Programación

Aplicaciones Gráficas

JavaFX

Unidad 13

Jesús Alberto Martínez
versión 0.1

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

Importante	Atención	Interesante

Unidad 13. Aplicaciones Gráficas JavaFX

1 Introducción

1.1 ¿Qué es JavaFX?

JavaFX es un conjunto de herramientas de software libre y de código abierto que te permite crear aplicaciones gráficas de escritorio y móviles con Java. Se basa en la plataforma Java y proporciona una API moderna y fácil de usar para crear interfaces de usuario ricas e interactivas.

1.2 Características principales de JavaFX:

- **Facilidad de uso:** La API de JavaFX está diseñada para ser fácil de aprender y usar, incluso para los principiantes en programación.
- **Potente:** JavaFX te permite crear interfaces de usuario complejas y sofisticadas.
- **Flexible:** JavaFX es una plataforma flexible que se puede utilizar para crear una amplia variedad de aplicaciones gráficas.
- **Portable:** Las aplicaciones JavaFX se pueden ejecutar en una amplia variedad de plataformas, incluyendo Windows, macOS, Linux y Android.
- **Gratuito y de código abierto:** JavaFX es un software libre y de código abierto, lo que significa que es gratuito de usar y modificar.

1.3 Ventajas de usar JavaFX:

- **Facilidad de desarrollo:** La API de JavaFX es intuitiva y fácil de aprender, lo que reduce el tiempo de desarrollo.
- **Alto rendimiento:** Las aplicaciones JavaFX se ejecutan con un alto rendimiento, incluso en plataformas con recursos limitados.
- **Amplia comunidad:** JavaFX tiene una comunidad activa y vibrante que proporciona soporte y recursos.
- **Flexibilidad:** JavaFX se puede utilizar para crear una amplia variedad de aplicaciones gráficas, desde aplicaciones de escritorio simples hasta juegos complejos.

1.4 Desventajas de usar JavaFX:

- **Curva de aprendizaje:** Aunque la API de JavaFX es fácil de aprender, todavía hay una curva de aprendizaje para los principiantes.
- **Menos popular que otras tecnologías GUI:** JavaFX no es tan popular como otras tecnologías GUI, como Swing o SWT.
- **Soporte limitado para algunas plataformas:** JavaFX no tiene soporte oficial para algunas plataformas, como iOS.

1.5 Casos de uso de JavaFX:

- **Aplicaciones de escritorio:** JavaFX se puede utilizar para crear una amplia variedad de aplicaciones de escritorio, como aplicaciones de productividad, juegos y herramientas de desarrollo.
- **Aplicaciones móviles:** JavaFX se puede utilizar para crear aplicaciones móviles para Android e iOS.
- **Aplicaciones web:** JavaFX se puede utilizar para crear aplicaciones web que se ejecutan en un navegador web.
- **Aplicaciones multimedia:** JavaFX se puede utilizar para crear aplicaciones multimedia que reproducen audio y video.

1.6 Comparación de JavaFX con otras tecnologías GUI:

Tecnología	Ventajas	Desventajas
JavaFX	Fácil de usar, potente, flexible, portable, gratuito y de código abierto	Curva de aprendizaje, menos popular que otras tecnologías GUI, soporte limitado para algunas plataformas
Swing	Amplia comunidad, soporte para todas las plataformas Java	API compleja, difícil de aprender, no tan flexible como JavaFX
SWT	API moderna, alto rendimiento	No es tan popular como JavaFX o Swing, soporte limitado para algunas plataformas

1.7 Elegir la tecnología GUI adecuada para tu proyecto:

La tecnología GUI adecuada para tu proyecto dependerá de tus necesidades específicas. Si necesitas una tecnología que sea fácil de aprender y usar, JavaFX es una buena opción. Si necesitas una tecnología con una amplia comunidad y soporte para todas las plataformas Java, Swing es una buena opción. Si necesitas una tecnología moderna con alto rendimiento, SWT es una buena opción.

2 Conceptos básicos

2.1 Escena y escenario en JavaFX

¿Qué es una escena?

La escena es el contenido visual de una aplicación JavaFX. Es donde se ubican todos los elementos gráficos de la aplicación, como botones, etiquetas, imágenes, etc. La escena se define mediante la clase `Scene`.

¿Qué es un escenario?

El escenario es la ventana en la que se muestra la escena. Es el contenedor principal de la aplicación JavaFX. El escenario se define mediante la clase `Stage`.

Crear una escena

Para crear una escena, se utiliza la siguiente sintaxis:

```
Scene scene = new Scene(new StackPane(), 300, 250);
```

En este código, se crea una escena con un panel `StackPane` como contenido. El tamaño de la escena se establece en 300 píxeles de ancho y 250 píxeles de alto.

Crear un escenario

Para crear un escenario, se utiliza la siguiente sintaxis:

```
Stage stage = new Stage();
```

En este código, se crea un nuevo escenario.

Mostrar el escenario

Para mostrar el escenario, se utiliza el método `show()`:

```
stage.show();
```

Cerrar el escenario

Para cerrar el escenario, se utiliza el método `close()`:

```
stage.close();
```

Ejemplo:

```
public class Main extends Application {

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

    @Override
    public void start(Stage stage) {
        Button button = new Button("Hola Mundo");
        StackPane root = new StackPane(button);
        Scene scene = new Scene(root, 300, 250);
        stage.setScene(scene);
        stage.show();
    }
}
```

Este código crea una ventana simple con un botón que muestra el texto "Hola Mundo".

Lanzar una aplicación JavaFX

Puede ser que al lanzar la aplicación nos genere un error

```
Error: faltan los componentes de JavaFX runtime y son necesarios para
ejecutar esta aplicación
```

Para solucionarlo vamos a crear una clase auxiliar que se encargue de ejecutar la aplicación JavaFX, vamos a llamarla Lanzador, pero lo correcto sería llamarla como nuestra aplicación, ya que es la clase principal que contendrá el método main, donde desde llamaremos al método launch con la clase JavaFX que queremos llamar con el método start.

```
public class Lanzador {
   public static void main(String[] args) {
         Application.launch(HolaMundo.class, args);
   }
}
```

2.2 Controles de JavaFX

Los controles son los elementos que permiten al usuario interactuar con la aplicación.

Tipos de controles

JavaFX ofrece una amplia variedad de controles, que se pueden clasificar en dos categorías principales:

- **Controles básicos:** Son los controles más comunes, como botones, etiquetas, campos de texto, etc.
- **Controles contenedores:** Son controles que se utilizan para organizar otros controles.

Controles básicos

Algunos de los controles básicos más comunes son:

- **Button:** Se utiliza para ejecutar una acción.
- **Label:** Se utiliza para mostrar texto.
- **TextField:** Se utiliza para que el usuario introduzca texto.
- **CheckBox:** Se utiliza para que el usuario seleccione una opción entre dos.
- **RadioButton:** Se utiliza para que el usuario seleccione una opción entre un conjunto de opciones.

Controles contenedores

Algunos de los controles contenedores más comunes son:

- **StackPane:** Se utiliza para apilar controles uno encima del otro.
- **VBox:** Se utiliza para organizar controles verticalmente.
- **HBox:** Se utiliza para organizar controles horizontalmente.
- **GridPane:** Se utiliza para organizar controles en una cuadrícula.

Controles avanzados

Además de los controles básicos y contenedores, JavaFX también ofrece una amplia variedad de controles avanzados, como:

- **ListView:** Se utiliza para mostrar una lista de elementos.
- **TableView:** Se utiliza para mostrar una tabla de datos.
- **TreeView:** Se utiliza para mostrar una estructura de árbol.
- **DatePicker:** Se utiliza para que el usuario seleccione una fecha.
- **TimePicker:** Se utiliza para que el usuario seleccione una hora.

Propiedades de los controles

Todos los controles tienen propiedades que se pueden modificar para cambiar su apariencia o comportamiento. Algunas de las propiedades más comunes son:

- **Text:** La propiedad `text` se utiliza para establecer el texto que se muestra en un control.
- **Font:** La propiedad `font` se utiliza para establecer la fuente del texto que se muestra en un control.
- **Color:** La propiedad `color` se utiliza para establecer el color del texto que se muestra en un control.
- **Alignment:** La propiedad `alignment` se utiliza para establecer la alineación del texto que se muestra en un control.

Eventos

Los controles pueden generar eventos cuando el usuario interactúa con ellos. Algunos de los eventos más comunes son:

Click: Se genera cuando el usuario hace clic en un control.

MouseEntered: Se genera cuando el cursor del mouse entra en un control.

MouseExited: Se genera cuando el cursor del mouse sale de un control.

Ejemplo:

```
public class Main extends Application {

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

```
    @Override
    public void start(Stage stage) {
        Button button = new Button("Hola Mundo");
        button.setOnAction(event -> System.out.println("¡Hola Mundo!"));
        StackPane root = new StackPane(button);
        Scene scene = new Scene(root, 300, 250);
        stage.setScene(scene);
        stage.show();
    }
}
```

Este código crea una ventana simple con un botón que muestra el texto "Hola Mundo". Cuando el usuario hace clic en el botón, se muestra el mensaje "¡Hola Mundo!" en la consola.

2.3 Eventos en JavaFX

Los eventos son una parte fundamental de la interacción con el usuario en JavaFX. Permiten detectar las acciones del usuario sobre los controles de la interfaz gráfica y ejecutar código en respuesta a ellas.

Tipos de eventos

JavaFX define una gran variedad de eventos, algunos de los más comunes son:

- **Eventos de ratón:**
  - `MouseEvent.MOUSE_CLICKED`: Se genera cuando el usuario hace clic en un control.
  - `MouseEvent.MOUSE_PRESSED`: Se genera cuando el usuario presiona un botón del ratón sobre un control.
  - `MouseEvent.MOUSE_RELEASED`: Se genera cuando el usuario libera un botón del ratón sobre un control.
  - `MouseEvent.MOUSE_ENTERED`: Se genera cuando el cursor del ratón entra en un control.
  - `MouseEvent.MOUSE_EXITED`: Se genera cuando el cursor del ratón sale de un control.
- **Eventos de teclado:**
  - `KeyEvent.KEY_PRESSED`: Se genera cuando el usuario presiona una tecla.
  - `KeyEvent.KEY_RELEASED`: Se genera cuando el usuario libera una tecla.
  - `KeyEvent.KEY_TYPED`: Se genera cuando el usuario introduce un carácter.
- **Eventos de foco:**
  - `FocusEvent.FOCUS_GAINED`: Se genera cuando un control recibe el foco.

- `FocusEvent.FOCUS_LOST`: Se genera cuando un control pierde el foco.

Manejo de eventos

Para responder a los eventos, se pueden utilizar los siguientes métodos:

- **`setOnXxx(EventHandler<XxxEvent> handler):`** Este método permite asociar un manejador de eventos a un control. El manejador de eventos es un objeto que implementa la interfaz `EventHandler<XxxEvent>`, donde `XxxEvent` es el tipo de evento que se desea manejar.
- **`addEventHandler(EventType<XxxEvent> eventType, EventHandler<XxxEvent> handler):`** Este método permite agregar un manejador de eventos a un control para un tipo de evento específico.

Ejemplo de manejo de eventos:

```
button.setOnAction(event -> System.out.println("¡Hola Mundo!"));
```

Este código muestra un mensaje en la consola cuando el usuario hace clic en el botón.

Eventos en profundidad:

- **Capturar y propagar eventos:** Los eventos se pueden capturar en diferentes fases de su propagación. La fase de captura se produce antes de que el evento llegue al control objetivo, mientras que la fase de propagación se produce después. Se pueden usar los métodos `addEventHandler(EventType<XxxEvent> eventType, EventHandler<XxxEvent> handler, EventFilter<XxxEvent> filter)` y `setEventHandler(EventType<XxxEvent> eventType, EventHandler<XxxEvent> handler, EventFilter<XxxEvent> filter)` para capturar y propagar eventos.
- **Consumir eventos:** Se puede consumir un evento para evitar que se propague a otros controles. Para ello, se debe llamar al método `event.consume()` dentro del manejador de eventos.
- **Eventos personalizados:** Se pueden crear eventos personalizados para detectar acciones específicas en la aplicación.

2.4 Manejo de diseños en JavaFX

El manejo de diseños es una parte fundamental de la creación de interfaces gráficas de usuario (GUI) con JavaFX. Permite organizar los controles de la interfaz de una manera atractiva y funcional.

Conceptos básicos de diseño

- **Layouts:** Los layouts son los encargados de organizar los controles en la pantalla. JavaFX ofrece una variedad de layouts predefinidos, como StackPane, VBox, HBox y GridPane.
- **Posicionamiento:** Los layouts permiten posicionar los controles en la pantalla de forma precisa. Se pueden utilizar propiedades como `top`, `bottom`, `left`, `right`, `margin` y `alignment` para controlar la posición de los controles.
- **Alineación:** Los layouts también permiten alinear los controles horizontalmente, verticalmente o en ambos sentidos.

Diseños básicos

- **StackPane:** Este layout apila los controles uno encima del otro. El control que se añade al layout en último lugar será el que se muestre en la parte superior.

```
StackPane root = new StackPane();
Button button = new Button("Hola Mundo!");
Label label = new Label("¡Hola Mundo!");

root.getChildren().addAll(button, label);
```

- **VBox:** Este layout organiza los controles verticalmente.

```
VBox vBox = new VBox();
Button button1 = new Button("Botón 1");
Button button2 = new Button("Botón 2");

vBox.getChildren().addAll(button1, button2);
```

- **HBox:** Este layout organiza los controles horizontalmente.

```
HBox hBox = new HBox();
Button button3 = new Button("Botón 3");
Button button4 = new Button("Botón 4");

hBox.getChildren().addAll(button3, button4);
```

- **GridPane:** Este layout organiza los controles en una cuadrícula.

```
GridPane gridPane = new GridPane();
Button button5 = new Button("Botón 5");
```

```
Button button6 = new Button("Botón 6");

gridPane.add(button5, 0, 0);
gridPane.add(button6, 1, 0);
```

Diseños avanzados

- **TilePane:** Este layout organiza los controles en mosaico.
- **FlowPane:** Este layout organiza los controles en un flujo, similar al texto.
- **AnchorPane:** Este layout permite anclar los controles a puntos específicos de la pantalla.
- **BorderPane:** Este layout divide la pantalla en cinco áreas: norte, sur, este, oeste y centro.

Ejemplo de TilePane:

```
TilePane tilePane = new TilePane();
Button button7 = new Button("Botón 7");
Button button8 = new Button("Botón 8");
Button button9 = new Button("Botón 9");

tilePane.getChildren().addAll(button7, button8, button9);
```

Ejemplo de FlowPane:

```
FlowPane flowPane = new FlowPane();
Button button10 = new Button("Botón 10");
Button button11 = new Button("Botón 11");
Button button12 = new Button("Botón 12");

flowPane.getChildren().addAll(button10, button11, button12);
```

Ejemplo de AnchorPane:

```
AnchorPane anchorPane = new AnchorPane();
Button button13 = new Button("Botón 13");
Label label2 = new Label("¡Hola Mundo!");

AnchorPane.setTopAnchor(button13, 10.0);
AnchorPane.setLeftAnchor(button13, 10.0);
AnchorPane.setBottomAnchor(label2, 10.0);
AnchorPane.setRightAnchor(label2, 10.0);

anchorPane.getChildren().addAll(button13, label2);
```

Ejemplo de BorderPane:

```
BorderPane borderPane = new BorderPane();
Button button14 = new Button("Botón 14");
Label label3 = new Label("¡Hola Mundo!");

borderPane.setTop(button14);
borderPane.setCenter(label3);
```

Responsive layouts

Los responsive layouts son layouts que se adaptan automáticamente al tamaño de la pantalla. Esto es importante para que las interfaces gráficas se vean bien en diferentes dispositivos, como ordenadores de escritorio, portátiles, tablets y teléfonos móviles.

JavaFX ofrece algunas herramientas para crear responsive layouts, como las siguientes:

- **SceneBuilder:** SceneBuilder es una herramienta gráfica que permite crear interfaces gráficas de usuario con JavaFX. SceneBuilder incluye una función para crear responsive layouts.
- **Media queries:** Las media queries permiten detectar el tamaño de la pantalla y aplicar diferentes estilos en función del tamaño de la pantalla.

Ejemplo de responsive layout:

```
@Override
public void start(Stage stage) {
    Scene scene = new Scene(new StackPane(), 300, 250);
    // ...
    stage.setScene(scene);
    stage.show();
}
@Override
public void init() {
    // Detectar el tamaño de la pantalla
    MediaQuery mediaQuery = new MediaQuery("(max-width: 480px)");

    // Aplicar diferentes estilos en función del tamaño de la pantalla
    mediaQuery.addListener((change) -> {
        if (change.matches()) {
            // Aplicar estilo para pantallas pequeñas
        } else {
            // Aplicar estilo para pantallas grandes
        }
    });
}
```

Consejos para crear responsive layouts:

- **Utiliza layouts flexibles:** Algunos layouts, como GridPane, son más flexibles que otros y se adaptan mejor a diferentes tamaños de pantalla.
- **Utiliza unidades relativas:** En lugar de utilizar unidades absolutas como píxeles, utiliza unidades relativas como porcentajes para que los elementos se adapten al tamaño de la pantalla.
- **Utiliza MediaQueries:** Las MediaQueries te permiten aplicar diferentes estilos en función del tamaño de la pantalla.
- **Prueba tu layout en diferentes dispositivos:** Es importante probar tu layout en diferentes dispositivos para asegurarte de que se ve bien en todos ellos.

2.5 Ficheros FXML

Los ficheros FXML (FX Markup Language) son una forma de definir interfaces gráficas de usuario (GUI) en JavaFX de forma declarativa. Esto significa que se pueden definir los elementos de la interfaz gráfica utilizando un lenguaje de marcado similar a XML, en lugar de hacerlo de forma programática utilizando código Java.

Ventajas de usar FXML:

- **Facilidad de uso:** FXML es un lenguaje sencillo y fácil de aprender, lo que lo hace ideal para diseñadores y desarrolladores que no tienen mucha experiencia con Java.
- **Separación de preocupaciones:** FXML permite separar la lógica de la interfaz gráfica de la presentación. Esto facilita la colaboración entre diseñadores y desarrolladores.
- **Reutilización de código:** Los ficheros FXML se pueden reutilizar en diferentes aplicaciones.
- **Diseño WYSIWYG:** Se pueden usar herramientas como SceneBuilder para crear interfaces gráficas de usuario de forma visual, lo que facilita el diseño y la prototipación.

Estructura de un fichero FXML:

Un fichero FXML tiene la siguiente estructura:

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

<fx:root xmlns:fx="http://javafx.com/fxml/1/elements">
  </fx:root>
```

Elementos básicos de FXML:

- **`fx:root`:** Este elemento define la raíz de la interfaz gráfica.
- **`fx:id`:** Este atributo se usa para asignar un identificador a un elemento.
- **`fx:type`:** Este atributo se usa para especificar el tipo de un elemento.
- **`fx:children`:** Este elemento se usa para contener los elementos secundarios de un elemento.

Ejemplo de un fichero FXML simple:

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

<fx:root xmlns:fx="http://javafx.com/fxml/1/elements">
  <Button text="Hola Mundo!" />
</fx:root>
```

Controles FXML:

FXML soporta una amplia variedad de controles, incluyendo los siguientes:

- Botones
- Etiquetas
- Campos de texto
- Casillas de verificación
- Botones de radio
- Listas
- Tablas
- Árboles

Enlace de datos:

FXML permite enlazar datos entre los elementos de la interfaz gráfica y las variables Java. Esto facilita la actualización de la interfaz gráfica en función de los cambios en los datos.

Ejemplo de enlace de datos:

```
<?xml version="1.0" encoding="UTF-8"?>
<fx:root xmlns:fx="http://javafx.com/fxml/1/elements">
  <Label text="${nombre}" />
</fx:root>
```

Controladores FXML:

Los controladores FXML son clases Java que se utilizan para controlar la lógica de la interfaz gráfica. Se pueden usar para manejar eventos, acceder a datos y realizar otras tareas.

Ejemplo de un controlador FXML:

```
public class MiControlador {

  @FXML
  private Button button;

  @FXML
  public void initialize() {
    button.setOnAction(event -> System.out.println("¡Hola Mundo!"));
  }
}
```

SceneBuilder:

SceneBuilder es una herramienta gráfica que permite crear interfaces gráficas de usuario con JavaFX de forma visual. SceneBuilder facilita el diseño y la prototipación de interfaces gráficas.

3 Ejemplo completo. LibretaDirecciones

Una vez que ya hemos creado nuestras primeras aplicaciones JavaFX, vamos a seguir un tutorial para crear una aplicación algo más compleja, a la que añadiremos funcionalidades en temas posteriores.

Vamos a seguir un tutorial de la página https://code.makery.ch/ realizandole pequeñas modificaciones a las ya realizadas por Jairo García Rincón, que también tomamos de base.

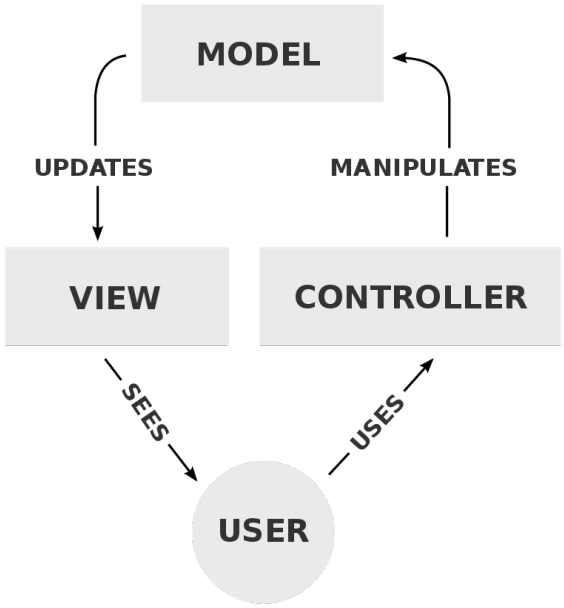
3.1 Creación del proyecto

Nuestro entorno debe estar ya completamente configurado, con Scene Builder funcionando y comprendiendo su funcionamiento básico.

Creamos un nuevo proyecto JavaFX, el identificador del grupo en el ejemplo va a ser iesthiar, y el identificador del proyecto libretadirecciones. Y eliminamos los ficheros java y fxml que nos crea el proyecto.

Desde el principio vamos a seguir buenos principios de diseño de software. Algunos de estos principios se traducen en el uso de la arquitectura denominada **Modelo-Vista-Controlador** (MVC). Esta arquitectura

promueve la división de nuestro código en tres apartados claramente definidos, uno por cada elemento de la arquitectura.



- Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio).

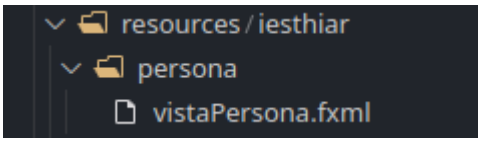
- Controlador: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta el 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'.
- Vista: Presenta el 'modelo' (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario), por tanto requiere de dicho 'modelo' la información que debe representar como salida.

Tradicionalmente se han separado los tipos de clase creando paquetes separados para las clases del modelo, del control y de la vista. Hay otro enfoque, que es la separación en paquetes por elementos, y dentro de cada elemento su vista, su control y su modelo, éste es el enfoque que vamos a seguir.

3.2 Diseño de la interfaz

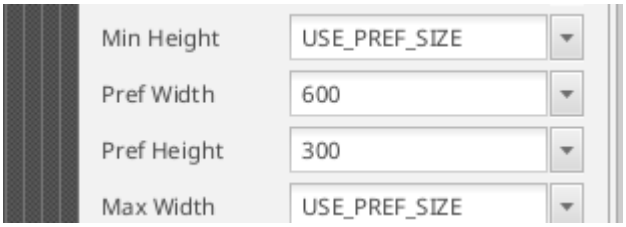
Creación del archivo fxml de diseño

Crea un fichero fxml en un nuevo directorio persona, dentro del directorio principal del apartado resources, con el nombre vistaPersona.fxml y ábrelo con la aplicación Scene Builder.



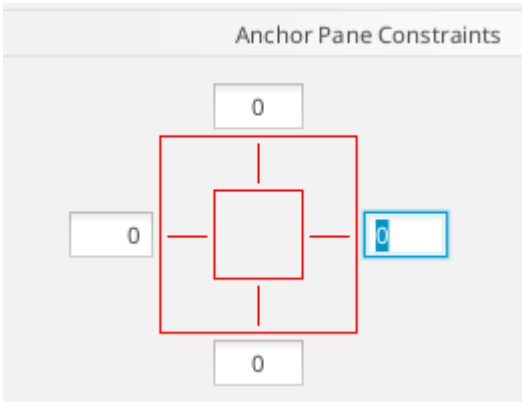
Diseño con Scene Builder

Una vez abierto Scene Builder, seleccionando el AnchorPane en la jerarquía de la izquierda, ajusta el tamaño en el apartado layout (a la derecha).

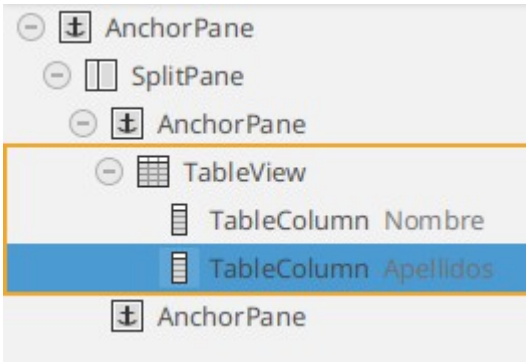


Añade un SplitPane (horizontal) arrastrándolo desde la librería (Library) al área principal de edición. Haz clic en el Split Pane y, desde el menú Modify, haz clic en Fit to Parent, para ajustarlo a la ventana (o pulsa Ctrl + K).

Añade una TableView (en Controls) y arrástralo al lado izquierdo del SplitPane. En el lado derecho, ajusta los cuatro AnchorPane Constraints de la TableView a 0 para que la TableView esté anclada a los bordes y "siga" el posible redimensionamiento de la ventana padre (Puedes hacer clic en el menú Preview -> Show Preview in window o pulsar Ctrl + P para comprobarlo).



Haciendo clic en cada TableColumn de la TableView, cambia los títulos C1 y C2 por Nombre y Apellidos, respectivamente.



Selecciona la TableView y en Properties (lado derecho) selecciona Column Resize Policy: constrained-resize para que las columnas utilicen todo el espacio derecho disponible. El resultado debería ser similar al siguiente:

Crea ahora un Label (Controls) en el lado derecho del SplitPane con el texto "Detalles". Ajusta sus Properties a tu gusto.

Añade un GridPane (Containers) debajo del Label y ajusta su apariencia usando anclajes (similares a los que usamos en la TableView, aunque en este caso con valores arriba a 30 y derecha e izquierda a 10.

Modifica el GridPane para que tenga 6 filas (haciendo clic derecho en un número de fila se pueden añadir nuevas con Add Row Above o Add Row Before) con las etiquetas Nombre, Apellidos, Dirección, Ciudad, Código Postal, Fecha de nacimiento. Añade también otras 6 etiquetas en la segunda columna.

Añade 3 Button (Controls) en la parte inferior derecha con los textos "Nuevo", "Editar" y "Borrar". Para ajustarlos más cómodamente de forma global, selecciona los 3 y con el botón derecho haz clic en Wrap in -> HBox. Ahora puedes ajustar el Layout del HBox para que el Spacing entre botones sea de 10 y los AnchorPane derecho e inferior sean 10.

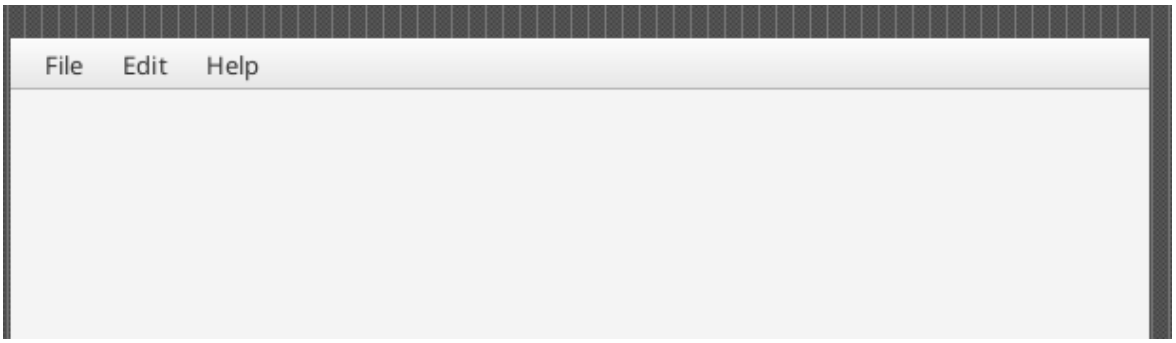
El resultado debe ser similar al siguiente cambiado el texto de los botones:

La vista principal

Con esto ya tendríamos creada nuestra VistaPersona, pero para nuestra aplicación necesitamos ademas una vista principal, que crearemos en un nuevo FXML llamado vistaPrincipal, en el directorio del paquete principal de resources.

Una vez abierto en Scene Builder, en este caso utilizaremos un BorderPane como contenedor principal, que puedes añadir arrastrando desde Containers en la parte superior izquierda.

El tamaño preferido del BorderPane (En Layout) debe ser 600x400 y de momento solo le añadiremos una barra de menú superior (MenuBar en Controls) en la parte superior. El resultado debería ser similar a este:



La clase principal

Vamos a crear nuestra clase principal con el nombre LibretaDirecciones dentro de nuestro paquete principal.

La clase generada (LibretaDirecciones.java) extiende a la clase Application y contiene dos métodos. Esta es la estructura básica que necesitamos para ejecutar una aplicación JavaFX. La parte más importante para nosotros es el método start(Stage primaryStage). Este método es invocado automáticamente cuando la aplicación es lanzada desde el método main.

Como puedes ver, el método start(...) toma un Stage como parámetro. El gráfico siguiente muestra la estructura de cualquier aplicación JavaFX:

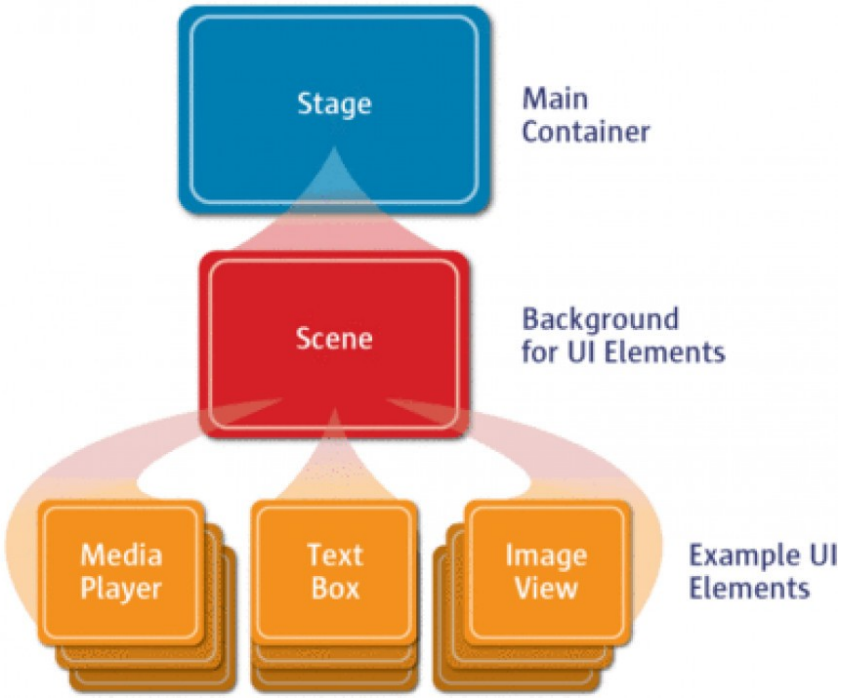


Figura 1: Estructura escena, escenario. Fuente Oracle

El funcionamiento es similar al de una obra de teatro: El Stage (escenario) es el contenedor principal, normalmente una ventana con borde y los típicos botones para maximizar, minimizar o cerrar la ventana. Dentro del Stage se puede añadir una Scene (escena), la cual puede cambiarse dinámicamente por otra Scene. Dentro de un Scene se añaden los nodos JavaFX, tales como AnchorPane, TextBox, MediaPlayer, ImageView, etc.

Para tener más información puedes consultar Working with the JavaFX Scene Graph.

Por defecto, la clase principal creada invoca una vista sencilla con un botón que permite decir "Hola" al hacer clic en él. Vamos a borrar todo el código de la clase y sustituirlo por el siguiente, que como puedes

ver está comentado indicando a qué corresponde cada apartado para que puedas comprender en detalle su funcionamiento:

```
package iesthiar;

import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class LibretaDirecciones extends Application {

    private Stage escenarioPrincipal;
    private BorderPane contenedorPrincipal;

    @Override
    public void start(Stage escenarioPrincipal) {
        // Necesario para cambiar las escenas
        this.escenarioPrincipal = escenarioPrincipal;
        // Establezco el título
        this.escenarioPrincipal.setTitle("Libreta de direcciones");
        // inicialización del contenedor principal
        initContenedorPrincipal();
        // muestro la vista persona
        mostrarVistaPersona();
    }

    /**
     * Initializes the root layout.
     */
    public void initContenedorPrincipal() {
        try {
            // Carga el contenedor principal desde el fxml.
            FXMLLoader loader = new FXMLLoader();

loader.setLocation(LibretaDirecciones.class.getResource("vistaPrincipal.fxml"));
            contenedorPrincipal = (BorderPane) loader.load();

            // Muestra la escena del contenedor principal
            Scene scene = new Scene(contenedorPrincipal);
            escenarioPrincipal.setScene(scene);
            escenarioPrincipal.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Carga y muestra la escena que
```

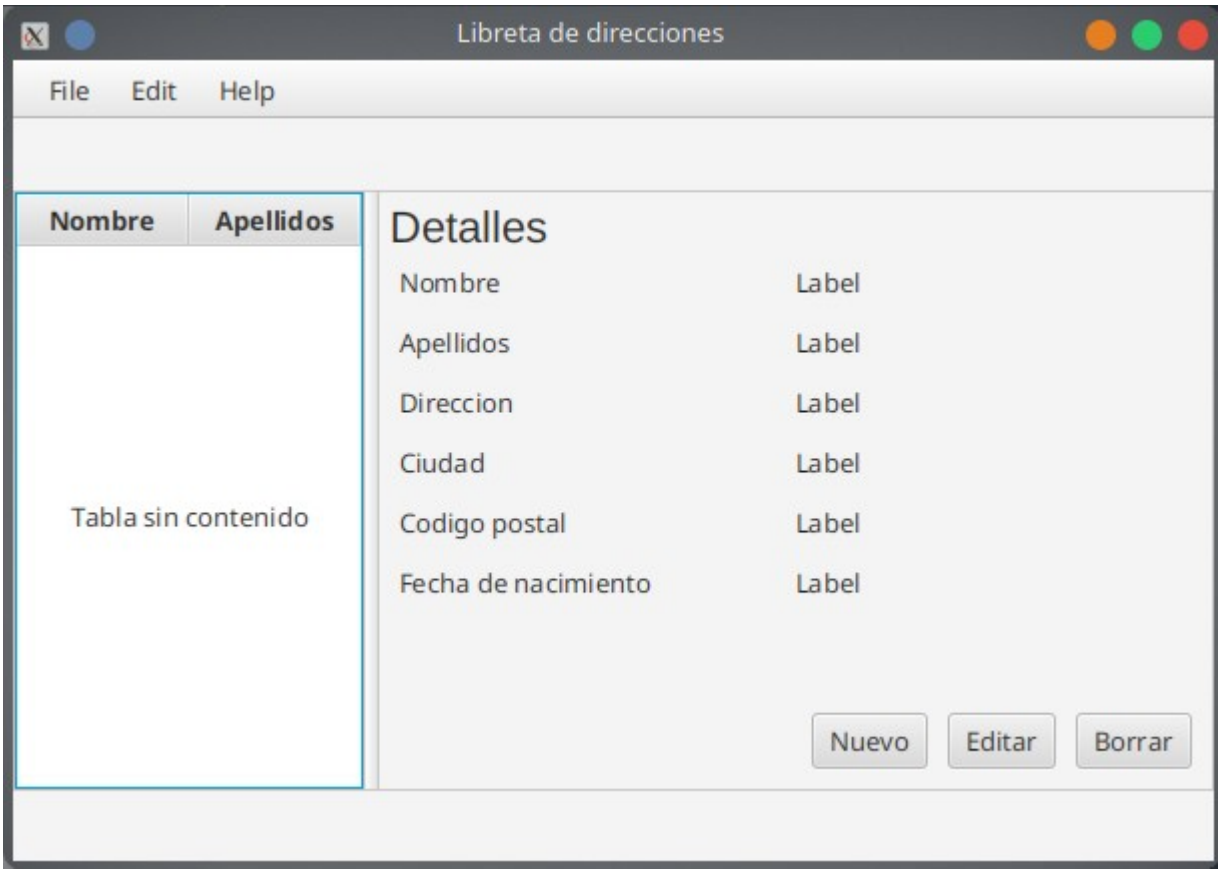
```
50       */
51      public void mostrarVistaPersona() {
52          try {
54              FXMLLoader loader = new FXMLLoader();
55
loader.setLocation(LibretaDirecciones.class.getResource("persona/vistaPersona.fxml"));
56              AnchorPane personOverview = (AnchorPane) loader.load();
57
58              // Añade la vista al centro del contenedor principal
59              contenedorPrincipal.setCenter(personOverview);
60          } catch (IOException e) {
61              e.printStackTrace();
62          }
63      }
64
65        /**
66         * Devuelve el escenario principal.
67         * @return
68         */
69        public Stage getEscenarioPrincipal() {
70               return escenarioPrincipal;
71        }
72
73      public static void main(String[] args) {
74          launch(args);
75      }
76  }
```

Si ejecutamos ahora la aplicación deberíamos ver algo parecido a la siguiente salida (en caso de no obtener esta salida o mostrar errores, deberemos encontrar el problema)

Si JavaFX no encuentra un archivo `fxml` puedes obtener el siguiente mensaje de error

```
1  java.lang.IllegalStateException: Location is not set.
```

Para resolverlo comprueba otra vez que no hayas escrito mal el nombre de tus archivos `fxml`.



3.3 Modelo y TableView

- Creación de una clase para el **modelo**
- Uso del modelo en una **ObservableList**
- Visualización del modelo mediante **TableView** y **Controladores**

El modelo Persona

Necesitamos un modelo para almacenar toda la información relativa a los contactos de la libreta. Para ello, crearemos una nueva clase Java llamada Persona dentro de un paquete persona.

El código para la clase Persona se detalla a continuación, y los aspectos más relevantes del mismo serían:

- Con JavaFX es habitual usar Propiedades para todos los atributos de una clase usada como modelo, ya que nos van a permitir, entre otras cosas, mantener sincronizados la vista y los datos cuando los valores de las variables cambien. Más información sobre Properties and Binding.

- LocalDate, el tipo que usamos para especificar la fecha de nacimiento. Más información sobre Standard Calendar.

```java
package iesthiar.persona;

import java.time.LocalDate;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Persona {
	private final StringProperty nombre;
	private final StringProperty apellidos;
	private final StringProperty direccion;
	private final IntegerProperty codigoPostal;
	private final StringProperty ciudad;
	private final ObjectProperty<LocalDate> fechaNacimiento;

	/**
	 * Constructor por defecto
	 */
	public Persona() {
		this(null, null);
	}

	/**
	 * Cosntructor con algunos datos.
	 *
	 * @param nombre
	 * @param apellidos
	 */
	public Persona(String nombre, String apellidos) {
		this.nombre = new SimpleStringProperty(nombre);
		this.apellidos = new SimpleStringProperty(apellidos);

		// datos iniciales para pruebas
		this.direccion = new SimpleStringProperty("tu calle");
		this.codigoPostal = new SimpleIntegerProperty(3190);
		this.ciudad = new SimpleStringProperty("Pilar de la horadada");
		this.fechaNacimiento = new
SimpleObjectProperty<LocalDate>(LocalDate.of(2002, 2, 20));
	}

	public String getNombre() {
		return nombre.get();
	}

	public void setNombre(String nombre) {
		this.nombre.set(nombre);
```

```
50        }
51
52        public StringProperty nombreProperty() {
53                return nombre;
54        }
55
56        public String getApellidos() {
57                return apellidos.get();
58        }
59
60        public void setApellidos(String apellidos) {
61                this.apellidos.set(apellidos);
62        }
63
64        public StringProperty apellidosProperty() {
65                return apellidos;
66        }
67
68        public String getDireccion() {
69                return direccion.get();
70        }
71
72        public void setDireccion(String direccion) {
73                this.direccion.set(direccion);
74        }
75
76        public StringProperty direccionProperty() {
77                return direccion;
78        }
79
80        public int getCodigoPostal() {
81                return codigoPostal.get();
82        }
83
84        public void setCodigoPostal(int codigoPostal) {
85                this.codigoPostal.set(codigoPostal);
86        }
87
88        public IntegerProperty codigoPostalProperty() {
89                return codigoPostal;
90        }
91
92        public String getCiudad() {
93                return ciudad.get();
94        }
95
96        public void setCiudad(String ciudad) {
97                this.ciudad.set(ciudad);
98        }
99
100       public StringProperty ciudadProperty() {
101               return ciudad;
102       }
```

```
103
104       public LocalDate getFechaNacimiento() {
105             return fechaNacimiento.get();
106       }
107
108       public void setFechaNacimiento(LocalDate fechaNacimiento) {
109             this.fechaNacimiento.set(fechaNacimiento);
110       }
111
112       public ObjectProperty<LocalDate> fechaNacimientoProperty() {
113             return fechaNacimiento;
114       }
115 }
```

La lista de personas

El objetivo de nuestro proyecto era almacenar y gestionar una lista de personas, con lo que vamos a crear una lista de objetos de tipo Persona dentro de la clase LibretaDirecciones a la que luego podremos acceder desde cualquiera de los otros controladores.

Lista observable (ObservableList)

Para poder pasar y mantener sincronizados los datos de la lista de personas en las clases gráficas de JavaFX, utilizamos las denominadas clases de colección de JavaFX, de las cuales usaremos una ObservableList.

Se ha modificado el código de la clase principal LibretaDirecciones, de modo que ahora incluya nuestra variable ObservableList y un método de consulta (get) público. Además, hemos añadido un constructor para crear datos de ejemplo:

```
1     /*
2      * Otros atributos
3      */
4
5  ivate ObservableList<Persona> datosPersona = FXCollections.observableArrayList();
6
7     /*
8      * Constructor inicializando con datos de ejemplo
9      */
10   public LibretaDirecciones(){
11        datosPersona.add(new Persona("Aitor","Tilla"));
12        datosPersona.add(new Persona("Paco","Jones"));
13        datosPersona.add(new Persona("Victor","Tazo"));
14        datosPersona.add(new Persona("Aquiles","Castro"));
15        datosPersona.add(new Persona("Elton","Tito"));
16        datosPersona.add(new Persona("Aitor","Menta"));
17   }
18
19   public ObservableList<Persona> getDatosPersona(){
```

```
20          return datosPersona;
21      }
22
23      // ... EL RESTO DE LA CLASE ...
```

El controlador para la vista de personas

Por último, tenemos que añadir los datos a nuestra tabla de vistaPersona.fxml, y para ello crearemos un controlador mediante una clase Java llamada VistaPersonaController dentro del paquete persona.

El código de la clase VistaPersonaController se muestra a continuación, y en él hay que destacar ciertos detalles:

- Los campos y métodos donde el archivo fxml necesita acceso deben ser anotados con @FXML. En realidad, sólo si son privados, pero es mejor tenerlos privados y marcarlos con la anotación.
- El método initialize() es invocado automáticamente tras cargar el fxml. En ese momento, todos los atributos FXML deberían ya haber sido inicializados.
- El método setCellValueFactory(...) que aplicamos sobre las columnas de la tabla se usa para determinar qué atributos de la clase Persona deben ser usados para cada columna particular. La flecha -> indica que estamos usando una característica desde Java 8 denominada Lambdas.
- Acuérdate siempre de importar javafx, NO AWT ó Swing.

```
1  package iesthiar.persona;
2
3  import java.net.URL;
4  import java.util.ResourceBundle;
5
6  import iesthiar.LibretaDirecciones;
7  import iesthiar.modelo.Persona;
8  import javafx.fxml.FXML;
9  import javafx.fxml.Initializable;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.TableColumn;
12 import javafx.scene.control.TableView;
13
14 public class VistaPersonaControlador implements Initializable{
15     @FXML
16      private TableView<Persona> personaTabla;
17      @FXML
18      private TableColumn<Persona, String> nombreColumna;
19      @FXML
20      private TableColumn<Persona, String> apellidosColumna;
21
22      @FXML
23      private Label nombreEtiqueta;
24      @FXML
```

```
25      private Label apellidosEtiqueta;
26      @FXML
27      private Label direccionEtiqueta;
28      @FXML
29      private Label codigoPostalEtiqueta;
30      @FXML
31      private Label ciudadEtiqueta;
32      @FXML
33      private Label fechaNacimientoEtiqueta;
34
35      // Reference to the main application.
36      private LibretaDirecciones libretaDirecciones;
37
38      /**
39       * El constructor es llamado antes que el método initialize()
40       */
41      public VistaPersonaControlador() {
42      }
43
44      /**
45       * Inicializa la clase controladora. Este método se llama automáticamente
46       * después de la carga del fichero fxml.
47       * Si la clase implementa Initializable obliga a crear el método
48       */
49      @Override
50      public void initialize(URL arg0, ResourceBundle arg1) {
51          nombreColumna.setCellValueFactory(cellData ->
cellData.getValue().nombreProperty());
52          apellidosColumna.setCellValueFactory(cellData ->
cellData.getValue().apellidosProperty());
53      }
54
55      /**
56       * Se llama desde la aplicación principal, para tener una referencia a si
misma
57       *
58       * @param libretaDirecciones
59       */
60      public void setLibretaDirecciones(LibretaDirecciones libretaD) {
61          this.libretaDirecciones = libretaD;
62
63          // Add observable list data to the table
64          personaTabla.setItems(libretaD.getDatosPersona());
65      }
66  }
```

La conexión de LibretaDirecciones con VistaPersonaController

Debemos invocar el método setLibretaDirecciones desde la clase LibretaDirecciones, de modo que podamos acceder al objeto LibretaDirecciones y, entre otras cosas, obtener la lista de Persona. Para ello,

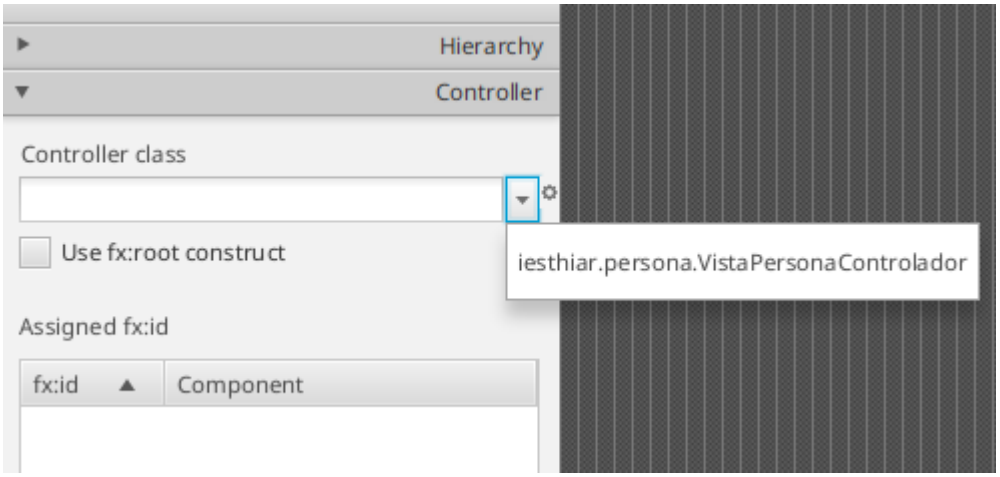
debemos modificar el método muestraVistaPersona() en LibretaDirecciones para que incluya dicho acceso (líneas 10 y 11):

```
1  public void mostrarVistaPersona() {
2      try {
3          // Carga la vista de persona.
4          FXMLLoader loader = new FXMLLoader();
5
loader.setLocation(LibretaDirecciones.class.getResource("vistaPersona.fxml"));
6          AnchorPane personOverview = (AnchorPane) loader.load();
7
8          // Añade la vista al centro del contenedor principal
9          contenedorPrincipal.setCenter(personOverview);
10         VistaPersonaControlador controlador=loader.getController();
11         controlador.setLibretaDirecciones(this);
12     } catch (IOException e) {
13         e.printStackTrace();
14     }
15 }
```

Vincular la vista al controlador

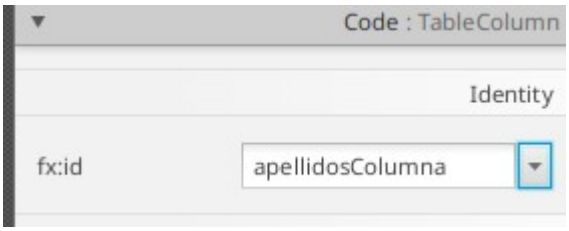
Para finalizar, debemos indicar a VistaPersona.fxml mediante Scene Builder cuál es su controlador y asociar los diferentes elementos de la TableView y del GridPane con las variables de VistaPersonaController:

- Selecciona el controlador en el desplegable Controller class de la sección Controller (lado izquierdo) de vistaPersona.fxml.

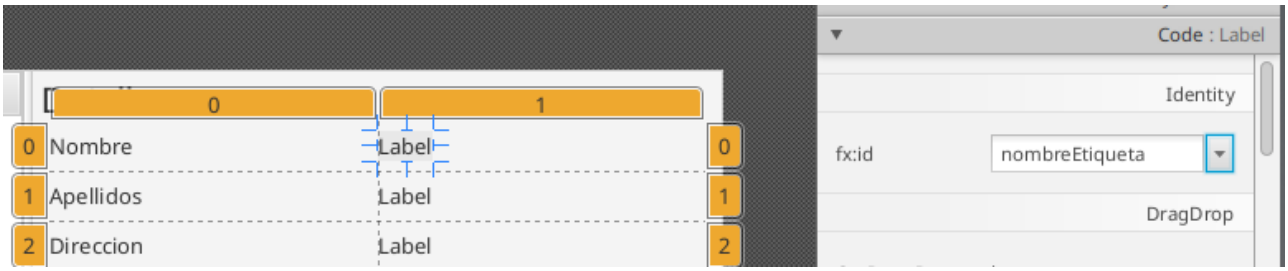


- Selecciona TableView en la sección Hierarchy y, en la pestaña Code (lado derecho), selecciona como fx:id personaTabla.

- Haz lo mismo para las columnas, seleccionando nombreColumna y apellidosColumna como sus fx:id, respectivamente.



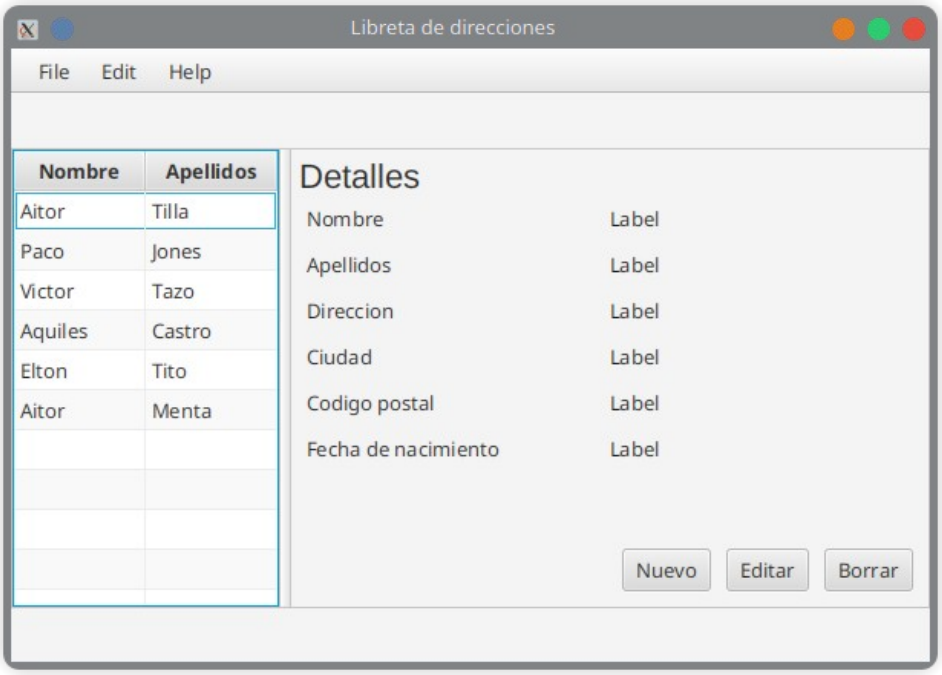
- Para cada etiqueta o Label de la segunda columna, selecciona el fx:id correspondiente.



Si hemos hecho todo correctamente, al lanzar la aplicación (recuerda ejecutar Clean and Build a tu proyecto antes) debemos encontrar una vista similar a la mostrada al inicio del presente capítulo.

Ejecutamos la aplicación

Y deberíamos obtener algo parecido a esto



3.4 Interacción con el usuario

- Respuesta a cambios en la selección dentro de la tabla.
- Añade funcionalidad de los botones añadir, editar, y borrar.
- Crear un diálogo emergente (popup dialog) a medida para editar un contacto.
- Validación de la entrada del usuario.

Respuesta a cambios en la selección de la Tabla

Todavía no hemos usado la parte derecha de la interfaz de nuestra aplicación. La intención es usar esta parte para mostrar los detalles de la persona seleccionada por el usuario en la tabla.

En primer lugar vamos a añadir un nuevo método dentro de VistaPersonaControlador que nos ayude a rellenar las etiquetas con los datos de una sola persona.

Crea un método llamado mostrarDetallesPersona(`Persona persona`). Este método recorrerá todas las etiquetas y establecerá el texto con detalles de la persona usando `setText(...)`. Si en vez de una instancia de `Persona` se pasa `null` entonces las etiquetas deben ser borradas.

```
    /**
     * Rellena todos los textos para mostrar los detalles de una persona
     * Si la persona es null, los textos se borran
     *
     * @param persona la persona o null
     */
    private void mostrarDetallesPersona(Persona persona) {
        if (persona != null) {
            // Fill the labels with info from the person object.
            nombreEtiqueta.setText(persona.getNombre());
            apellidosEtiqueta.setText(persona.getApellidos());
            direccionEtiqueta.setText(persona.getDireccion());

codigoPostalEtiqueta.setText(Integer.toString(persona.getCodigoPostal()));
            ciudadEtiqueta.setText(persona.getCiudad());

            // TODO: Tenemos que convertir la fecha de necimiento a texto
            // fechaNacimientoEtiqueta.setText(...);
        } else {
            // Person is null, remove all the text.
            nombreEtiqueta.setText("");
            apellidosEtiqueta.setText("");
            direccionEtiqueta.setText("");
            codigoPostalEtiqueta.setText("");
            ciudadEtiqueta.setText("");
            fechaNacimientoEtiqueta.setText("");
        }
    }
```

Trabajar con fechas

Dado que nuestra propiedad fechaNacimiento es de tipo LocalDate, no podemos trabajar con ella directamente, sino que tenemos que realizar una conversión de LocalDate a String.

No obstante, ya que en muchos sitios (y en futuros proyectos) vamos a necesitar esta conversión en ambos sentidos, vamos a crear una clase auxiliar que contenga los métodos estáticos necesarios para realizar estas conversiones.

Para ellos, vamos a crear una clase llamada UtilidadDeFechas dentro de un nuevo paquete (package) llamado util:

```
package iesthiar.util;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;

public class UtilidadDeFechas {
    // El patrón utilizado para la conversión
    private static final String FECHA_PATTERN = "dd/MM/yyyy";
```

```
10
11      // El formateador de fecha
12      private static final DateTimeFormatter FECHA_FORMATTER =
DateTimeFormatter.ofPattern(FECHA_PATTERN);
13
14      // Devuelve la fecha de entrada como un string formateado
15      public static String formato(LocalDate fecha) {
16          if (fecha == null) {
17              return null;
18          }
19          return FECHA_FORMATTER.format(fecha);
20      }
21
22      // Convierte un string en un objeto de tipo LocalDate
23      // (o null si no puede convertirlo)
24      public static LocalDate convertir(String fecha) {
25          try {
26              return FECHA_FORMATTER.parse(fecha, LocalDate::from);
27          } catch (DateTimeParseException e) {
28              return null;
29          }
30
31      }
32
33      // Compruseba si un string de fecha e una fecha válida y devuelve 1 o 0
34      // Usamos el método anterior para la comprobación
35      public static boolean fechaValida(String fecha) {
36          return UtilidadDeFechas.convertir(fecha) != null;
37      }
38  }
```

Como vemos, hemos utilizado el formato de fecha dd/MM/yyyy, si bien podríamos utilizar cualquier otro consultando als diferentes opciones que nos ofrece DateTimeFormatter.

Y por último, una vez creada la clase anterior, ya podemos sustituir el TODO del método mostrarDetallesPersona(Persona persona) para que quede como sigue:

```
fechaNacimientoEtiqueta.setText(UtilidadDeFechas.formato(persona.getFechaNacimiento()));
```

Detectar cambios en la selección de la tabla

Para saber cuando el usario ha seleccionado a una persona de la tabla y mostrar sus detalles, necesitamos "escuchar" dichos cambios.

Para ello, implementaremos el interface de JavaFX **ChangeListener** con el método **changed(...)**, que solo tiene 3 parámetros: observable, oldValue y newValue.

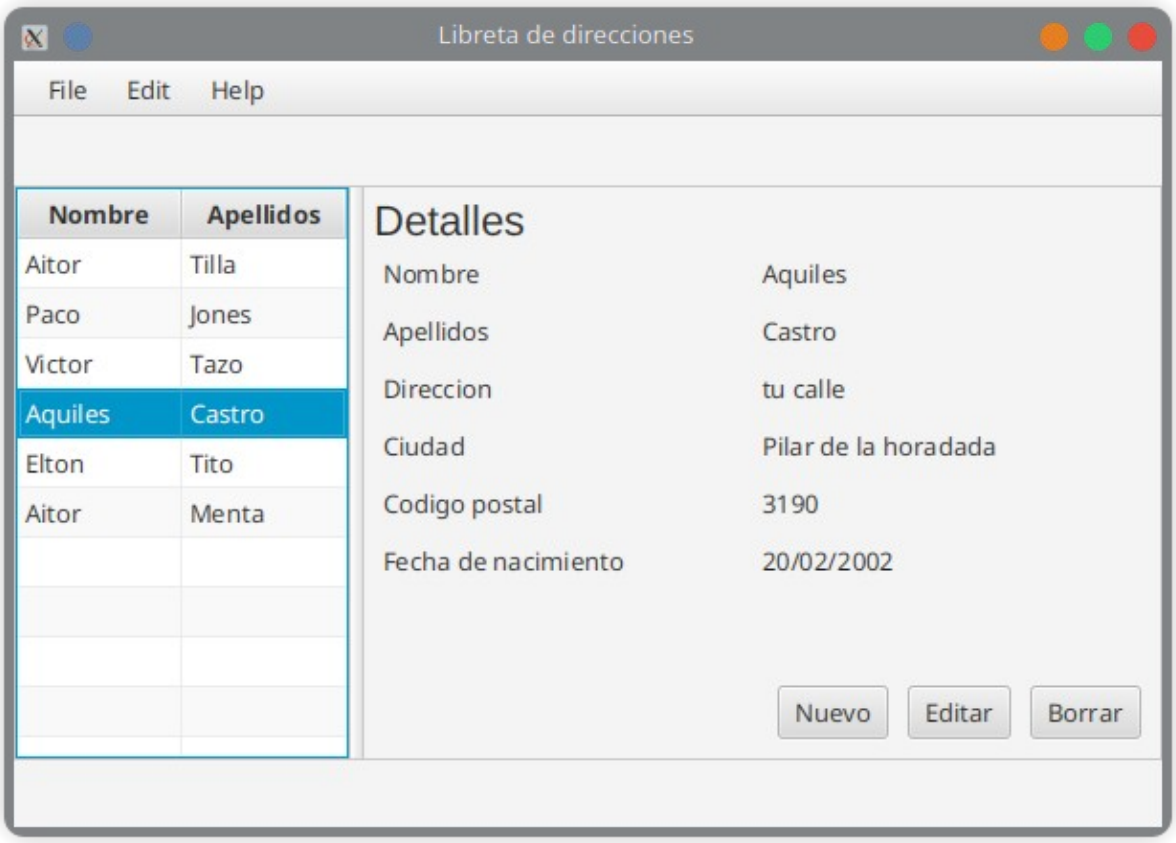
Esto lo vamos a hacer añadiendo al método **initialize()** de **VistaPersonaController** una expresión lambda.

```
1       @Override
2       public void initialize(URL arg0, ResourceBundle arg1) {
3           nombreColumna.setCellValueFactory(cellData -> cellData.getValue().nombreProperty());
4           apellidosColumna.setCellValueFactory(cellData ->
cellData.getValue().apellidosProperty());
5           // Borramos los textos de los datos de una persona
6           mostrarDetallesPersona(null);
7
8           // Escuchamos cambios en la selección de la tabla para mostrar los detalles
9           personaTabla.getSelectionModel().selectedItemProperty().addListener(
10                  (observable, oldValue, newValue) -> mostrarDetallesPersona(newValue));
11      }
```

Con la instrucción mostrarDetallesPersona(null) borramos los detalles de una persona.

Con personaTabla.getSelectionModel()... obtenemos la selectedItemProperty de la tabla de personas, y le añadimos un listener. De este modo, cuando el usuario seleccione a una persona de la tabla, nuestra lambda expression será ejecutada, cogiendo a la persona recién seleccionada y pasándosela al método mostrarDetallesPersona(...).

Si ahora ejecutamos nuestra aplicación (ejecutando Clean and build previamente si es necesario), deberíamos conseguir la funcionalidad implementada y al ir seleccionando diferentes personas en la tabla de la izquierda veremos todos los detalles a la derecha.

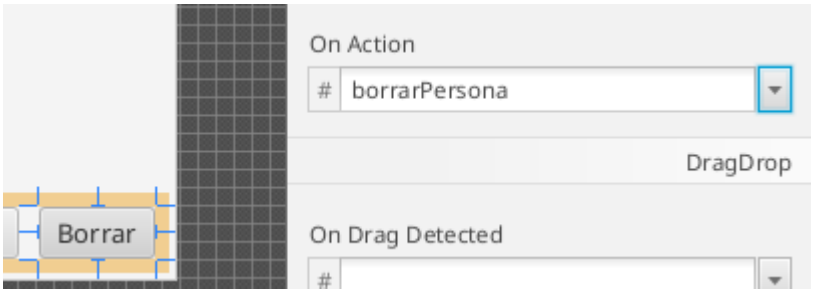


El botón de borrar

Nuestro interfaz de usuario ya contiene un botón de borrar, pero sin funcionalidad. Podemos seleccionar la acción a ejecutar al pulsar un botón desde el *Scene Builder*. Cualquier método de nuestro controlador anotado con `@FXML` (o declarado como *public*) es accesible desde *Scene Builder*. Así pues, empecemos añadiendo el método de borrado al final de nuestra clase `VistaPersonaControlador`:

```
/**
 * Se llama cuando pulsamos el boton Borrar
 */
@FXML
private void borrarPersona() {
    int indiceSeleccionado = personaTabla.getSelectionModel().getSelectedIndex();
    personaTabla.getItems().remove(indiceSeleccionado);
}
```

Ahora, abre el archivo vistaPersona.`fxml` en el *Scene Builder*. Selecciona el botón *Delete*, abre el apartado *Code* y pon `handleDeletePerson` en el menú desplegable denominado **On Action**.



Gestión de errores

Si ejecutas tu aplicación en este punto deberías ser capaz de borrar personas de la tabla. Pero, ¿qué ocurre si pulsas el botón de borrar sin seleccionar a nadie en la tabla.

Se produce un error de tipo **ArrayIndexOutOfBoundsException** porque no puede borrar una persona en el índice -1, que es el valor devuelto por el método **getSelectedIndex()** cuando no hay ningún elemento seleccionado.

Ignorar semejante error no es nada recomendable. Deberíamos hacerle saber al usuario que tiene que seleccionar una persona previamente para poderla borrar (incluso mejor sería deshabilitar el botón para que el usuario ni siquiera tenga la oportunidad de realizar una acción incorrecta).

Con algunos cambios en el método borrarPersona() podemos mostrar una simple ventana de diálogo emergente en el caso de que el usuario pulse el botón Delete sin haber seleccionado a nadie en la tabla de contactos:

```
@FXML
private void borrarPersona() {
  int indiceSeleccionado = personaTabla.getSelectionModel().getSelectedIndex();
  if (indiceSeleccionado >= 0) {
    personaTabla.getItems().remove(indiceSeleccionado);
  } else {
    // Muestro alerta
    Alert alerta = new Alert(AlertType.WARNING);
    alerta.setTitle("Atención");
    alerta.setHeaderText("Persona no seleccionada");
    alerta.setContentText("Por favor, selecciona una persona de la tabla");
    alerta.showAndWait();
  }
}
```

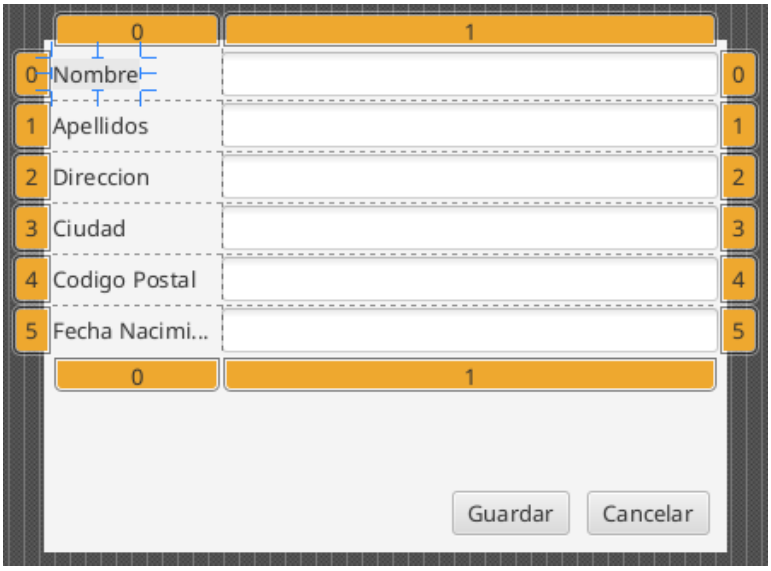
> Toda la información relativa al uso de diálogos en JavaFX la puedes encontrar haciendo clic AQUÍ o en la Documentación oficial de JavaFX.

Diálogos para crear y editar contactos

Las acciones de editar y crear nuevo contacto necesitan algo más de elaboración: vamos a necesitar una ventana de diálogo a medida (es decir, un nuevo stage) con un formulario para preguntar al usuario los detalles sobre la persona.

Vista EditarPersona

Dentro del paquete persona, añadimos un nuevo archivo editarPersona.fxml y, usando un panel de rejilla (GridPane), etiquetas(Label), campos de texto (TextField) y botones (Button) sobre AnchorPane creamos una ventana de diálogo similar a la siguiente:



El controlador

Creamos el controlador para vincularlo a esta ventana, creamos la clase EditorPersonaControlador.java en el paquete persona.

```java
package iesthiar.persona;

import java.net.URL;
import java.util.ResourceBundle;

import iesthiar.util.UtilidadDeFechas;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Alert;
import javafx.scene.control.TextField;
import javafx.stage.Stage;

public class EditorPersonaControlador implements Initializable {
    // TextField para los campos

    @FXML
    private TextField nombreTextField;

    @FXML
    private TextField apellidosTextField;

    @FXML
    private TextField direccionTextField;

    @FXML
    private TextField codigoPostalTextField;

    @FXML
    private TextField ciudadTextField;

    @FXML
    private TextField fechaNacimientoTextField;

    private Stage escenarioEdicion; // Escenario de edición
    private Persona persona; // Referencia a la clase persona
    private boolean guardarClicked = false;

    // Inicializa la clase controller y es llamado justo DESPUÉS de cargar el
    // archivo FXML
    @Override
    public void initialize(URL arg0, ResourceBundle arg1) {
        // TODO Auto-generated method stub

    }

    // Establece el escenario de edición
    public void setEscenarioEdicion(Stage escenarioEdicion) {
        this.escenarioEdicion = escenarioEdicion;
    }

    // Establece la persona a editar
    public void setPersona(Persona persona) {
        this.persona = persona;
        nombreTextField.setText(persona.getNombre());
```

```
 55          apellidosTextField.setText(persona.getApellidos());
 56          direccionTextField.setText(persona.getDireccion());
 57          codigoPostalTextField.setText(Integer.toString(persona.getCodigoPostal()));
 58          ciudadTextField.setText(persona.getCiudad());
 59
fechaNacimientoTextField.setText(UtilidadDeFechas.formato(persona.getFechaNacimiento()));
 60          fechaNacimientoTextField.setPromptText("dd/mm/yyyy");
 61      }
 62
 63      // Devuelve true si se ha pulsado Guardar, si no devuelve false
 64      public boolean isGuardarClicked() {
 65          return guardarClicked;
 66      }
 67
 68      // LLamado cuando se pulsa Guardar
 69      @FXML
 70      private void guardar() {
 71          if (datosValidos()) {
 72              // Asigno datos a propiedades de persona
 73              persona.setNombre(nombreTextField.getText());
 74              persona.setApellidos(apellidosTextField.getText());
 75              persona.setDireccion(direccionTextField.getText());
 76              persona.setCodigoPostal(Integer.parseInt(codigoPostalTextField.getText()));
 77              persona.setCiudad(ciudadTextField.getText());
 78              persona.setFechaNacimiento(null);
 79              guardarClicked = true; // Cambio valor booleano
 80              escenarioEdicion.close(); // Cierro el escenario de edición
 81          }
 82      }
 83
 84      // LLamado cuando se pulsa Cancelar
 85      @FXML
 86      private void cancelar() {
 87          escenarioEdicion.close();
 88      }
 89
 90      // Validación de datos
 91      private boolean datosValidos() {
 92          // Inicializo string para mensajes
 93          String mensajeError = "";
 94          // Compruebo los campos
 95          if (nombreTextField.getText() == null || nombreTextField.getText().length() == 0) {
 96              mensajeError += "Nombre no válido.\n";
 97          }
 98          if (apellidosTextField.getText() == null || apellidosTextField.getText().length()
== 0) {
 99              mensajeError += "Apellidos no válidos.\n";
100          }
101          if (direccionTextField.getText() == null || direccionTextField.getText().length()
== 0) {
102              mensajeError += "Dirección no válida.\n";
103          }
104          if (codigoPostalTextField.getText() == null ||
codigoPostalTextField.getText().length() == 0) {
105              mensajeError += "Código postal no válido.\n";
106          } else {
107              // Convierto el código postal a entero
108              try {
109                  Integer.parseInt(codigoPostalTextField.getText());
110              } catch (NumberFormatException e) {
```

```
                    mensajeError += "Código postal no válido (debe ser un entero).\n";
                }
            }
            if (ciudadTextField.getText() == null || ciudadTextField.getText().length() == 0) {
                mensajeError += "Ciudad no válida.\n";
            }
            if (fechaNacimientoTextField.getText() == null ||
fechaNacimientoTextField.getText().length() == 0) {
                mensajeError += "Fecha de nacimiento no válida.\n";
            } else {
                if (!UtilidadDeFechas.fechaValida(fechaNacimientoTextField.getText())) {
                    mensajeError += "Fecha de nacimiento no válida (debe tener formato
dd/mm/yyyy).\n";
                }
            }
            // Si no hay errores devuelvo true, si no, una alerta con los errores y false
            if (mensajeError.length() == 0) {
                return true;
            } else {
                // Muestro alerta y devuelvo false
                Alert alerta = new Alert(Alert.AlertType.ERROR);
                alerta.setTitle("Error");
                alerta.setHeaderText("Datos no válidos");
                alerta.setContentText("Por favor, corrige los errores");
                alerta.showAndWait();
                return false;
            }
        }
    }
```

Algunas cuestiones relativas a este controlador:

- El método setPersona(...) lo invocaremos desde otra clase para establecer la persona que será editada.
- Cuando el usuario pulse Guardar, el método guardar() es invocado. Primero se valida la entrada del usuario mediante la ejecución del método datosValidos().
- Sólo si la validación tiene éxito el objeto Persona es modificado con los datos introducidos por el usuario. Esos cambios son aplicados directamente sobre el objeto pasado como argumento del método setPersona(...)
- El método boolean isGuardarClicked() se utiliza para determinar si el usuario ha pulsado el botón Guardar o el botón Cancelar.

Enlaza la vista y el controlador

Una vez creadas la vista (FXML) y el controlador, necesitamos vincular el uno con el otro:

- Abre el archivo PersonEditDialog.fxml.
- En la sección Controller a la izquierda selecciona EditorPersonaControlador como clase de control.

- Establece el campo fx:id de todas los TextField con los identificadores de los atributos del controlador correspondientes.
- Especifica el campo onAction de los dos botones con los métodos del controlador correspondientes a cada acción.

Abrir la vista EditarPersona

La vista EditarPersona la abriremos desde un método nuevo llamado mostrarEditarPersona dentro de LibretaDirecciones.java:

```
// Vista editarPersona
public boolean muestraEditarPersona(Persona persona) {
  // Cargo la vista persona a partir de VistaPersona.fxml
  AnchorPane editarPersona = null;
  FXMLLoader loader = new FXMLLoader();

  try {
    URL location =
        LibretaDirecciones.class.getResource("persona/editarPersona.fxml");
    loader.setLocation(location);
    editarPersona = (AnchorPane) loader.load();
  } catch (IOException ex) {
    // ex.printStackTrace();
    System.err.println("--------------------------------------");
    return false;
  }

  // Creo el escenario de edición (con modal) y establezco la escena
  Stage escenarioEdicion = new Stage();
  escenarioEdicion.setTitle("Editar Persona");
  escenarioEdicion.initModality(Modality.WINDOW_MODAL);
  escenarioEdicion.initOwner(escenarioPrincipal);
  Scene escena = new Scene(editarPersona);
  escenarioEdicion.setScene(escena);

  // Asigno el escenario de edición y la persona seleccionada al controlador
  EditorPersonaControlador controlador = loader.getController();
  controlador.setEscenarioEdicion(escenarioEdicion);
  controlador.setPersona(persona);

  // Muestro el diálogo hasta que el usuario lo cierre
  escenarioEdicion.showAndWait();

  // devuelvo el botón pulsado
  return controlador.isGuardarClicked();
}
```

Añade los siguientes métodos a la clase VistaPersonaControlador. Esos métodos llamarán al método anterior muestraEditarpersona(...) de LibretaDirecciones.java cuando el usuario pulse en los botones Crear o Editar

```
// Muestro el diálogo editar persona cuando el usuario hace clic en el botón de Crear
@FXML
```

```
3  private void crearPersona() {
4    Persona temporal = new Persona();
5    boolean guardarClicked = libretaDirecciones.muestraEditarPersona(temporal);
6    if (guardarClicked) {
7      libretaDirecciones.getDatosPersona().add(temporal);
8    }
9  }
10
11 // Muestro el diálogo editar persona cuando el usuario hace clic en el botón de Editar
12 @FXML
13 private void editarPersona() {
14   Persona seleccionada = personaTabla.getSelectionModel().getSelectedItem();
15   if (seleccionada != null) {
16     boolean guardarClicked =
17         libretaDirecciones.muestraEditarPersona(seleccionada);
18     if (guardarClicked) {
19       mostrarDetallesPersona(seleccionada);
20     }
21   } else {
22     // Muestro alerta
23     Alert alerta = new Alert(Alert.AlertType.WARNING);
24     alerta.setTitle("Alerta");
25     alerta.setHeaderText("Persona no seleccionada");
26     alerta.setContentText("Por favor, selecciona una persona");
27     alerta.showAndWait();
28   }
29 }
```

Para finalizar, abre el archivo vistaPersona.fxml mediante Scene Builder y elige los métodos correspondientes (crearPersona() y editarPersona() para el campo On Action de la sección Code (derecha) de los botones Crear y Editar.

Llegados a este punto deberíamos tener nuestra aplicación LibretaDirecciones en funcionamiento, con un aspecto similar al mostrado a continuación. Esta aplicación es capaz de añadir, editar y borrar personas. Tiene incluso algunas capacidades de validación para evitar que el usuario introduzca información incorrecta.

Otras funcionalidades para agregar a tu proyecto

Funcionalidad que nos permita añadir la fecha de nacimiento mediante un desplegable de selección de fechas, usando para ello un JavaFX DatePicker. Puedes seguir ESTE TUTORIAL.

Funcionalidad que nos permita ordenar y filtrar los datos de la tabla, usando para ello las clases SortedList y FilteredList. Puedes seguir ESTE TUTORIAL.

Funcionalidad que nos permita renderizar las celdas de la tabla en función de su contenido, usando para ello Cell Factory y Cell Value Factory. Puedes seguir ESTE TUTORIAL

3.5 Hojas de estilo CSS

- Estilos mediante CSS
- Añadiendo un Icono de Aplicación

En JavaFX puedes dar estilo al interfaz de usuario utilizando hojas de estilo en cascada (CSS).

En este tutorial vamos a crear un tema oscuro (DarkTheme) inspirado en el diseño de Windows 8 Metro. El código CSS de los botones está basado en el artículo de blog JMetro - Windows 8 Metro controls on Java de Pedro Duque Vieira.

Para información más específica de CSS y JavaFX puedes consultar:

- Skinning JavaFX Applications with CSS - Tutorial by Oracle
- JavaFX CSS Reference - Official Reference

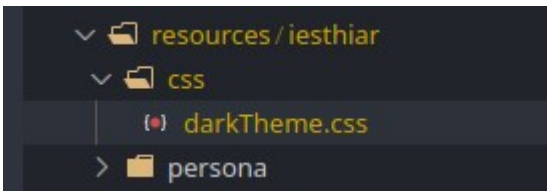
Los estilos por defecto en JavaFX

Los estilos por defecto de JavaFX 8 se encuentran en un archivo denominado modena.css. Este archivo CSS se encuentra dentro del archivo jfxrt.jar que se encuentra en tu directorio de instalación de JavaFX.

Dentro de ese archivo, el fichero modena.css se encuentra en el paquete com.sun.javafx.scene.control.skin.modena

Este estilo se aplica siempre a una aplicación JavaFX. Añadiendo un estilo personal podremos reescribir los estilos por defecto.

Crear el archivo CSS



Crea un nuevo directorio llamado css dentro del paquete principal de resources y dentro de él un fichero CSS denominado darkTheme.css, con el siguiente contenido:

```
.background {
    -fx-background-color: #1d1d1d;
}

.label {
    -fx-font-size: 11pt;
    -fx-font-family: "Segoe UI Semibold";
    -fx-text-fill: white;
    -fx-opacity: 0.6;
```

```
}

.label-bright {
    -fx-font-size: 11pt;
    -fx-font-family: "Segoe UI Semibold";
    -fx-text-fill: white;
    -fx-opacity: 1;
}

.label-header {
    -fx-font-size: 32pt;
    -fx-font-family: "Segoe UI Light";
    -fx-text-fill: white;
    -fx-opacity: 1;
}

.table-view {
    -fx-base: #1d1d1d;
    -fx-control-inner-background: #1d1d1d;
    -fx-background-color: #1d1d1d;
    -fx-table-cell-border-color: transparent;
    -fx-table-header-border-color: transparent;
    -fx-padding: 5;
}

.table-view .column-header-background {
    -fx-background-color: transparent;
}

.table-view .column-header, .table-view .filler {
    -fx-size: 35;
    -fx-border-width: 0 0 1 0;
    -fx-background-color: transparent;
    -fx-border-color:
        transparent
        transparent
        derive(-fx-base, 80%)
        transparent;
    -fx-border-insets: 0 10 1 0;
}

.table-view .column-header .label {
    -fx-font-size: 20pt;
    -fx-font-family: "Segoe UI Light";
    -fx-text-fill: white;
    -fx-alignment: center-left;
    -fx-opacity: 1;
}

.table-view:focused .table-row-cell:filled:focused:selected {
    -fx-background-color: -fx-focus-color;
}

.split-pane:horizontal > .split-pane-divider {
    -fx-border-color: transparent #1d1d1d transparent #1d1d1d;
    -fx-background-color: transparent, derive(#1d1d1d,20%);
}

.split-pane {
    -fx-padding: 1 0 0 0;
```

```
}

.menu-bar {
    -fx-background-color: derive(#1d1d1d,20%);
}

.context-menu {
    -fx-background-color: derive(#1d1d1d,50%);
}

.menu-bar .label {
    -fx-font-size: 14pt;
    -fx-font-family: "Segoe UI Light";
    -fx-text-fill: white;
    -fx-opacity: 0.9;
}

.menu .left-container {
    -fx-background-color: black;
}

.text-field {
    -fx-font-size: 12pt;
    -fx-font-family: "Segoe UI Semibold";
}

/*
 * Metro style Push Button
 * Author: Pedro Duque Vieira
 * http://pixelduke.wordpress.com/2012/10/23/jmetro-windows-8-controls-on-java/
 */
.button {
    -fx-padding: 5 22 5 22;
    -fx-border-color: #e2e2e2;
    -fx-border-width: 2;
    -fx-background-radius: 0;
    -fx-background-color: #1d1d1d;
    -fx-font-family: "Segoe UI", Helvetica, Arial, sans-serif;
    -fx-font-size: 11pt;
    -fx-text-fill: #d8d8d8;
    -fx-background-insets: 0 0 0 0, 0, 1, 2;
}

.button:hover {
    -fx-background-color: #3a3a3a;
}

.button:pressed, .button:default:hover:pressed {
  -fx-background-color: white;
  -fx-text-fill: #1d1d1d;
}

.button:focused {
    -fx-border-color: white, white;
    -fx-border-width: 1, 1;
    -fx-border-style: solid, segments(1, 1);
    -fx-border-radius: 0, 0;
    -fx-border-insets: 1 1 1 1, 0;
}
```

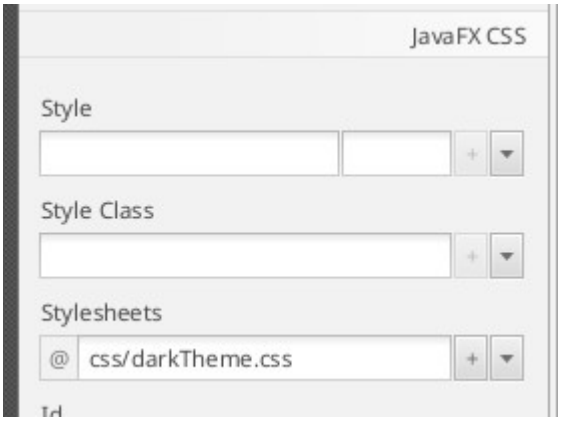
```
.button:disabled, .button:default:disabled {
    -fx-opacity: 0.4;
    -fx-background-color: #1d1d1d;
    -fx-text-fill: white;
}

.button:default {
    -fx-background-color: -fx-focus-color;
    -fx-text-fill: #ffffff;
}

.button:default:hover {
    -fx-background-color: derive(-fx-focus-color,30%);
}
```

Vincular vistas y estilos

Para vincular y el archivo CSS y asociar la clases correspondientes podríamos utilizar Java, si bien en este primer tutorial vamos a hacerlo mediante Scene Builder para que sea más visual:

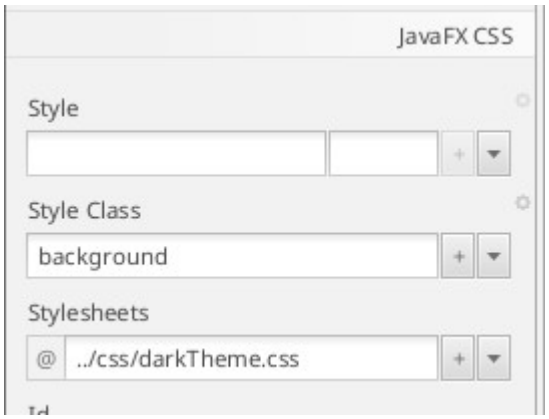
VistaPrincipal.fxml

- Abrimos el archivo VistaPrincipal.fxml y seleccionamos el BorderPane raíz.
- En la sección Properties (derecha) añadimos la ruta de nuestro Stylesheet.



EditarPersona.fxml

- Abrimos el archivo EditarPersona.fxml y seleccionamos el AnchorPane raíz.
- En la sección Properties (derecha) añadimos la ruta de nuestro Stylesheet.
- Como el fondo aún es blanco, añadimos la clase background al AnchorPane raíz mediante la propiedad Style Class.



- Selecciona el botón Guardar y elige Default Button en la vista Properties. Eso cambiará su color y lo convertirá en el botón "por defecto", el que se ejecutará si el usuario aprieta la tecla Enter.
- Selecciona el botón Cancelar y elige Cancel Button en la vista Properties.
- Posiblemente tengas que ajustar el tamaño de algunos botones y paneles para que se muestre todo el texto.

VistaPersona.fxml

- Abrimos el archivo VistaPersona.fxml y seleccionamos el AnchorPane raíz.
- En la sección Properties (derecha) añadimos la ruta de nuestro Stylesheet.
- Selecciona el panel AnchorPane de la derecha, dentro del SplitPane.
- En Properties, slecciona background como clase de estilo. El fondo debería volverse negro.
- Selecciona la etiqueta (Label) Detalles y añade label-header como clase de estilo.
- Para cada etiqueta en la columna de la derecha (donde se muestran los detalles de una persona), añade la clase de estilo label-bright.

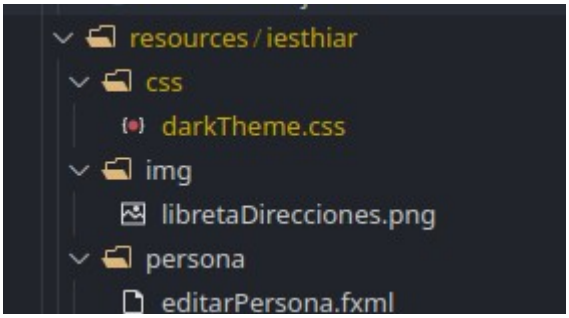
Posiblemente tengas que ajustar el tamaño de algunos botones y paneles para que se muestre todo el texto.

Icono de aplicación

Ahora mismo nuestra aplicación utiliza el icono por defecto para la barra de título y la barra de tareas, pero quedaría mucho mejor con un icono personalizado.

Un posible sitio para obtener iconos gratuitos es Icon Finder. Yo por ejemplo descargué este icono de libreta de direcciones.

Una vez descargado el icono, crea un nuevo directorio img dentro del directorio principal de resources y añade el archivo descargado, en mi caso libretaDirecciones.png , redimensionado a 32px de altura.

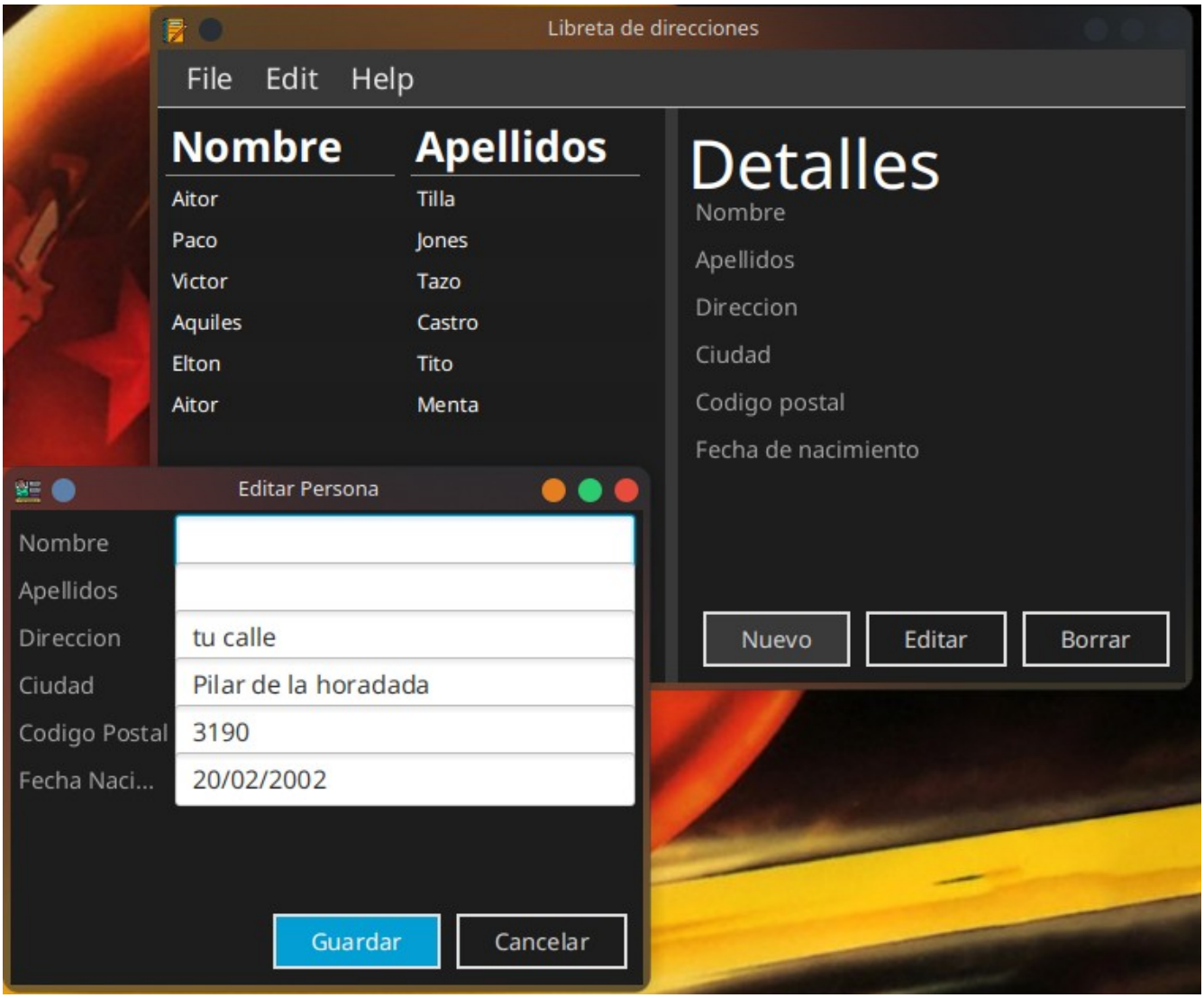


Ahora modifica el método start() de LibretaDirecciones.java para que quede como sigue:

```
@Override
public void start(Stage escenarioPrincipal) {
  // Necesario para cambiar las escenas
  this.escenarioPrincipal = escenarioPrincipal;
  // Establezco el título
  this.escenarioPrincipal.setTitle("Libreta de direcciones");
  // Establezco el icono de aplicación
  this.escenarioPrincipal.getIcons().add(
      new Image(LibretaDirecciones.class.getResourceAsStream(
          "img/libretaDirecciones.png")));
  // inicialización del contenedor principal
  initContenedorPrincipal();
  // muestro la vista persona
  mostrarVistaPersona();
}
```

Por supuesto, podrías hacer lo mismo para el método muestraEditarPersona y asignarle su propio icono de edición.

El resultado final de este capítulo debería ser similar a esto:


Libreta de direcciones
File
Edit
Help
Nombre
Apellidos
Aitor
Tilla
Paco
Jones
Victor
Tazo
Aquiles
Castro
Elton
Tito
Aitor
Menta
Detalles
Nombre
Apellidos
Direccion
Ciudad
Codigo postal
Fecha de nacimiento
Nuevo
Editar
Borrar
Editar Persona
Nombre
Apellidos
Direccion
tu calle
Ciudad
Pilar de la horadada
Codigo Postal
3190
Fecha Naci...
20/02/2002
Guardar
Cancelar