Creamos una lista de controles

ObservableList<Control> controls =
FXCollections.observableArrayList(lbl_username, field_username,
                                  lbl_password, field_password, btn_send);

VBox vbox = new VBox();
// Añadimos nuestra lista de controles al layout
vbox.getChildren().addAll(controls);

vbox.setPadding(new Insets(20));
vbox.setSpacing(10);

// Alineación
vbox.setAlignment(Pos.CENTER);

Scene scene = new Scene(vbox, 400, 300);
primaryStage.setTitle("Demo de VBox");
primaryStage.setScene(scene);
primaryStage.show();
}
```

Sobre los nuevos controles: **Label**, **TextField** y **PasswordField**, no hay mucho que decir, los iremos viendo poco a poco conforme aumente la complejidad de los ejemplos.

Por otro lado, JavaFX introduce una colección, **ObservableList<E>** (<https://openjfx.io/javadoc/13/javafx.base/javafx/collections/ObservableList.html>). Se trata de una colección que implementa las conocidas interfaces de *java.util* **Collection<E>** y **List<E>** entre otras, y que aporta la ventaja de poder, mediante **listeners**, observar cambios en los elementos de la lista para poder asociarles **eventos**. También veremos más adelante, ejemplos más específicos acerca de este tipo de listas.

Método `setAlignment`

Vbox nos permite invocar a este método para especificarle cómo queremos alinear su contenido:

`vbox.setAlignment(posicion)`

El parámetro **posición** puede tomar los valores:

- `Pos.TOP_LEFT`
- `Pos.TOP_CENTER`
- `Pos.TOP_RIGHT`

- Pos.CENTER_LEFT
- Pos.CENTER
- Pos.CENTER_RIGHT
- Pos.BOTTOM_LEFT
- Pos.BOTTOM_CENTER
- Pos.BOTTOM_RIGHT
- Pos.BASELINE_LEFT
- Pos.BASELINE_CENTER
- Pos.BASELINE_RIGHT

Por ejemplo, si cambiamos las dimensiones de la escena a 600x600, y ponemos una alineación CENTER, tenemos:

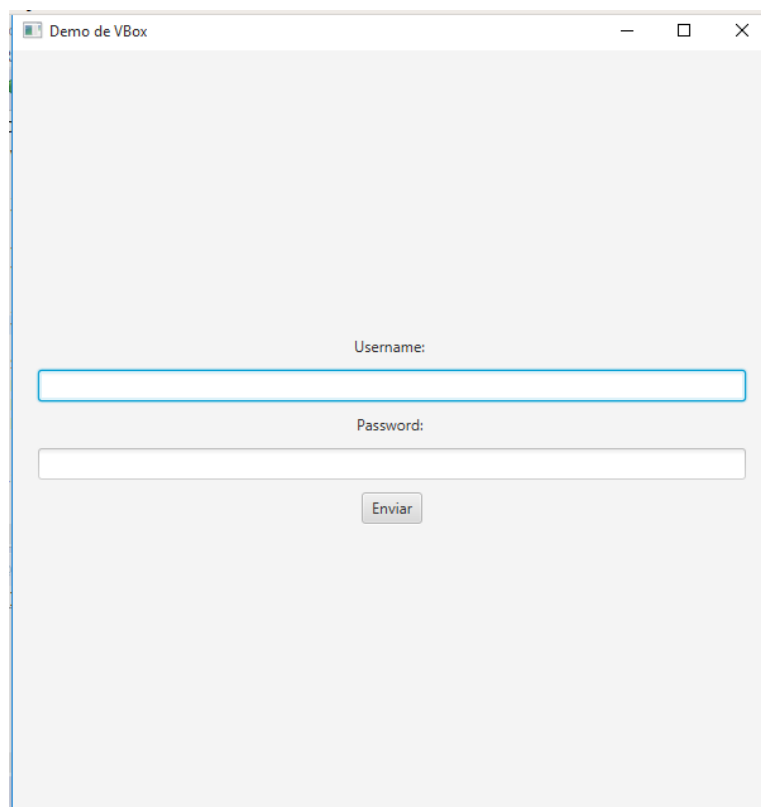


Figura 37: Alineación Pos.CENTER

4.4. BorderPane

BorderPane es un layout que divide la pantalla en 5 zonas, permitiéndonos añadir, de una manera sencilla, controles y paneles a cada una de estas zonas.

Las **áreas en cuestión son:**

- Top
- Left
- Center
- Right
- Bottom

En la siguiente figura, podemos ver cómo se distribuyen:

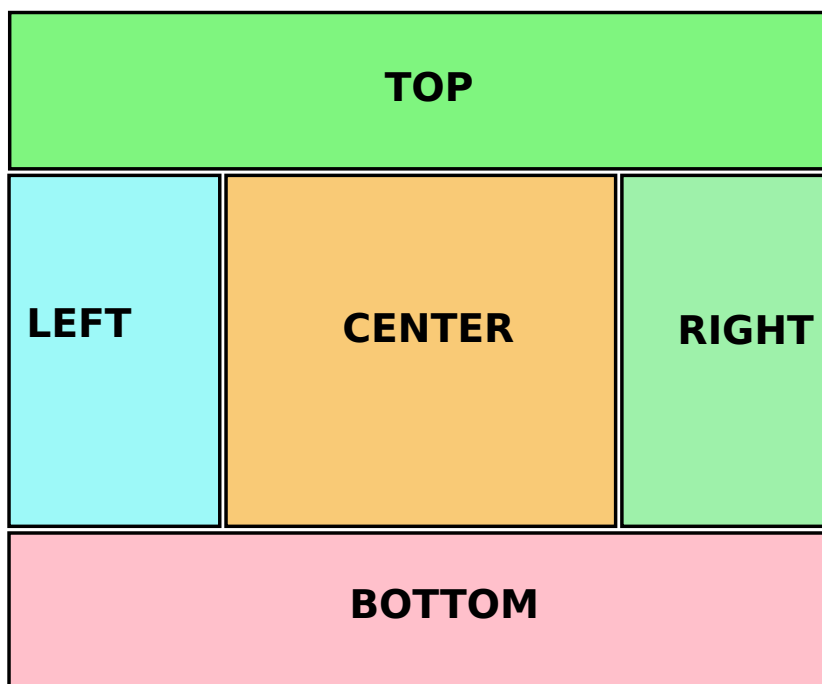


Figura 38: Distribución de las zonas de un BorderPane

El vídeo guiado para crearlo es el siguiente: <https://youtu.be/nbXWTs2vxTk>

En nuestro caso, vamos a crear el siguiente ejemplo:

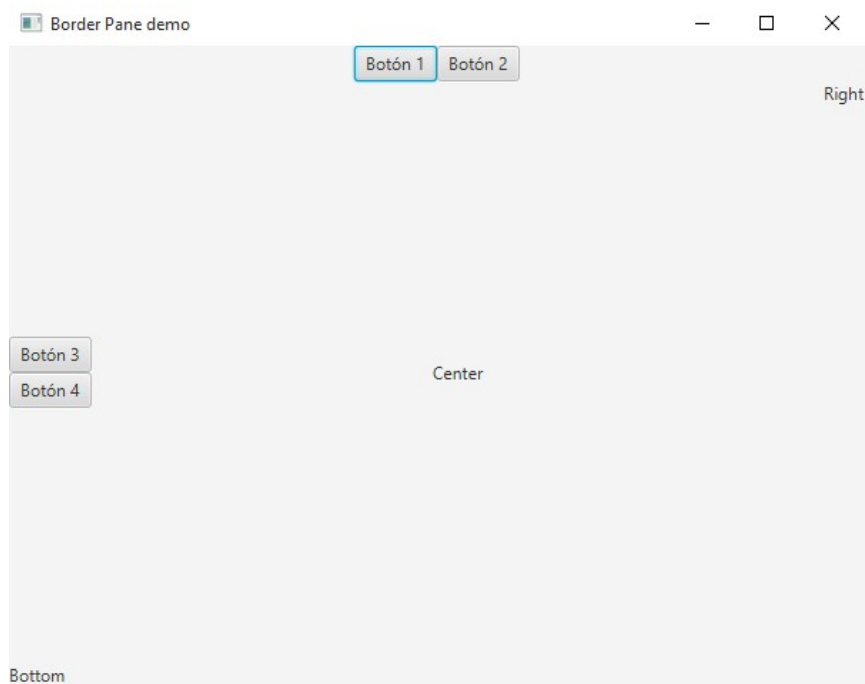


Figura 39: Ejemplo de border pane

- En la zona TOP hemos metido un Hbox con 2 botones, el Botón 1 y el Botón 2
- En la zona LEFT hemos metido un VBox con otros 2 botones: el Botón 3 y el Botón 4.
- En la zona RIGHT hemos metido un solo Label con el texto 'Right'
- En la zona BOTTOM, tenemos un label con el texto 'Bottom',
- En la zona CENTER, tenemos un label con el texto 'Center'.

El código principal de nuestra clase es el siguiente:

```
@Override
public void start(Stage primaryStage) {
    // Creación de elementos para el TOP
    Button btn_top_1 = new Button("Botón 1");
    Button btn_top_2 = new Button("Botón 2");

    // Creación de elementos para el LEFT
    Button btn_left_1 = new Button("Botón 3");
    Button btn_left_2 = new Button("Botón 4");

    // Creación de elementos para el RIGHT
    Label lbl_right = new Label("Derecha");
```

```
// Creación de elementos para el BOTTOM
Label lbl_bottom = new Label("Bottom de mi aplicación usando un border
Pane");

// Creación de elementos para el RIGHT
Label lbl_center = new Label("Centro");

//Creación de layouts
HBox hbox_top = new HBox();
hbox_top.setAlignment(Pos.CENTER);

VBox vbox_left = new VBox();
vbox_left.setAlignment(Pos.CENTER);
BorderPane bp = new BorderPane();

hbox_top.getChildren().addAll(btn_top_1, btn_top_2);
vbox_left.getChildren().addAll(btn_left_1, btn_left_2);

// Añadimos nodos al border pane
bp.setTop(hbox_top);
bp.setRight(lbl_right);
bp.setBottom(lbl_bottom);
bp.setLeft(vbox_left);
bp.setCenter(lbl_center);

Scene scene = new Scene(bp, 600, 450);
primaryStage.setTitle("Border Pane Demo");
primaryStage.setScene(scene);
primaryStage.show();
}
```

4.5. FlowPane

FlowLayout es muy parecido a VBox y a Hbox. Organiza los elementos uno a continuación de otro, de manera horizontal (por defecto), o vertical, según le indiquemos.

Veamos el siguiente ejemplo. Se llama **FlowPane05.java**:

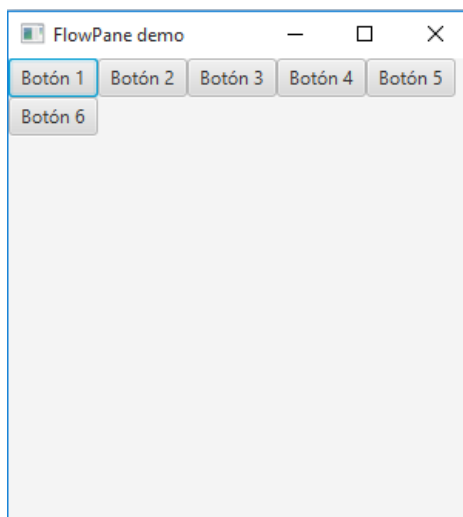


Figura 40: FlowPane con elementos dispuestos horizontalmente

Veamos el código para esto:

```
public class FlowPane05 extends Application {  
    @Override  
    public void start(Stage primaryStage) {  
        Button btn1 = new Button("Botón 1");  
        Button btn2 = new Button("Botón 2");  
        Button btn3 = new Button("Botón 3");  
        Button btn4 = new Button("Botón 4");  
        Button btn5 = new Button("Botón 5");  
        Button btn6 = new Button("Botón 6");  
  
        FlowPane fp = new FlowPane();  
        fp.getChildren().addAll(btn1, btn2, btn3, btn4, btn5, btn6);  
  
        Scene scene = new Scene(fp, 300, 300);  
    }  
}
```

```
primaryStage.setScene(scene);
primaryStage.setTitle("FlowPane demo");
primaryStage.show();
}
public static void main(String[] args) {
    launch(args);
}
}
```

En principio es bastante sencillo. Una cosa destacable, es que si los elementos dispuestos horizontalmente llegan al borde de la ventana, bajan automáticamente a la siguiente línea (Prueba a redimensionar la ventana con el ratón):

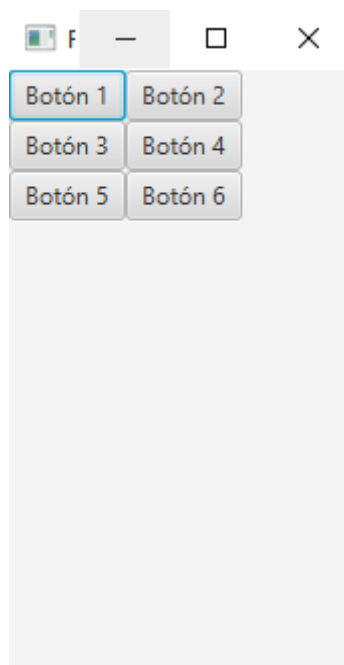


Figura 41: FlowPane

Por defecto, el FlowPane organiza los elementos horizontalmente, pero podemos cambiar la organización y ponerlo **vertical**. El cambio sería este:

```
...
FlowPane fp = new FlowPane(Orientation.VERTICAL);
fp.getChildren().addAll(btn1, btn2, btn3, btn4, btn5, btn6);
...
```


El resultado es el siguiente:

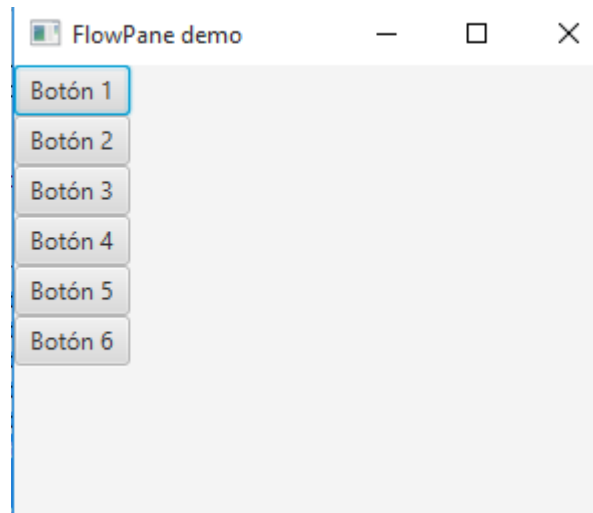


Figura 42: FlowPane organizando sus nodos hijos verticalmente

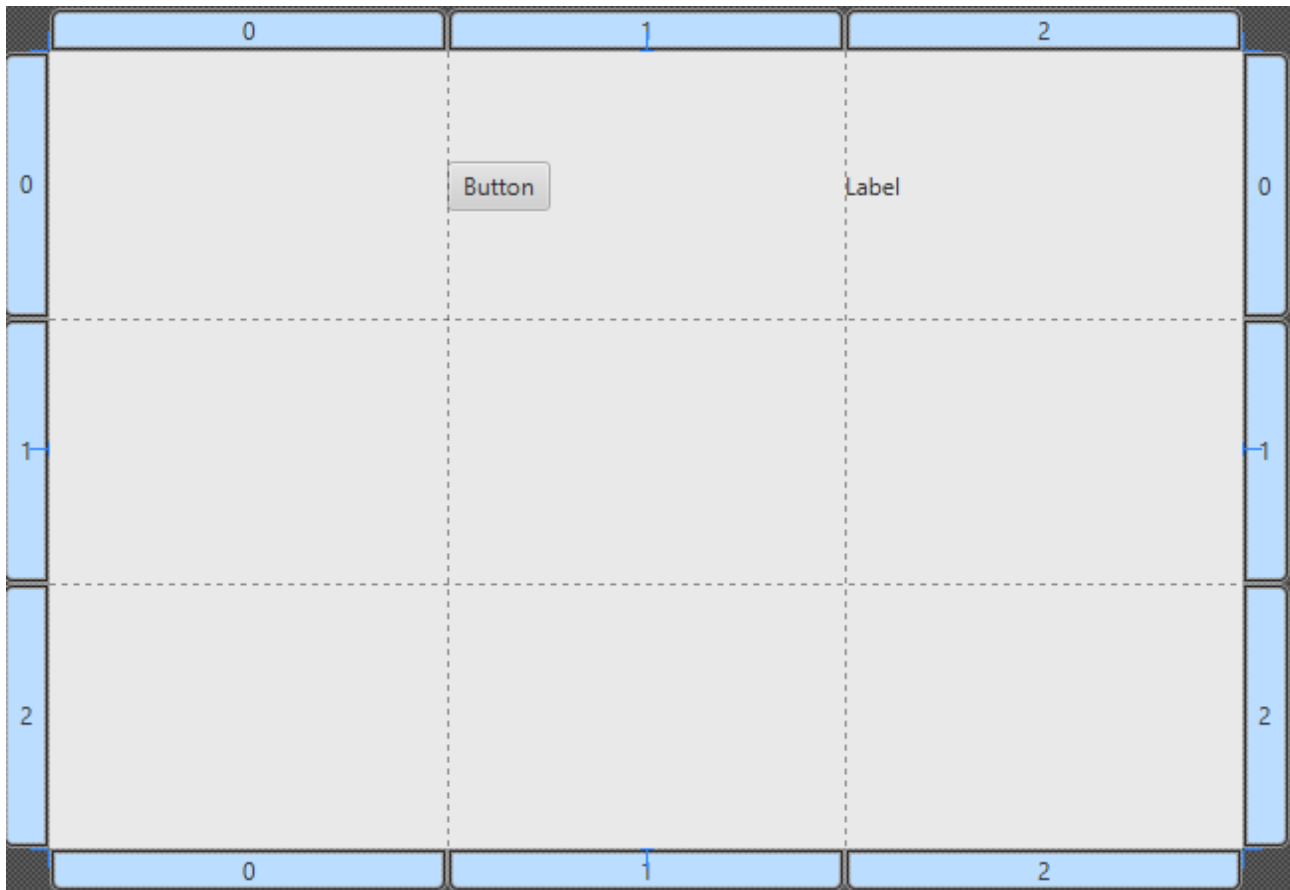
No sólo podemos indicarlo en el constructor, sino cambiarlo en cualquier momento con **setOrientation**:

```
fp.setOrientation(Orientation.VERTICAL);
```

4.6. GridPane

GridPane es un layout que organiza los elementos en forma de tabla. Puede ser útil para ciertas ventanas como un formulario.

Básicamente, colocamos cada control en el layout indicando su columna y su fila. Imagina que tenemos una rejilla de 3 x 3, sería algo así:



Vamos a probar un ejemplo de manera programática, y otro con fxml.

Veamos cómo hacer el siguiente ejemplo:

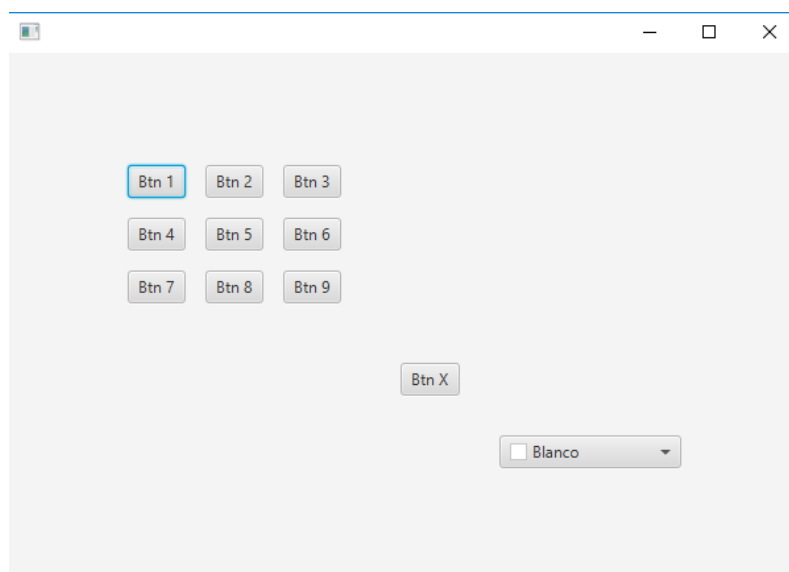


Figura 43: Ejemplo de Grid Layout

Los controles que hemos colocado, se están organizando según la siguiente rejilla:

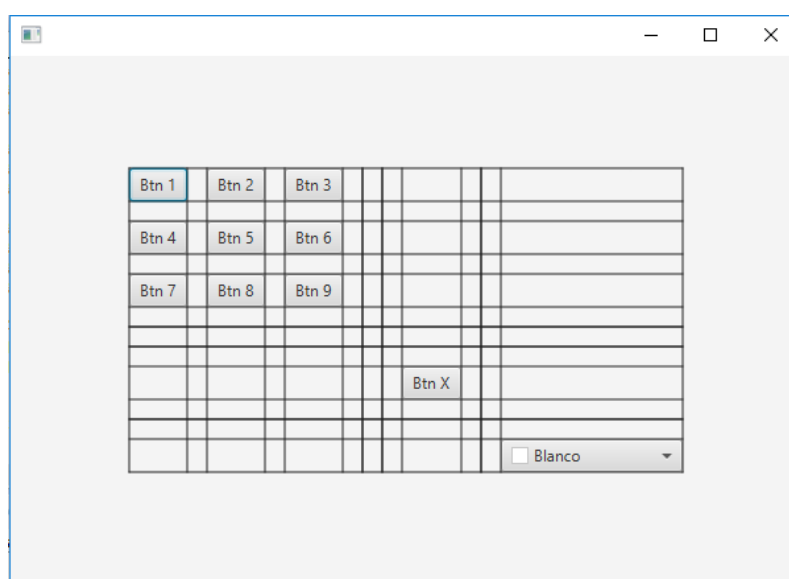


Figura 44: Tabla sobre la que se distribuye nuestro ejemplo

4.5.1. De manera programática

Vamos a hacerlo, en primer lugar creando los controles y el GridPane en java, es decir, de manera programática.

La clase se llama GridLayoutEjemplo06.java, y puedes consultar su código fuente en <https://github.com/pes130/02-DemoLayouts/blob/master/src/es/cursojavafx/demolayouts/GridLayoutEjemplo06.java>

Puedes ver el vídeo de su realización paso a paso en <https://youtu.be/DJ9U0XT4voA>.

Lo más importante es lo siguiente:

```
GridPane gridPane = new GridPane();
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setAlignment(Pos.CENTER);
gridPane.setPadding(new Insets(20));
gridPane.setGridLinesVisible(true);

// Añadimos la primera fila
gridPane.add(btn1, 0, 0);
gridPane.add(btn2, 1, 0);
gridPane.add(btn3, 2, 0);

// Añadimos la segunda fila
gridPane.add(btn4, 0, 1);
gridPane.add(btn5, 1, 1);
gridPane.add(btn6, 2, 1);

// Añadimos la tercera fila
gridPane.add(btn7, 0, 2);
gridPane.add(btn8, 1, 2);
gridPane.add(btn9, 2, 2);

// Añadimos un color picker
gridPane.add(cp, 5, 5);
```

Como ves, para añadir un control al gridPane, usamos el método:

gridPane (control, columna, fila)

Donde columna y fila son el número de columna y fila respectivamente, empezando en 0. Es decir, si tenemos una tabla de 3 x 3, y queremos añadir el botón btn1 en el centro, hemos de escribir:

gridPane (btn1, 1, 1)

Al no indicar, a priori, un tamaño en filas y columnas para la tabla, podemos añadir un control donde queramos. En este caso, hemos añadido un ColorPicker (botón que abre una selector de colores) en la fila 6 columna 6 (5,5).

Otros métodos interesantes de **GridPane** son:

- **setHgap(valor)** → sirve para meter un espacio de **valor** píxeles entre las columnas
- **setVgap(valor)** → sirve para meter un espacio de **valor** píxeles entre las filas de la tabla



Figura 45: Hgap y Vgap

4.5.2. Separando la vista en FXML

En este caso, vamos a hacer lo mismo, pero separando la vista en un archivo FXML.

En este caso, usamos 2 ficheros:

- Nuestra clase principal, **GridLayoutFXML07.java**, disponible en: <https://github.com/pes130/02-DemoLayouts/blob/master/src/es/cursojavafx/demolayouts/GridLayoutFXML07.java>
- Nuestro fichero **GridLayoutFXML07.fxml**, con nuestra vista, disponible en: <https://github.com/pes130/02-DemoLayouts/blob/master/resources/es/cursojavafx/demolayouts/GridLayoutFXML07.fxml>

Puedes ver el **proceso paso a paso de creación** en el vídeo: <https://youtu.be/yG-jkxveE3E>

Al final, el método start de nuestra clase principal, se queda como

```
@Override
public void start(Stage primaryStage) throws IOException {
    FXMLLoader fxmlLoader = new
FXMLLoader(getClass().getResource("GridLayoutFXML07.fxml"));
    Parent gp = fxmlLoader.load();
    Scene scene = new Scene(gp);
    primaryStage.setTitle("Gridlayout con FXML");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

5. Eventos y Listeners

El proyecto que hemos creado para este apartado, lo tienes en <https://github.com/pes130/03-DemoEventos>

Hasta ahora, nos hemos dedicado a la creación de Containers y de Controles, pero no hemos visto nada sobre controles.

Como siempre, vamos a abordarlo de 2 maneras (aunque recomendando siempre el patrón MVC):

- Creando controles, containers y todo, en Java
- Creando controles y containers en FXML.

5.1. Creando eventos SIN controlador

El vídeo paso a paso de este apartado, puede encontrarse en <https://youtu.be/M1rMVcgjIvQ>.

Vamos a hacer el siguiente ejemplo. Es muy simple y consiste en una ventana con 2 botones y una etiqueta. Cada vez que pulsemos el botón sumar, se nos incrementa un contador, y cuando le demos a reiniciar, se pone a 0.

Además, hemos puesto a la etiqueta otro evento, para que cuando le pases el ratón por encima, aumente su tamaño de letra, y cuando no, vuelva a su tamaño normal:

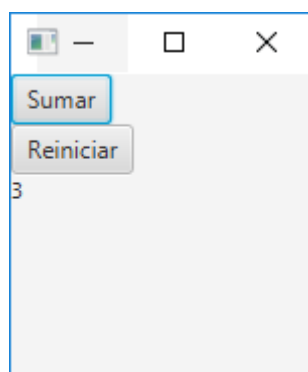


Figura 46: Nuestro ejemplo

En este caso, al no separar la vista en un fichero FXML aparte, no tenemos clase controladora, y por tanto, creamos los controles y el layout en nuestra clase principal, y los eventos asociados a los botones también.

Por tanto, todo lo importante de este ejemplo está en **ActiionsDemo01.java**:

<https://github.com/pes130/03-DemoEventos/blob/master/src/es/cursojavafx/demoeventos/ActionsDemo01.java>

Podemos asociar un evento a un control de 2 maneras:

1. **Usando una Lambda Expression**, disponible desde Java 8. Es la opción más recomendada, ya que queda un código más simple y contenido.
2. **Usando una clase anónima**. En programación de eventos es muy típico hacer esto. En este caso, el evento en sí debe extender la clase abstracta de JavaFX `EventHandler<ActionEvent>`. Ya que normalmente, la funcionalidad de un botón no va a ser reutilizada (y si lo fuera, podríamos encerrarla en un método privado), no tiene mucho sentido hacer clases con nombre sólo para la funcionalidad de un control, por lo que se suele optar por una clase abstracta.

Veamos como hacer lo mismo, incrementar el contador al darle al botón `btnSumar`, de las 2 maneras:

Asociar evento con lambda expression:

```
btnSumar.setOnAction( event -> {  
    contador++;  
    lblResultado.setText (String.valueOf (contador));  
});
```

Asociar evento con clase anónima:

```
btnReiniciar.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        contador = 0;  
        lblResultado.setText (String.valueOf (contador));  
    }  
});
```

Ambas soluciones son válidas.

Para aumentar o disminuir el tamaño de la fuente del **lblResultado** al pasar el ratón por encima, lo hacemos con los eventos **onMouseEntered**, y **onMouseExited**:

```
lblResultado.setOnMouseEntered(event -> {  
    lblResultado.setFont (new Font (20));  
});  
  
lblResultado.setOnMouseExited(event -> {  
    lblResultado.setFont (new Font (11));  
});
```


5.2. Creando eventos CON controlador (MVC Friendly)

El vídeo paso a paso para este apartado está en <https://youtu.be/k2HwAq1fHmc>

En este caso, en lugar de crear la interfaz y la lógica asociada a los botones en la clase principal, optamos por el enfoque MVC (siempre, mucho más recomendable).

El código que hemos usado es:

- Un FXML con la vista **ActionDemoFXML02.fxml**: <https://github.com/pes130/03-DemoEventos/blob/master/resources/es/cursojavafx/demoeventos/ActionDemoFXML02.fxml>
- Nuestra clase principal **ActionDemoFXML02.java**: <https://github.com/pes130/03-DemoEventos/blob/master/src/es/cursojavafx/demoeventos/ActionDemoFXML02.java>
- Una clase controladora asociada a nuestra vista en FXML **ActionDemoControllerFXML02.java**
<https://github.com/pes130/03-DemoEventos/blob/master/src/es/cursojavafx/demoeventos/controllers/ActionDemoControllerFXML02.java>

Lo importante aquí es que asociamos la vista (**ActionDemoFXML02.fxml**) con el controlador con el siguiente atributo en nuestro FXML:

```
...  
<VBox alignment="TOP_CENTER" prefHeight="200.0" prefWidth="150.0" spacing="15.0"  
xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/11.0.1"  
fx:controller="es.cursojavafx.demoeventos.controllers.ActionDemoControllerFXML02">  
    <children>  
        <Button fx:id="btnSumar" mnemonicParsing="false" onAction="#incrementarContador"  
text="Sumar" />  
        <Button fx:id="btnReiniciar" mnemonicParsing="false" onAction="#reiniciarContador"  
text="Reiniciar" />  
    </children>  
...
```

Y en nuestra clase controladora, **ActionDemoControllerFXML02**, el código queda mucho más sencillo que antes:

```
@FXML  
void incrementarContador(ActionEvent event) {  
    this.contador++;  
    this.lblContador.setText(String.valueOf(this.contador));  
}
```

```
@FXML
void incrementarFuente(MouseEvent event) {
    this.lblContador.setFont(new Font(30));
}

@FXML
void reestablecerFuente(MouseEvent event) {
    this.lblContador.setFont(new Font(11));
}

@FXML
void reiniciarContador(ActionEvent event) {
    this.contador = 0;
    this.lblContador.setText(String.valueOf(this.contador));
}
```

Recuerda que, tanto el **id** de los controles, como el **método** al que se llamará al producirse algún evento, se establecen en **SceneBuilder** en la sección **code** de cada control:

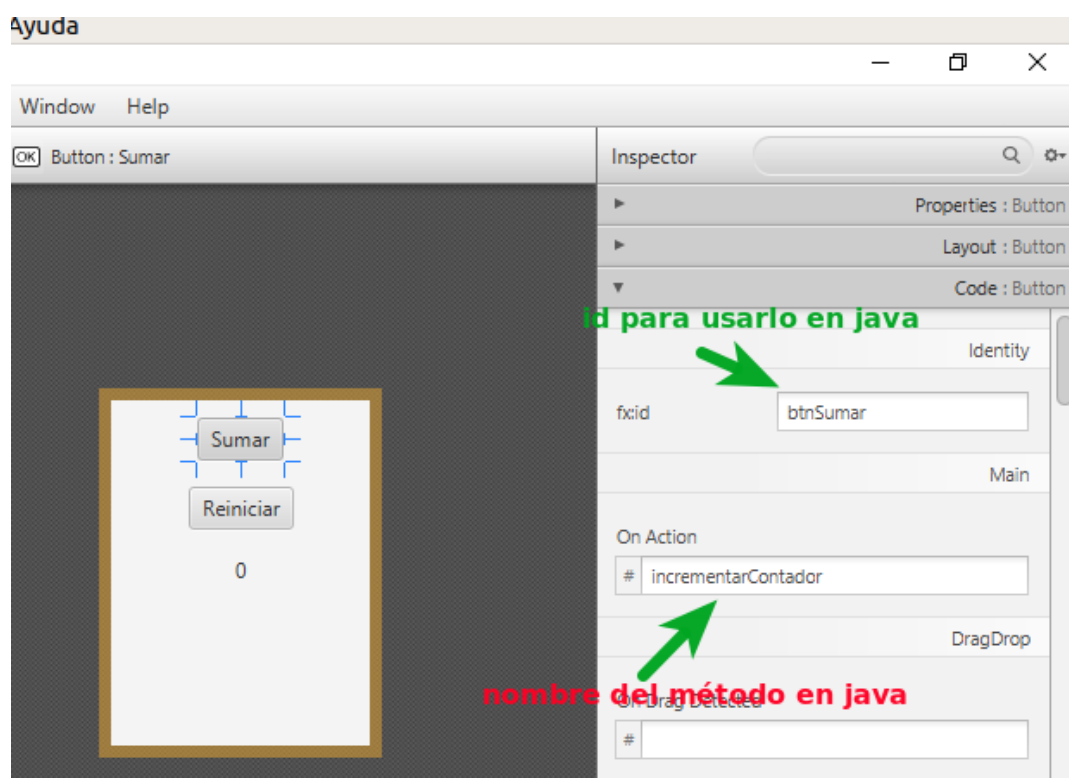


Figura 47: Id y método para la acción en SceneBuilder

6. Threads

Existe un **hilo de ejecución principal** (JavaFX Main Thread) que maneja nuestra GUI, y en el que se crean todos los nodos y componentes gráficos de nuestra aplicación. Es posible que en nuestra aplicación necesitemos realizar tareas pesadas que llegen a tardar unos cuantos segundos (consultar algo en una base de datos, en un servicio rest, algún cálculo pesado como manejo de media, ...).

En cualquier caso, hemos de evitar hacer este tipo de tareas en el hilo de ejecución principal de nuestra aplicación, ya que al hacerlo, la bloqueamos durante el tiempo que dure la tarea, generando una experiencia de usuario muy mala.

Para evitar esto, JavaFX nos proporciona la clase abstracta **Servicio**, que nos permite ejecutar tareas en otro hilo de ejecución diferente al principal, de modo que no bloqueamos la interfaz de usuario.

El vídeo con el paso a paso del ejemplo que vamos a hacer está en <https://youtu.be/M2-8jWDU63E>

El código fuente completo, lo tienes disponible en:

Vamos a tratar de hacer lo siguiente:

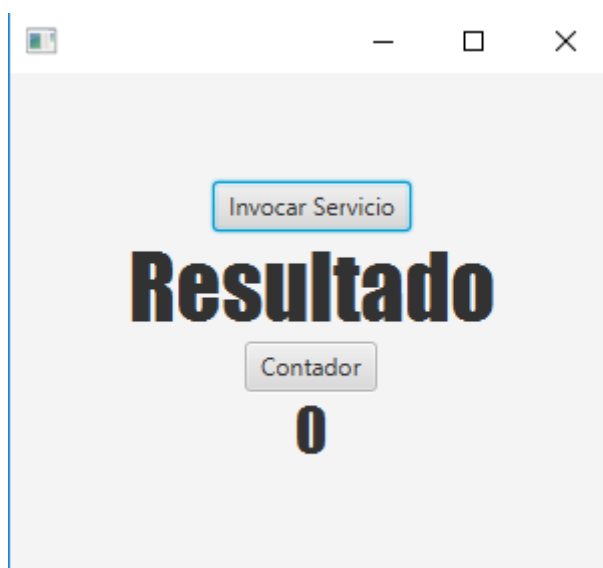


Figura 48: Ejemplo de aplicación con servicio pesado

Tenemos 2 botones:

- **Invocar servicio:** simula hacer una tarea pesada. En realidad, acaba llamando a un `Thread.sleep(3000)` para dormir 3 segundos.
- **Contador:** incrementa un contador cada vez que lo pulsamos

En primer lugar, vamos a hacer las cosas de manera incorrecta. Es decir, vamos a ejecutar esta supuesta tarea pesada en el hilo de ejecución principal, para ver cómo se bloquea. Y luego, lo implementaremos en un servicio aparte usando la clase abstracta **Service**.

El código usado en nuestra aplicación es:

- **Clase principal**, DemoServiciosApp.java -
<https://github.com/pes130/05-Servicios/blob/master/src/es/cursojavafx/demoservicios/DemoServiciosApp.java>
- **Archivo FXML**, DemoServiciosApp.fxml -
<https://github.com/pes130/05-Servicios/blob/master/resources/es/cursojavafx/demoservicios/DemoServiciosApp.fxml>
- **Controlador**, DemoServiciosControlador.java.
<https://github.com/pes130/05-Servicios/blob/master/src/es/cursojavafx/demoservicios/controller/DemoServiciosController.java>

En nuestro controlador, cuando pulsamos el botón de ‘Invocar Servicio’, ejecutamos lo siguiente:

```
@FXML  
  
void invocarServicio(ActionEvent event) {  
    System.out.println("Comienzo a ejecutar mi servicio");  
    try {  
        Thread.sleep(3000);  
        Random r = new Random();  
        int resultado = r.nextInt();  
        this.lblResultado.setText(String.valueOf(resultado));  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    System.out.println("Termino mi servicio");  
}
```

Es decir, esperamos 3 segundos, y cuando terminamos, generamos un entero aleatorio, y lo ponemos al texto de la etiqueta lblResultado. Si ejecutamos esto, al pulsar el botón ‘Invocar servicio’, nuestra interfaz queda bloqueada durante 3 segundos y no podemos hacer nada salvo esperar.

Es por eso que debemos ejecutar esta tarea en otro thread, y la manera de conseguirlo es haciendo una clase que extienda la clase Service de JavaFX. Vamos allá:

Clase **DemoServicio.java**:

```
public class DemoServicio extends Service<Integer>{

    @Override
    protected Task<Integer> createTask() {

        return new Task<Integer>() {

            @Override
            protected Integer call() throws Exception {

                System.out.println("Estoy dentro del servicio!!");
                Thread.sleep(3000);
                System.out.println("Termino ejecución del servicio");
                Random r = new Random();
                int resultado = r.nextInt();
                return resultado;

            }

        };

    }

}
```

Es decir,

- **Mi clase extiende la clase Service<T>**, siendo T la clase del resultado que devuelve mi servicio. En mi caso es un Integer, porque voy a devolver un número entero aleatorio. (Puede ser una lista de registros de una base de datos, un fichero de vídeo procesado, ...)
- La clase **Service<T>** me obliga a implementar el método **Task<Integer> createTask()**.
- El método **createTask()** tiene que devolver una **Task<T>**, una tarea en la que debemos obligatoriamente implementar el método **call()** (Task es otra clase abstracta). Recuerda que T sigue siendo el tipo a devolver. En mi caso, Integer.
- Es dentro de este **call()**, donde esperamos 3 segundos y generamos un número aleatorio.

Ya tenemos un servicio implementado, que al invocarlo, se ejecutará en otro hilo de ejecución, y al terminar nos devolverá un resultado. Pero **¿Cómo usamos esto desde el controlador?**

La respuesta no es fácil. Puede hacerse de muchas maneras: Crear un objeto de tipo `DemoServicio`, tener un singleton, si usamos SpringBoot, quizás podemos tener un Bean, ... nosotros vamos a optar por algo sencillo: Crear una instancia del objeto `DemoServicio`.

En nuestro controlador, ahora tenemos esto:

```
@FXML
void invocarServicio(ActionEvent event) {
    DemoServicio miServicio = new DemoServicio();
    miServicio.start();

    miServicio.setOnSucceeded(e -> {
        Integer resultado = miServicio.getValue();
        this.lblResultado.setText(resultado.toString());
    });
}
```

Como ves:

1. Creamos una instancia de nuestro servicio: `miServicio`.
2. Llamamos al método `miServicio.start()`
3. Añadimos un manejador de eventos para el evento `onSucceeded`. Este manejador (hemos optado por una **lambda expression**) es una funcionalidad que se ejecutará cuando nuestro servicio termine e invoque el evento `OnSucceeded`.

La cosa es que la llamada a un servicio puede lanzar distintos eventos:

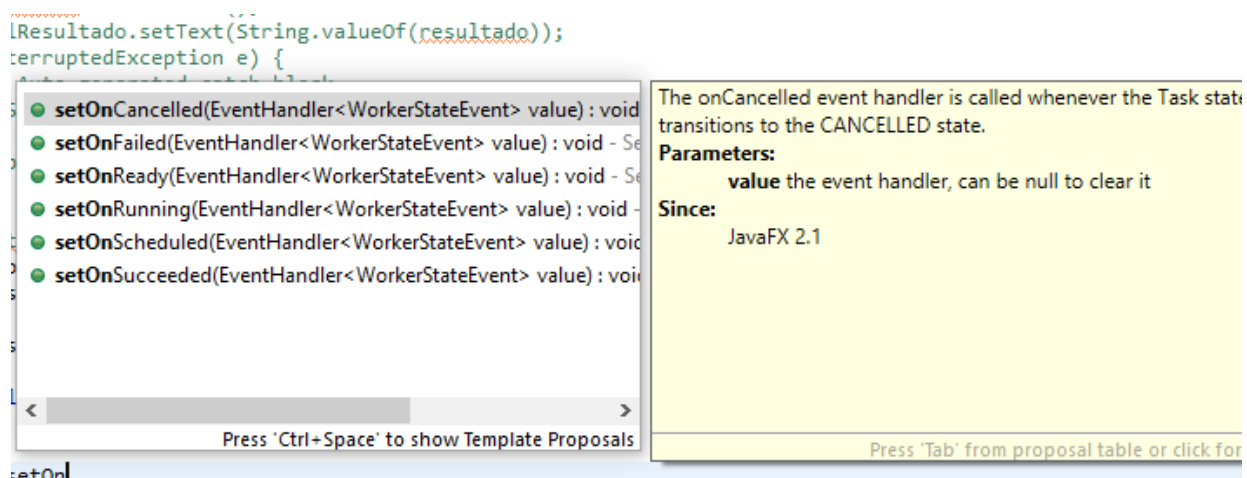


Figura 49: Distintos estados de finalización de un servicio

De este modo, podemos capturar cuando el momento en el que el servicio esté preparado (**onReady**), entre en ejecución (**onRunning**), falle (**onFailed**), ... y por supuesto, cuando acabe correctamente: **onSucceeded**.

7. Aplicando estilos con CSS

El código fuente para esta sección está de nuevo en el repositorio:

<https://github.com/pes130/02-DemoLayouts>

Para cada uno de los ejemplos hemos usado una clase java distinta que veremos en cada apartado.

JavaFX nos permite modificar la apariencia de nuestra aplicación con estilos con una nomenclatura muy similar a css. Podemos hacerlo de varias maneras:

- Programáticamente con estilos en línea
- Programáticamente con css
- Desde FXML.

Nuestra aplicación base es la siguiente:

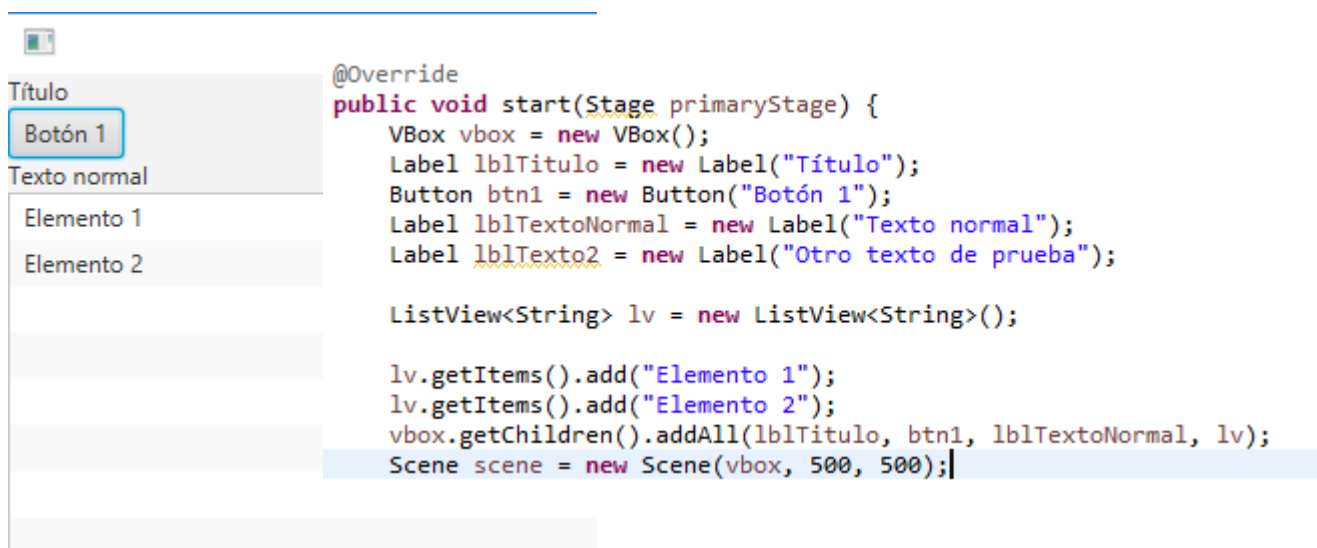


Figura 50: Aplicación sin estilo

7.1. Programáticamente con estilos en línea

Para este apartado, hemos usado:

- la clase **CssEjemplo08.java**, disponible en: <https://github.com/pes130/02-DemoLayouts/blob/master/src/es/cursojavafx/demolayouts/CssEjemplo08.java>
- La css **DarkTheme.css**, disponible en: <https://github.com/pes130/02-DemoLayouts/blob/master/resources/es/cursojavafx/demolayouts/DarkTheme.css>
- El archivo fxml **CssEjemplo09.fxml**, disponible en: <https://github.com/pes130/02-DemoLayouts/blob/master/resources/es/cursojavafx/demolayouts/CssEjemplo09.fxml>
- El controlador **CssEjemploController.java**, disponible en: <https://github.com/pes130/02-DemoLayouts/blob/master/src/es/cursojavafx/demolayouts/controller/CssEjemploController.java>

Para aplicar estilo en línea, igual que en html podemos hacer con el atributo **style=""**, usamos el método que tiene la clase **Node** **setStyle()**:

```
lblTitulo.setStyle("-fx-text-fill: red;-fx-font: 40 Impact");
```

Esto tiene como efecto:

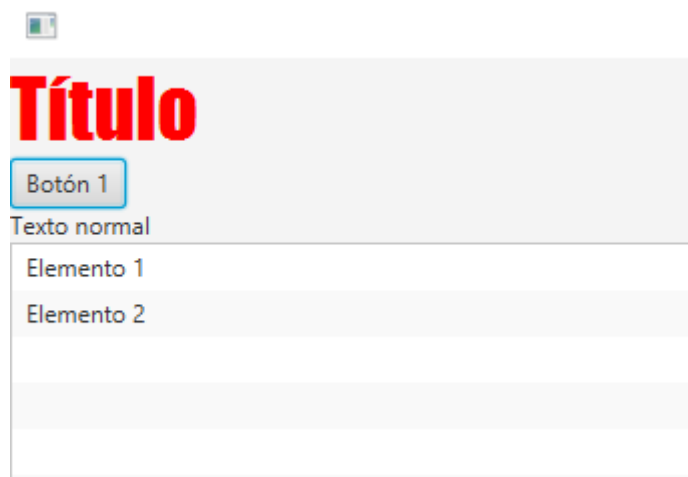


Figura 51: Aplicación de estilo en línea

Las propiedades son muy parecidas a CSS. En cualquier caso, la referencia completa puedes encontrarla aquí:

<https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/doc-files/cssref.html>

7.2. Programáticamente con una hoja de Estilos

Algo más elegante y práctico que lo anterior, al igual que ocurre en CSS, es usar una hoja de estilos externa.

Para este apartado, hemos usado los fuentes:

- la clase **CssEjemplo09.java**, disponible en:
<https://github.com/pes130/02-DemoLayouts/blob/master/src/es/cursojavafx/demolayouts/CssEjemplo09.java>
- La css **DarkTheme.css**, disponible en:
<https://github.com/pes130/02-DemoLayouts/blob/master/resources/es/cursojavafx/demolayouts/DarkTheme.css>

En nuestro caso, vamos a usar una hoja llamada DarkTheme.css que hemos colocado en nuestra carpeta **resources**:

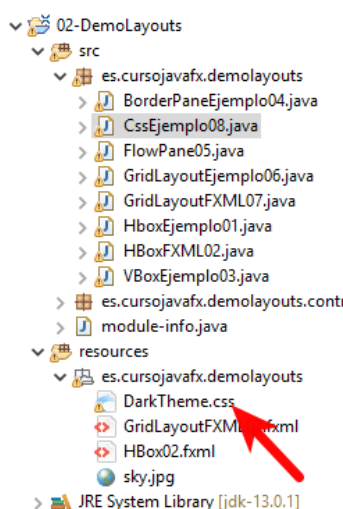


Figura 52: CSS en resources

Para cargarla en nuestra aplicación, ejecutamos:

```
// Estilos desde CSS
scene.getStylesheets().add(getClass().getResource("DarkTheme.css").toExternalForm());
```

Esta CSS es igual que las de W3C, con la salvedad de que las propiedades suelen cambiar ligeramente y llevar el prefijo -fx-.

Cada componente lleva ya incorporada una clase (como el class de html, con su nombre en minúscula). El elemento raíz, lleva la clase .root. De modo que si en la css añadimos:

```
.root {
    -fx-background-color: #111;
```

```
-fx-control-inner-background: gray;  
-fx-control-inner-background-alt: lightgray;  
}
```

El resultado es:



Figura 53: Estilo .root desde CSS

Análogamente, si añadimos el siguiente:

```
.label{  
    -fx-text-fill: #FFF;  
}
```

Tenemos:

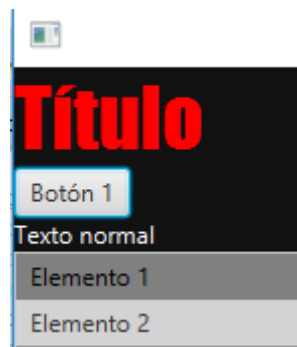


Figura 54: Label blancas

Podemos añadir clases adicionales, ejecutando con el Node deseado el método `getStyleClass().add("clase")`. Por ejemplo, para crear una negrita:

```
lblTexto2.getStyleClass().add("negrita");
```

Y en la css:

```
.negrita {  
    -fx-font-weight: bold;  
}
```

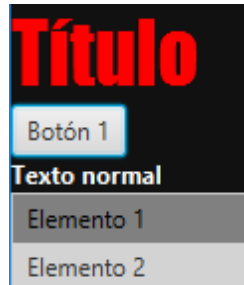


Figura 55: Añadir una clase

También podemos usar el concepto de id de HTML/CSS, pero para establecer el id, hemos de ejecutar el método setId. Por ejemplo, si queremos poner el botón btn1 de color verde:

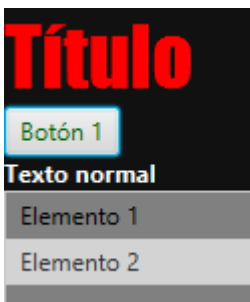
En java:

```
btn1.setId("btn1");
```

En css:

```
#btn1 {  
    -fx-text-fill: darkgreen;  
}
```

Resultado:



*Figura 56:
Usando un id*

7.3. Estilos con FXML con SceneBuilder

Vamos a usar la misma CSS que antes. Para asociar una hoja de estilos a una escena, hemos de irnos al apartado properties → StyleSheet del elemento raíz, en nuestro caso un VBox:

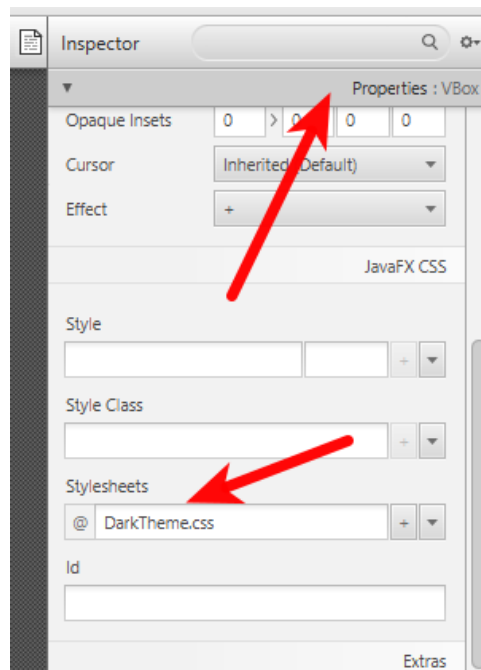


Figura 57: Asociar css desde SceneBuilder

Para asociar una nueva clase a un nodo, hacemos clic sobre el nodo en cuestión, en nuestro caso el que tiene el texto 'Texto normal', y le asociamos la clase '.negrita':

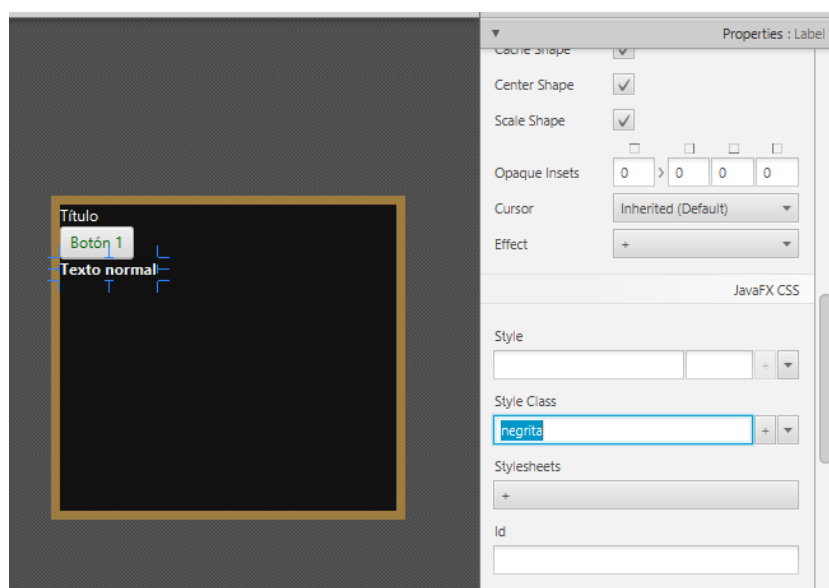


Figura 58: Asociar clase a un elemento

También podemos asociar un id a un elemento para asociarle un id de css. Con poner la propiedad Code → fx:id es suficiente:

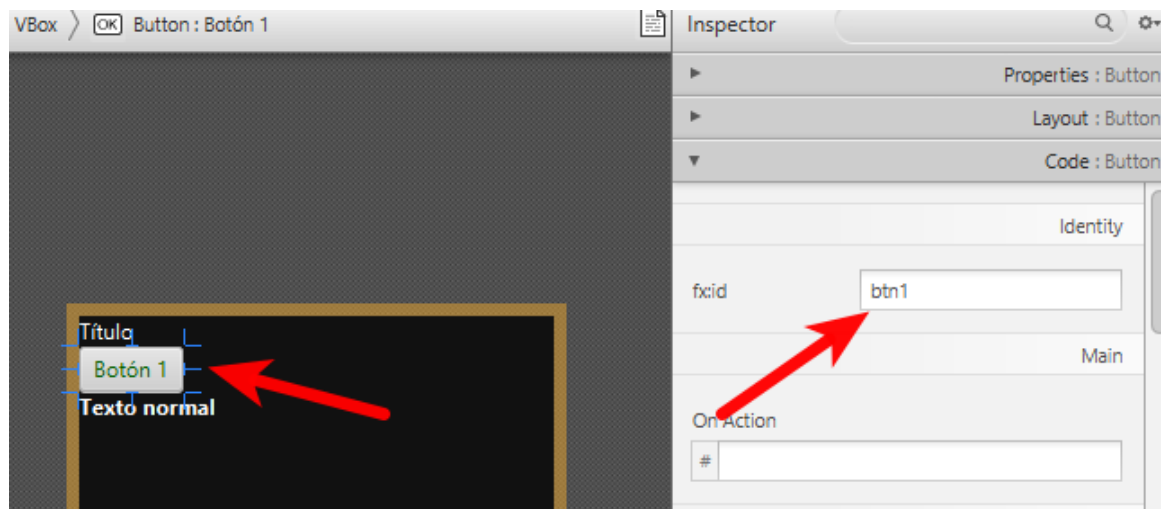


Figura 59: Asociar un id de css

8. Crear un proyecto Maven

Maven es una herramienta de gestión de proyectos de software muy poderosa que va a facilitar labores como la gestión de dependencias de software, manejar el ciclo de vida de un proyecto (compilar, guardar en nuestro repositorio, guardar en un repositorio remoto, ejecutar tests, desplegar, ...). Es **imprescindible** usar una herramienta como Maven o Gradle en cualquier empresa de desarrollo, por las incontables ventajas que ofrece.

La gestión de dependencias de software no es algo trivial. Hasta ahora, todas las aplicaciones que hemos visto sólo tenían una dependencia: javafx13. Pero una aplicación real, fácilmente puede tener cientos: jackson, java.mail, jersey, ... Resulta inviable descargarlas todas, incluirlas, y luego mantenerlas. ¿Qué ocurriría si tengo que actualizar una versión de algún jar? O Peor aún ¿Y si ese cambio de versión en un jar implica cambios en versiones de otros jar que tengo? Y peor todavía ¿Y si esta reacción es en cadena? Sólo por esta razón, debería ser obligatorio usarlo. Sin embargo, maven ofrece muchas más ventajas.

En este caso, vamos a hacer una aplicación muy rápida con Maven y JavaFX. Maven utiliza unas plantillas para generar un proyecto llamadas arquetipos (archetypes). Cuando vas a crear un proyecto maven, puedes hacer uno en blanco, o basándote en una de estas plantillas o arquetipos. Nosotros optamos por la segunda opción, y esto nos genera una aplicación con 2 FXML, 2 controladores, y una funcionalidad básica que nos permite ir de una escena a otra.

El vídeo paso a paso con la explicación lo tienes aquí: <https://youtu.be/tFTpLsT7U2k>

9. Demo de aplicación – CalculadoraJavaFX

Se trata de una aplicación que cubre algunas cosas no cubiertas hasta ahora, y que pueden resultar interesantes:

- El layout `AnchorPane`.
- Un `TableView`: Para mostrar resultados tabulados
- Una barra de menús.
- Pruebas con `ObservableList`

El código fuente de esta aplicación, puedes encontrarlo en <https://github.com/pes130/06-CalculadoraFX>

La ventana de la aplicación es la siguiente:



Figura 60: Interfaz de la aplicación Calculadora