Controles en JavaFX

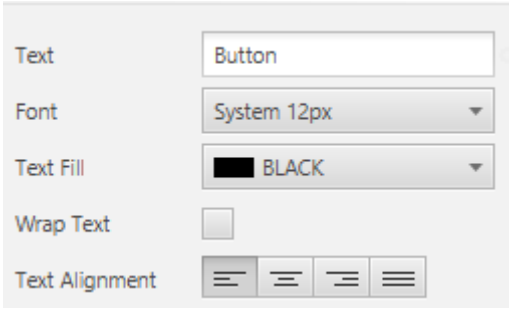
1. Propiedades comunes 2
1.1. Pestaña Properties. Propiedades relacionadas con texto 2
1.2. Pestaña Properties. Propiedades relacionadas con alineación y otros estilos 2
1.3. Pestaña Layout. Propiedades relacionadas con espaciados y posicionamiento 3
1.4. IDs y controladores 4
1.5. Acceso a propiedades de controles desde el código Java 6
2. Controles simples estáticos 7
2.1. Botón 7
2.2. Label 8
2.3. CheckBox 8
2.4. RadioButton 9
2.5. TextField 9
2.6. PasswordField 10
2.7. TextArea 10
2.8. ColorPicker 10
2.9. ImageView 10
2.10. Slider 11
2.11. Hyperlink 12
3. Controles simples dinámicos 12
3.1. ChoiceBox 13
3.2. ComboBox 14
3.3. ListView 15
3.4. TableView 16
3.5. TreeView 20
4. Controles gráficos avanzados 23
4.1. MenuBar 23
4.2. ContextMenu 26
4.3. TabPane 28
4.4. TitledPane 29
4.5. Accordion 30
4.6. ScrollPane 31
4.7. ToolBar 33
4.8. ToolTip 33

El objetivo de este documento es explicar el funcionamiento los controles más importantes en JavaFX

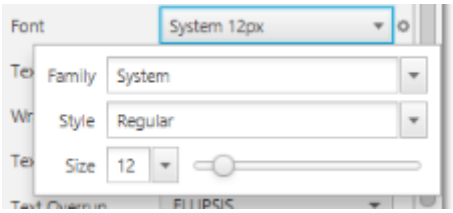
1. Propiedades comunes

Antes de describir cada uno de los componentes en profundidad, se indicarán una serie de propiedades comunes a todos ellos que pueden resultar de ayuda para abordar cualquier diseño.

1.1. Pestaña Properties. Propiedades relacionadas con texto



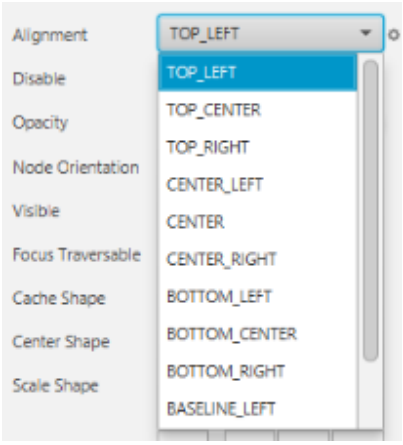
- Text: Es el contenido de aquellas etiquetas que requieren texto.
- Font: Permite cambiar la fuente y el estilo a negrita, cursiva u otros.

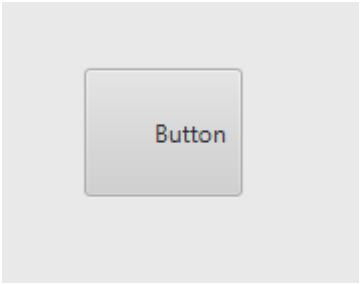


- Text Fill: Es el color del texto y permite añadir efectos como degradados.
- Text Alignment: Sirve para alinear texto en los controles que así lo permiten.

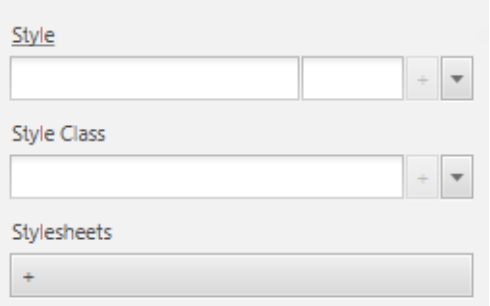
1.2. Pestaña Properties. Propiedades relacionadas con alineación y otros estilos

- Alignment: Permite alinear el contenido de manera horizontal y vertical. Por ejemplo, el siguiente botón tiene los valores CENTER_RIGHT



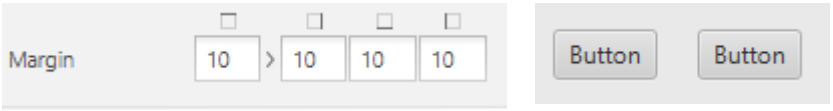


- Style: Permite añadir estilos con la misma nomenclatura que en CSS. Se pueden asignar directamente, con clases o añadiendo un CSS externo.

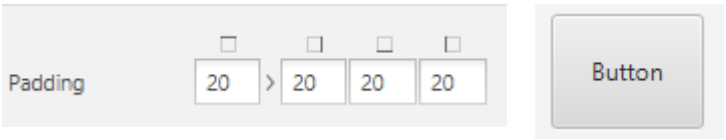


1.3. Pestaña Layout. Propiedades relacionadas con espaciados y posicionamiento

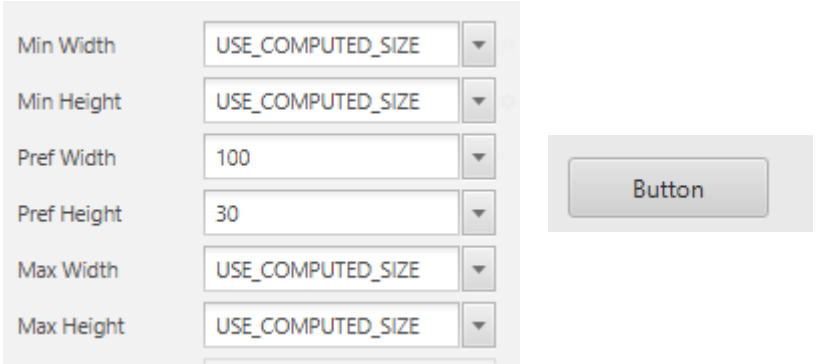
- Margin: Permite desplazar un control respecto a los elementos contiguos en cuatro posiciones: arriba, derecha, abajo e izquierda. Por ejemplo, los dos botones siguientes tienen un margen de 10px entre sí que hace que haya un espacio entre ellos.



- Padding: Añade un relleno en blanco entre un control y su contenido. Permite aplicar las mismas coordenadas que margin. Por ejemplo, el siguiente botón tiene un padding 20px en todos los lados.

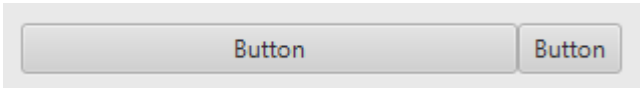


- Altos y anchos: Por defecto, el alto y ancho de un control se adapta a su contenido. No obstante, podemos indicar el ancho y alto manualmente con **Pref Width** y **Pref Height**. Estas coordenadas son orientativas si se redimensiona la pantalla y nunca serán inferiores o superiores a los valores **Min** y **Max** del ancho y alto respectivamente. Por ejemplo el botón de la imagen tiene unas dimensiones de 100 x 30.



- Hgrow: Si tenemos un contenedor con un ancho predefinido y queremos que todos los controles ocupen el total de este tamaño, podemos diseñarlo fácilmente con Hgrow. Para ello, necesitamos indicar en los controles que su ancho o alto será el máximo posible. Esto se puede hacer con **Max Width** o **Max Height** y el valor **MAX_VALUE.** Con Hgrow es posible indicar el crecimiento de un control respecto a los demás con los valores NEVER, SOMETIMES y ALWAYS para establecer la prioridad. Por ejemplo, si dos elementos con ancho **MAX_VALUE** tienen el valor de Hgrow **ALWAYS** crecerán hasta ocupar el ancho del contenedor.

Si el valor de alguno de los botones fuese NEVER, entonces ocuparía el mínimo espacio posible a pesar de haber asignado MAX_VALUE al ancho máximo.

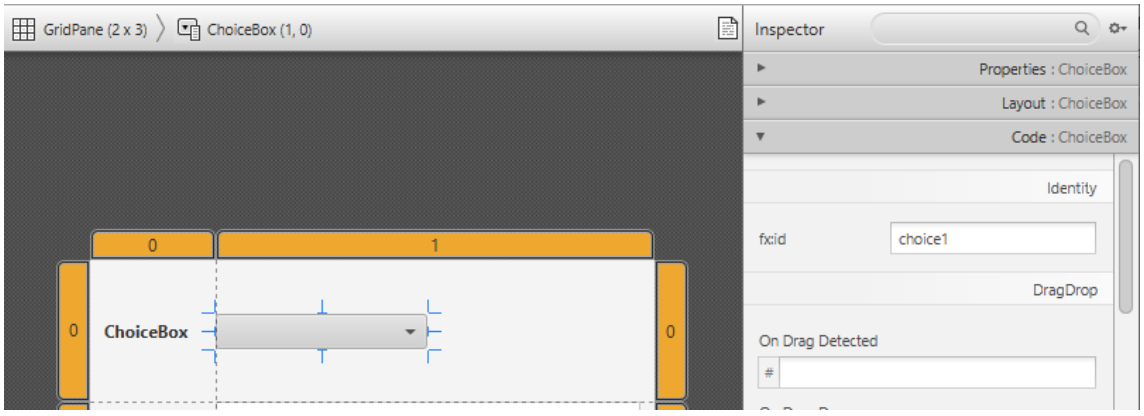


1.4. IDs y controladores

De cara a emplear en Java los controles creados en el archivo FXML, necesitamos poder acceder desde el código de alguna manera.

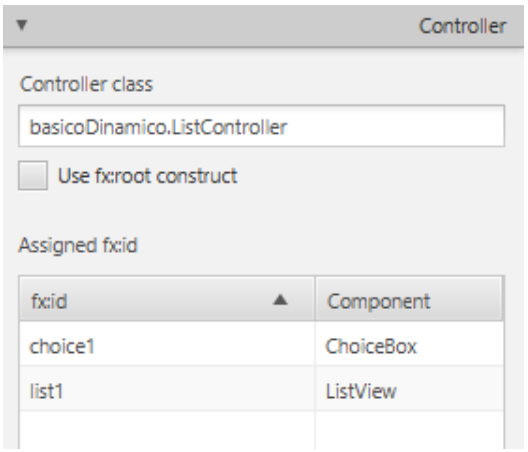
Para ello, en primer lugar necesitamos asignar un **identificador** a todos los controles cuya información queramos manipular posteriormente.

El identificador se crea en la pestaña **code** a la derecha mediante el atributo fx:id. Por ejemplo, en la imagen se asigna a un control **ChoiceBox**.

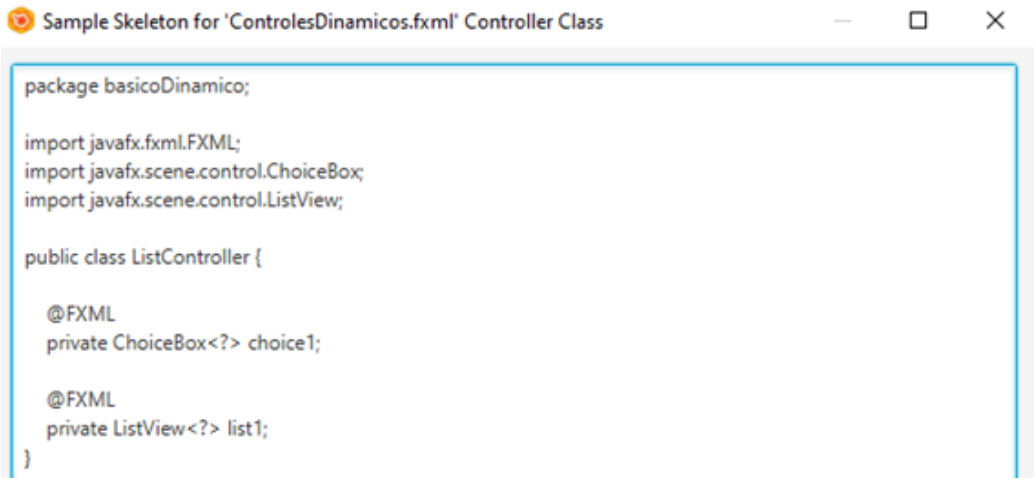


Una vez asignados todos los IDs necesarios, la clase de Java que permite acceder a los controles creados en FXML se denomina **controlador**

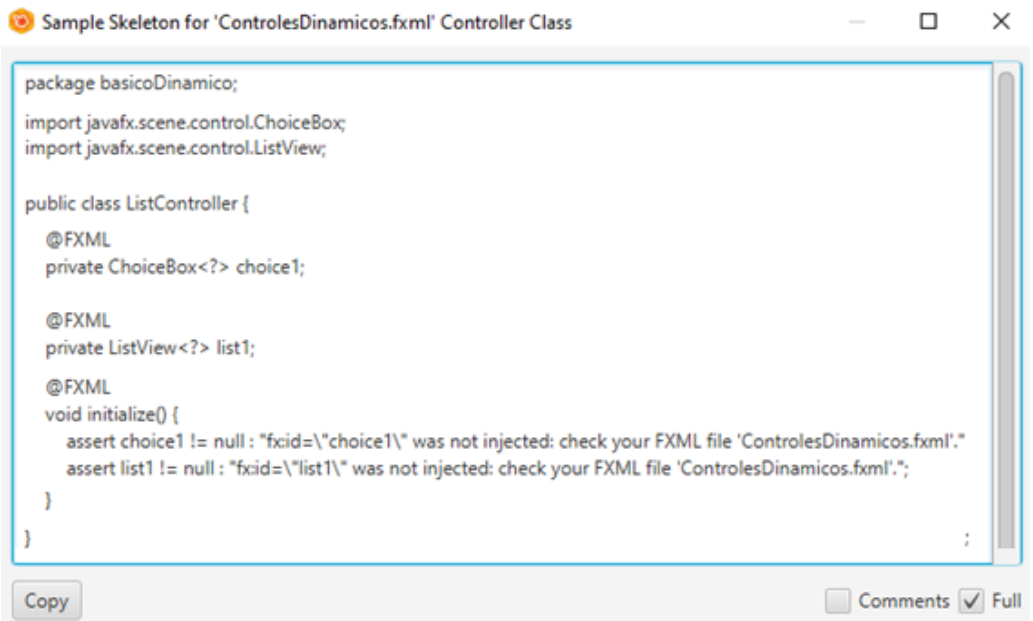
Primero asignamos el nombre del controlador (se puede incluir el paquete a la izquierda) en la pestaña de la izquierda **Controller**. Aquí también nos aparecen los diferentes ID creados.



Por último podemos generar el código de un controlador desde SceneBuilder en el menú **View -> Show Sample Controller Skeleton**.

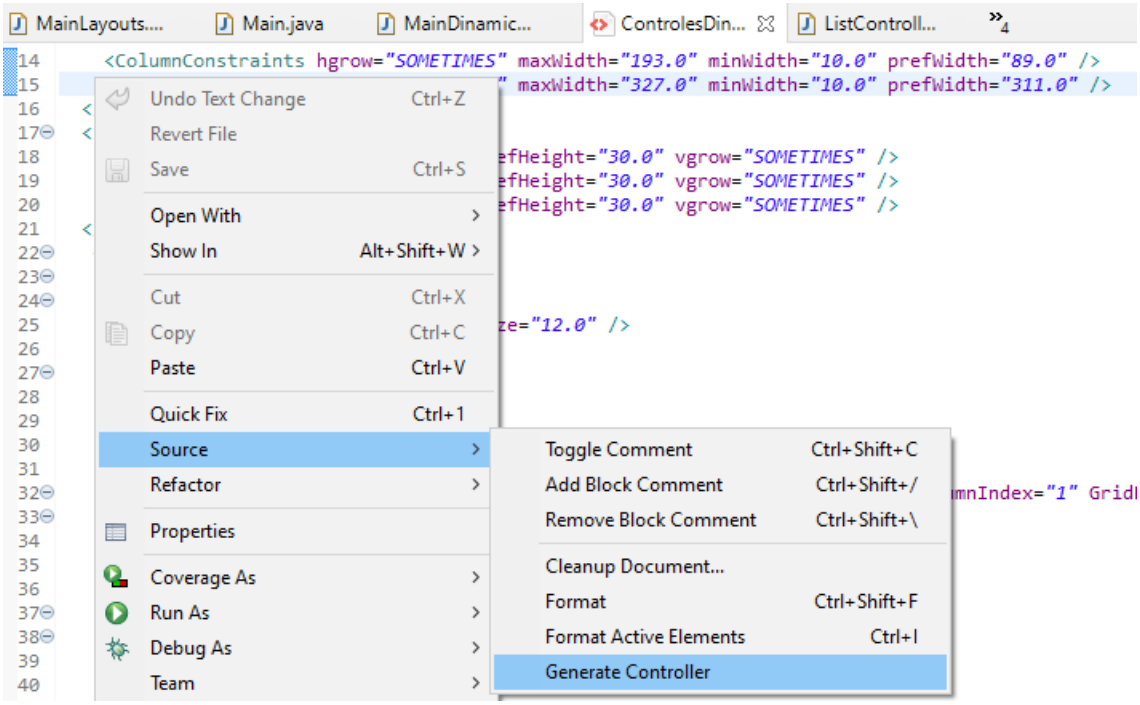


Los controladores requieren un método **initialize** que se ejecuta cada vez que el método Main carga el controlador. Lo podemos generar también marcando la casilla **Full**.



Una vez realizado este paso, desde Java podemos crear una **clase con este código y siempre debe estar localizada en el mismo paquete que el archivo FXML**.

Otra alternativa para generar el controlador sería desde el propio código del archivo FXML. Botón derecho y desde la opción **Source -> Generate Controller**

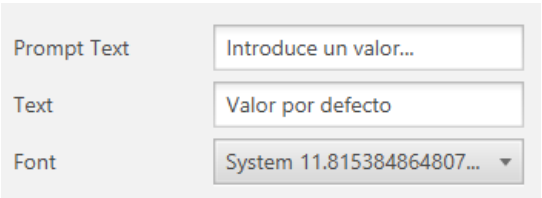


1.5. Acceso a propiedades de controles desde el código Java

En principio vamos a trabajar con **SceneBuilder** y el formato **FXML.** No obstante, en el apartado anterior hemos visto que en algunos momentos es necesario acceder a las propiedades de un control desde el código Java, por ejemplo, a través del controlador.

Una vez que hemos asignado un ID a un control y tenemos acceso desde Java, es bastante común acceder a las propiedades que se indican a continuación:

- **getText** y **setText:** Lo utilizaremos en unidades posteriores y sirve para acceder y modificar dinámicamente el texto de ítems como un **TextField** o **TextArea**.



- **getValue** y **setValue**: Permiten acceder o modificar dinámicamente el contenido de controles en los que se puede seleccionar un único valor, como un **ComboBox** o **ChoiceBox**. Por ejemplo, en el **ComboBox** de debajo asignamos por defecto el valor "Combo2".

```
combo1.getItems().addAll("Combo1", "Combo2", "Combo3");
combo1.setValue("Combo2");
```

- **getItems** o **setItems:** Para los controles dinámicos que veremos más adelante, nos permiten añadir directamente a través del código Java los ítems que van a contener, ya sea en listados, tablas o cuadros desplegables. En la imagen anterior se puede observar cómo inicializar los elementos de un **ComboBox**.
- **getSelectionModel:** Es una propiedad muy importante que permite verificar qué ítems se han seleccionado en aquellos controles que lo permiten, por ejemplo **ComboBox**, **ListView, TableView**, etc. En la unidad 10 lo veremos con más detalle, aunque lo utilizaremos antes para configurar que en un control se puedan seleccionar varios elementos a la vez, entre otras opciones.
- En relación a lo anterior, hay ítems más sencillos como un **CheckBox** o **RadioButton** que directamente tienen una propiedad booleana "selected" que permite verificar con los getter y setter correspondientes si se han seleccionado o no.

En definitiva, por cada control podemos acceder a la especificación y modificar desde el código Java cualquier propiedad. Hay algunas propiedades que no es posible cambiar desde **SceneBuilder**. Veamos algunos ejemplos:

- Si queremos seleccionar más de un ítem a la vez en un listado, solo se puede definir desde Java.

```
// Para seleccionar más de un ítem a la vez
list1.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

- En controles como **TreeView**, por defecto los ítems no se expanden y se maneja desde el código Java.

```
// Expadimos por defecto el ítem raíz
rootItem.setExpanded(true);
```

Por tanto, como conclusión, no hay mayor secreto que tener presente la especificación y por cada control conocer qué propiedades se pueden modificar desde **SceneBuilder** y cuándo es necesario acceder al código Java.

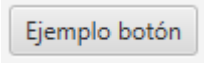
2. Controles simples estáticos

En esta sección se resumen los controles estáticos más comunes a la hora de diseñar una interfaz gráfica con JavaFX. Con estáticos se hace referencia a que se pueden incluir en el diseño desde el SceneBuilder sin apenas modificar código en Java, salvo propiedades muy específicas que no se incluyen en esta herramienta.

Solamente se explicarán las propiedades específicas de cada uno, ya que los estilos comunes se han descrito en la sección anterior.

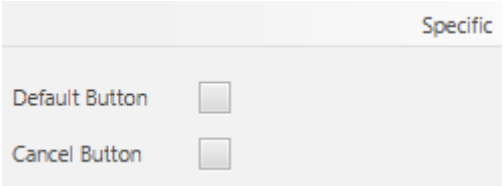
2.1. Botón

Su funcionamiento es simple y ejecuta una acción cuando el usuario hace clic sobre él.



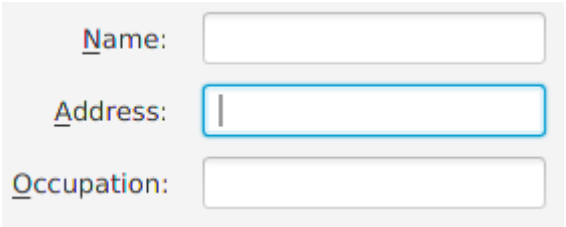
Algunas propiedades específicas de un botón en la pestaña Properties son los modos que se indican a continuación:

- **Default**, un botón que se puede accionar directamente con la tecla ENTER.
- **Cancel**, un botón que se puede accionar directamente con la tecla ESC.



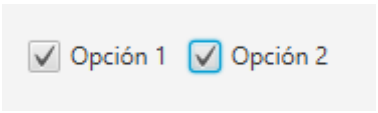
2.2. Label

Se emplea para incluir texto no editable. Su uso principal es para añadir el título de cualquier sección como los Name, Address u Occupation de la imagen de debajo.

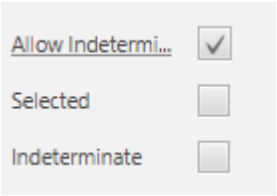


2.3. CheckBox

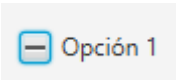
Es un control que admite dos estados: **true** o **false**. Si hay varios CheckBox, no se pueden agrupar y además se pueden activar o desactivar todos al mismo tiempo.



JavaFX permite un tercer estado en un CheckBox denominado “Indeterminate” que se puede activar con la propiedad **Allow Indeterminate**



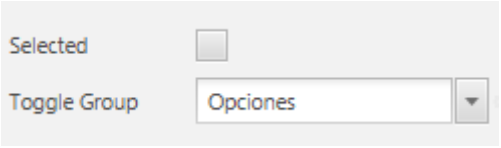
Visualmente, esta propiedad sería como se indica a continuación y a través del código Java se puede comprobar si un checkbox presenta actualmente este estado:



2.4. RadioButton

Al igual que los controles CheckBox, permiten dos estados, en este caso **ON** y **OFF**.

A diferencia de un CheckBox, se pueden agrupar diferentes **RadioButton** para que solamente se pueda seleccionar uno de ellos al mismo tiempo. Esto se logra con la propiedad **ToggleGroup** dentro de la pestaña Properties.

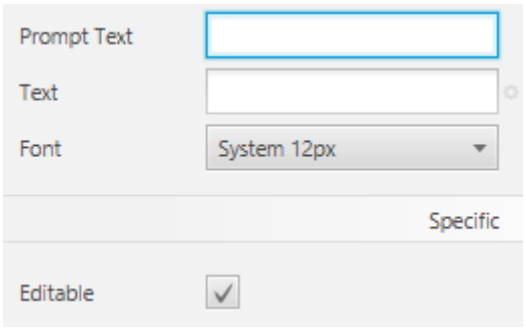


De esta manera, de todos los botones que tienen el mismo valor de **ToggleGroup**, solamente se podrá seleccionar uno a la vez.

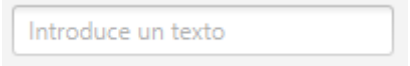
2.5. TextField

Es un control que permite editar una línea de texto con la aplicación ya iniciada y recoger así valores proporcionados por el usuario.

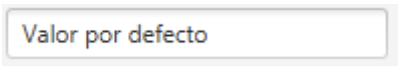
Algunas propiedades específicas de **TextField** son las que se especifican en la siguiente imagen y se explican justo a continuación:



- **Prompt Text:** Texto por defecto que aparece difuminado y que se reemplaza en cuanto el usuario introduce el valor deseado.



- **Text:** Texto por defecto que presenta el control si no introducimos ningún otro valor. Si se asigna Prompt Text y Text a la vez, tiene prioridad Prompt Text.



- **Font:** Permite cambiar el estilo del texto introducido en este control.
- **Editable:** Un TextField puede no ser editable.

2.6. PasswordField

Exactamente igual que **TextField** y con las mismas propiedades, aunque con la diferencia de que está pensado para introducir contraseñas y el texto aparece encriptado.

2.7. TextArea

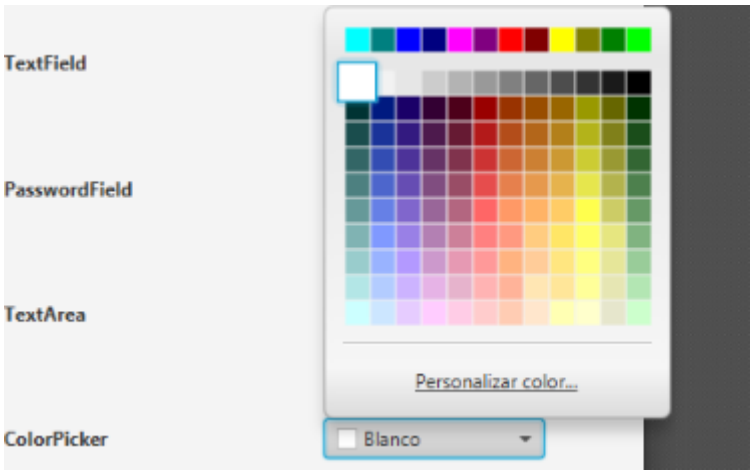
Exactamente igual que **TextField**, aunque está diseñado para añadir texto en varias líneas.

La propiedad más importante es **setWrapText** que permite indicar si se añade un salto de línea cuando se supera el ancho del control. Si no se activa, solamente será posible cuando el usuario presione ENTER.

El resto de propiedades son las mismas que se explicaban anteriormente en TextField

2.8. ColorPicker

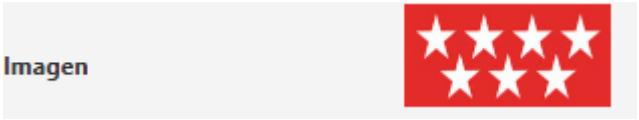
Muestra un panel que permite seleccionar entre varios colores. El valor recogido mediante el código de Java con **getValue()** puede servir para asignar colores a otros controles.



2.9. ImageView

Permite añadir una imagen externa en una aplicación de JavaFX

Para ello, podemos asignar la ruta en la propiedad específica de la pestaña **Properties**.



Otras propiedades a tener en cuenta son las que se indican en la imagen de debajo y se explican a continuación:

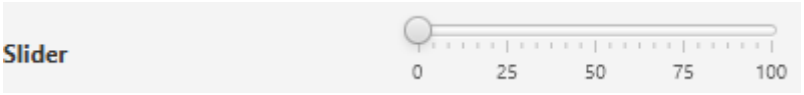
- **Preserve Ratio:** En caso de redimensionar la imagen, el alto y el ancho son proporcionales al tamaño original:
- **Smooth:** El valor true permite aplicar un algoritmo de JavaFX para mejorar la calidad de la imagen.

En ciertas ocasiones necesitaremos crear un **ImageView** en el código Java para emplearlo con otro componente. La manera más sencilla sería la siguiente:

```
new ImageView(new Image(getClass().getResourceAsStream("root.png")));
```

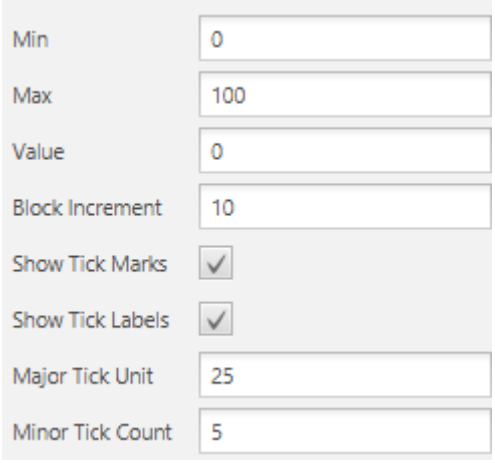
2.10. Slider

Permite seleccionar entre un rango de valores numéricos como se muestra a continuación:



También existe el **slider** vertical, cuyo funcionamiento es igual simplemente cambiando la disposición.

Las propiedades más comunes son las siguientes y se añade una explicación debajo:

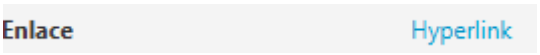


Si echamos un vistazo a la imagen anterior con el **slider**, podemos visualizar de una forma gráfica a qué se corresponde cada valor:

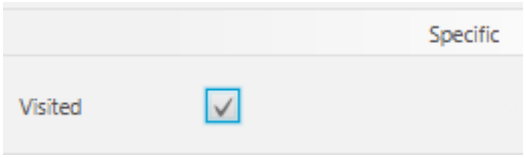
- **Min:** Valor mínimo.
- **Max:** Valor máximo.
- **Value:** Valor actual (si se configura desde SceneBuilder sería el valor por defecto).
- **Block Increment:** Cuando el foco está sobre el **slider**, podemos cambiar sus valores con las flechas del teclado. Con Block Increment se especifica cuántos valores corresponden a cada desplazamiento del teclado (por ejemplo, si es de 10, con la flecha derecha pasamos de 10 a 20, de 20 a 30, etc.)
- **Show Tick Label:** Permiten mostrar una serie de valores del rango según las propiedades Major Tick Unit y Minor Tick Count
- **Show Tick Marks:** Permiten mostrar una marca por cada valor que se muestra según las propiedades Major Tick Unit y Minor Tick Count.
- **Major Tick Unit:** La distancia entre los diferentes valores que se muestran en el slider si Show Tick Label está activado. En la imagen se muestran números de 25 en 25 debido al valor que se ha asignado.
- **Minor Tick Count:** Es el número de ticks que se muestran entre cada rango. Por ejemplo, si en la imagen de arriba se van mostrando valores cada 25 números, por cada uno de estos rangos mostramos 5 líneas o "ticks".

2.11. Hyperlink

Permite añadir un texto con formato de enlace, cuyos estilos y comportamiento se cambian desde el código en Java y a través de CSS.



En SceneBuilder lo más sencillo de cambiar es si queremos que aparezca el enlace como visitado por defecto.



3. Controles simples dinámicos

En esta sección se resumen los controles dinámicos más comunes a la hora de diseñar una interfaz gráfica con JavaFX. Con dinámicos se hace referencia a que, aparte del diseño gráfico, requieren código en Java para añadir datos o personalizar casi todas sus características.

Se explicarán las propiedades específicas de cada uno, ya que los estilos comunes se han descrito en la primera sección.

Para este tipo de controles, necesitaremos trabajar con un **controlador** como se explicaba en la sección 1.4. Además, muchos de estos ítems requieren un listado de valores que se asignan dinámicamente desde el código Java.

Estos valores son de la clase **ObservableList<?>,** cuyo tipo puede ser de cualquier clase. En principio trabajaremos con String, pero más adelante veremos que se puede personalizar con más detalle.

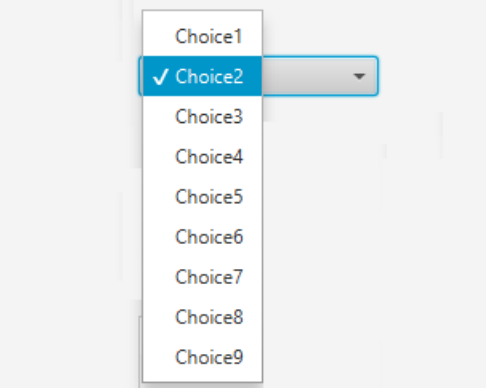
Se requiere una clase específica para JavaFX denominada **ObservableList** que, a diferencia de otras que implementan la interfaz Collection, permite sincronizar la vista creada en el FXML con los datos de nuestro controlador.

A continuación se muestra un ejemplo con las operaciones más comunes para inicializar una serie de controles empleando ObservableList. Esto se hace en el método **initialize de un controlador**.

```
private ObservableList<String> list;

@FXML
private void initialize() {
    // Añadiendo items directamente a un ObservableList
    list.add("Item 1");
    list.addAll("Item 1", "Item2", "Item3");

    // Crear un ObservableList a partir de otra lista de Collections
    ArrayList<String> arrayList1 = new ArrayList<String>();
    arrayList1.add("Item1");
    arrayList1.add("Item2");
    list = FXCollections.observableList(arrayList1);

    // Asignando una ObservableList a un control de JavaFX
    choice1.setItems(list);

    // Controles de JavaFX a los que se añaden directamente los items
    choice1.getItems().addAll("Choice1", "Choice2", "Choice3");
    list1.getItems().addAll("Lista1", "Lista2", "Lista3");
    combo1.getItems().addAll("Combo1", "Combo2", "Combo3");
}
```

3.1. ChoiceBox

Es un control que permite seleccionar un ítem entre varias opciones.



Se puede inicializar mediante Java con el siguiente código dentro del controlador.

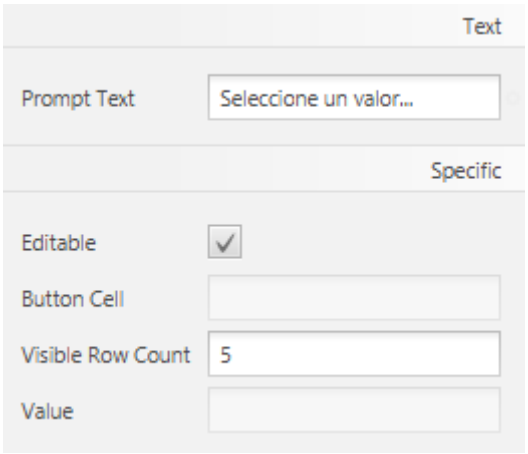
```
public class ListController {
    @FXML
    private ChoiceBox<String> choice1;

    @FXML
    private void initialize() {
        // Controles de JavaFX a los que se añaden directamente los items
        choice1.getItems().addAll("Choice1", "Choice2", "Choice3",
             "Choice4", "Choice5", "Choice6", "Choice7", "Choice8",
             "Choice9");
    }
}
```

3.2. ComboBox

Los **ChoiceBox** tienen una serie de limitaciones. Solo admiten contenido textual y además no permiten asignar valores por defecto o modificar otras propiedades que si es posible con **ComboBox**. Por lo demás, los datos se inicializan de manera muy parecida.

```
public class ListController {
    @FXML
    private ComboBox<String> combo1;

    @FXML
    private void initialize() {
        // Controles de JavaFX a los que se añaden directamente los items
        combo1.getItems().addAll("Combo1", "Combo2", "Combo3",
             "Combo4", "Combo5", "Combo6", "Combo7", "Combo8", "Combo9");
    }
}
```

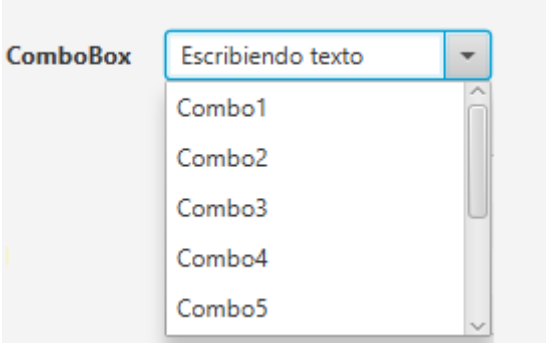
Las propiedades específicas de **ComboBox** son las siguientes:



- **Prompt Text:** Es el texto que se muestra por defecto en un ComboBox.
- **Editable:** Si se marca esta opción, es posible añadir un texto diferente dinámicamente respecto a los ya incluidos en el ComboBox.

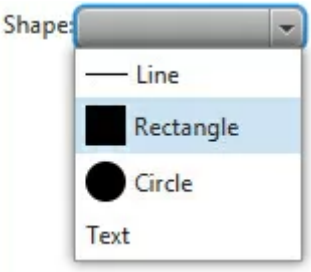
- **Visible Row Count:** Se refiere a la cantidad de filas con valores que se muestran al mismo tiempo.

Un **ComboBox** con las propiedades anteriores según la imagen de arriba tendría el siguiente aspecto (y el texto de cada opción es editable):



El resto de las propiedades se editan desde el controlador:

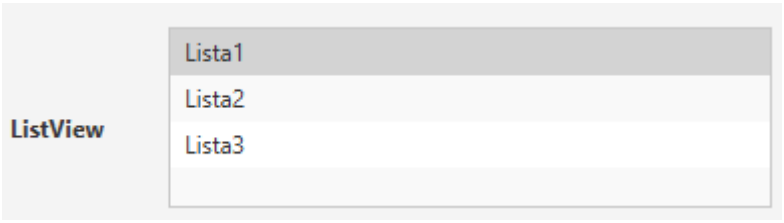
- **Button Cell:** Permite personalizar el contenido de cada ítem del ComboBox para mostrar valores diferentes a texto, como por ejemplo en la siguiente imagen. Lo veremos más adelante.



- **Value:** Se refiere al valor actual. Solamente se puede asignar desde el código en Java.

3.3. ListView

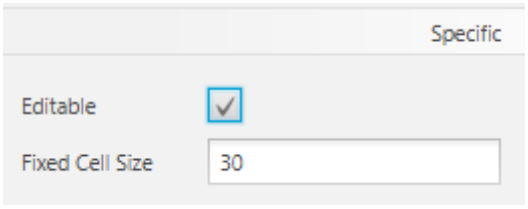
Consiste en un recuadro que permite mostrar un número determinado de ítems. A diferencia de **ChoiceBox** y **ComboBox**, en este caso se pueden visualizar todos a la vez en un listado con un ancho determinado.



El listado se inicializa de manera similar a como hemos visto anteriormente:

```
public class ListController {
    @FXML
    private ListView<String> list1;

    @FXML
    private void initialize() {
        // Controles de JavaFX a los que se añaden directamente los items
        list1.getItems().addAll("Lista1", "Lista2", "Lista3");
    }
}
```

Las propiedades específicas de **ListView** son:

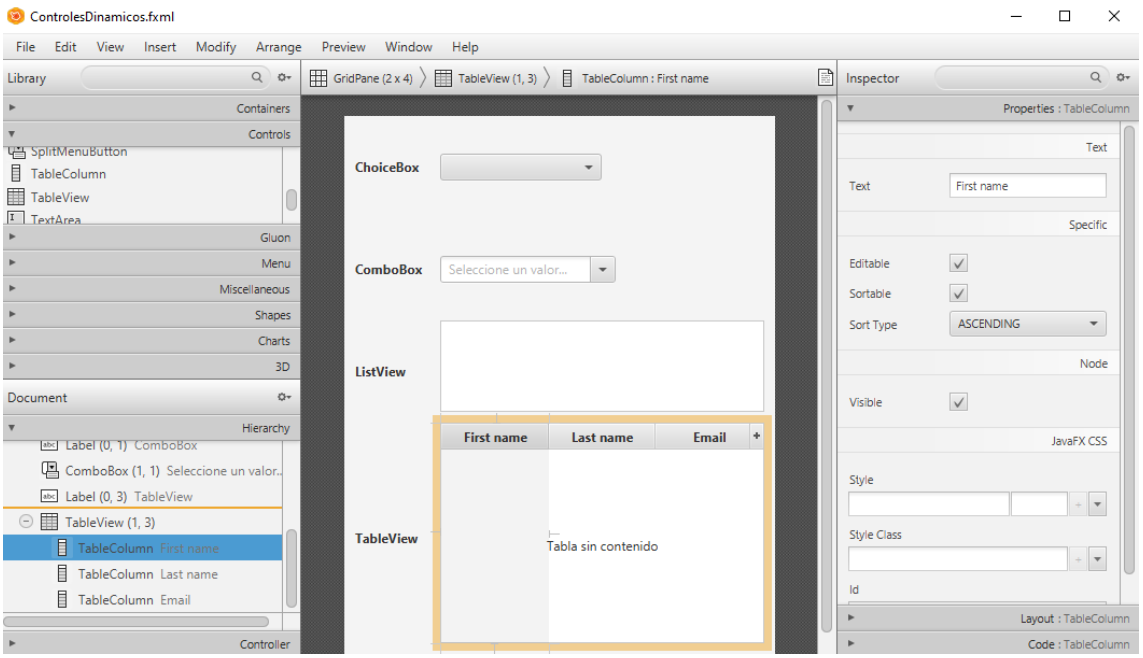


- **Fixed Cell Size:** Permite ampliar la altura de cada ítem mostrado en el listado.
- **Editable:** Permite cambiar el texto de cada ítem del ListView con la aplicación en funcionamiento. Solo funciona empleando CellFactory, que trabaja con la interfaz Callback y permite personalizar la forma de mostrar las celdas. La forma más sencilla de que funcione Editable es inicializar un CellFactory para los ítems de texto de un ListView mediante la línea de código siguiente
  `list1.setCellFactory(TextFieldListCell.forListView());`

Otro aspecto interesante de un **ListView** es que permite seleccionar varios ítems a la vez presionando CTRL o SHIFT. Esto se puede configurar solamente mediante el código Java con la línea de código:
`list1.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);`

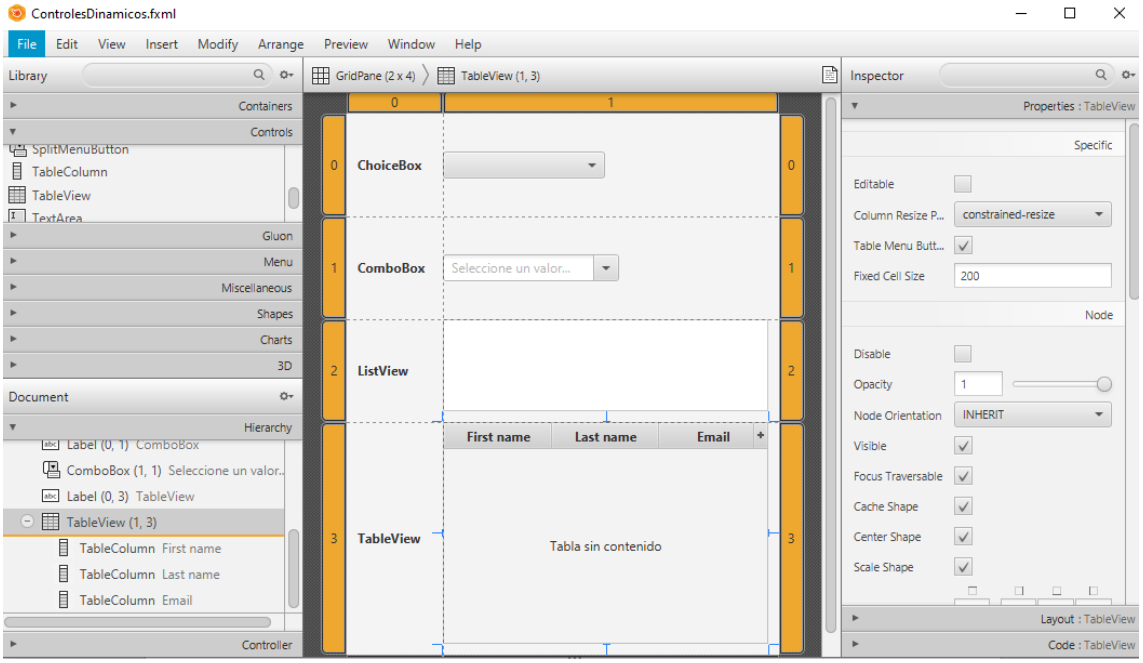
3.4. TableView

Sirve para representar datos en un formato tabular. Es posible crear una tabla desde SceneBuilder arrastrando tantos componentes **TableColumn** como necesitemos tal y como se puede observar debajo. Sin embargo, para que contenga datos necesitaremos de nuevo acceder al código Java.



En la imagen anterior, podemos ver que cada **TableColumn** del **TableView** admite una serie de propiedades que nos permiten indicar el nombre de la columna, si es editable, si el contenido se puede ordenar (sortable) y si se ordena de manera ascendente o descendente (sort type) e incluso podemos crear columnas ocultas con la propiedad visible.

Además, cada tabla presenta una serie de propiedades comunes a todas las celdas:



- **Editable:** Si se permite editar el contenido de la tabla dinámicamente, aunque solo se podrá editar cada columna en función de su propiedad editable específica como veíamos con las propiedades de TableColumn.
- **Column Resize Policy:** Por defecto unconstrained-size, que no rellena el espacio disponible en el ancho de la table, mientras que constrained-size distribuye el ancho equitativamente para todas las columnas.
- **Fixed Cell Size:** El alto fijo de cada celda una vez que contiene datos.

- **Table Menu Button:** Si se activa, permite mostrar u ocultar las celdas que queramos cuando la aplicación está iniciada, tal y como se indica en la siguiente imagen:

El proceso de añadir datos resulta más complejo que en los casos anteriores. En primer lugar, necesitamos tener un **bean con el modelo** que incluya todas las propiedades de la tabla. Por ejemplo, vamos a suponer una clase **Person** con cuatro propiedades: firstName, lastName, email y age.

```
public class Person {
    private final SimpleStringProperty firstName;
    private final SimpleStringProperty lastName;
    private final SimpleStringProperty email;
    private final SimpleIntegerProperty age;

    public Person(String fName, String lName, String email, Integer age) {
        this.firstName = new SimpleStringProperty(fName);
        this.lastName = new SimpleStringProperty(lName);
        this.email = new SimpleStringProperty(email);
        this.age = new SimpleIntegerProperty(age);
    }

    public String getFirstName() {
        return firstName.get();
    }

    public void setFirstName(String fName) {
        firstName.set(fName);
    }

    ...

}
```

Como se puede observar, las propiedades no se corresponden con tipos de datos estándar de Java, sino que tenemos que emplear el equivalente correspondiente a **Property.** Esto ocurre porque clases como String o Integer no son capaces de ser "observables" y no funcionan cuando ocurre alguna modificación una vez que ha arrancado la aplicación gráfica.

Los getter y setter del resto de propiedades serían igual que con firstName (se emplean los get y set de SimpleStringProperty).

Una vez creada esta clase, nos creamos en nuestro controlador una propiedad para la tabla parametrizada con **Person** (o la que hayamos creado en cada caso) y otra por cada columna de la tabla asociada a la clase **Person** y el tipo de datos que va a contener. En este caso se puede incluir String o Integer, ya que en realidad **SimpleStringProperty** o **SimpleIntegerProperty** son subclases de String o Integer y luego se inicializa en Person con las clases que permiten ser observables.

```
public class ListController {
    @FXML
    private TableView<Person> table1;

    @FXML
    private TableColumn<Person, String> firstNameCol;

    @FXML
    private TableColumn<Person, String> lastNameCol;

    @FXML
    private TableColumn<Person, String> emailCol;

    @FXML
    private TableColumn<Person, Integer> ageColumn;

}
```

En el método initialize tenemos que asociar cada columna con la propiedad de la clase Person que corresponda. Esto se puede llevar a cabo con la interfaz **CellFactory** que permite renderizar el contenido de diversos componentes en JavaFX mediante la implementación de la interfaz **Callback**.

En este caso nos basta con emplear el constructor **PropertyValueFactory** que ya implementa esta interfaz.

```
private void initialize() {
    // Asociamos cada columna a una propiedad de la clase Person
    firstNameCol.setCellValueFactory(new
            PropertyValueFactory<Person,String>("firstName"));
    lastNameCol.setCellValueFactory(new
            PropertyValueFactory<Person,String>("lastName"));
    emailCol.setCellValueFactory(new
            PropertyValueFactory<Person,String>("email"));
    ageColumn.setCellValueFactory(new
            PropertyValueFactory<Person,Integer>("age"));

    ...
```

Estamos asociando a cada columna una **implementación de una propiedad de tabla** a las propiedades correspondientes de la clase Person.

Por último, queda rellenar la tabla con datos de manera similar al resto de componentes que hemos visto en este apartado. En este caso se empleará **setItems** y se creará la **observableList** aparte, pero se puede igualmente con **getItems.addAll()**

```
private void initialize() {
    ...

    ObservableList<Person> data = FXCollections.observableArrayList(
      new Person("Jacob", "Smith", "jacob.smith@example.com", 30),
      new Person("Isabel", "Johnson", "isabella.johnson@example.com", 40),
      new Person("Ethan", "Williams", "ethan.williams@example.com", 50),
      new Person("Emma", "Jones", "emma.jones@example.com", 61),
      new Person("Michael", "Brown", "michael.brown@example.com", 34)
    );

    // Se rellena la tabla con objetos de la clase Person
    table1.setItems(data);
}
```

El resultado sería como se muestra a continuación

TableView

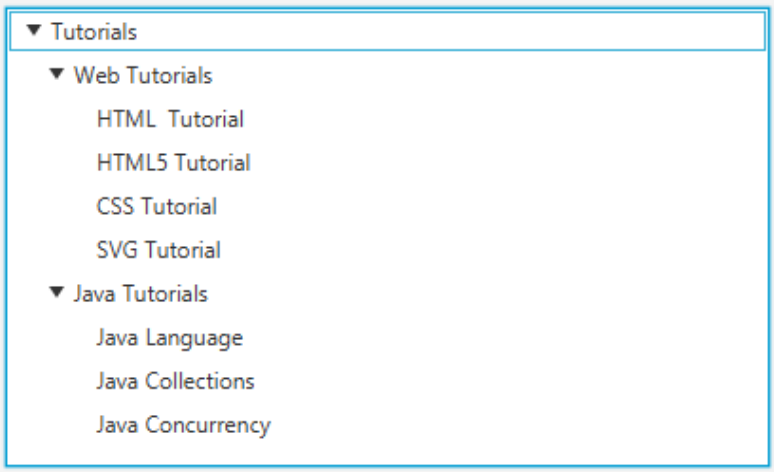
First name	Last name	Email	Age
Jacob	Smith	jacob.smith@ex...	30
Isabella	Johnson	isabella.johnson...	40
Ethan	Williams	ethan.williams@...	50
Emma	Jones	emma.jones@ex...	61
Michael	Brown	michael.brown@...	34

Otra opción para evitar crear una clase sería definir un mapa con pares de valores por cada columna y dato.

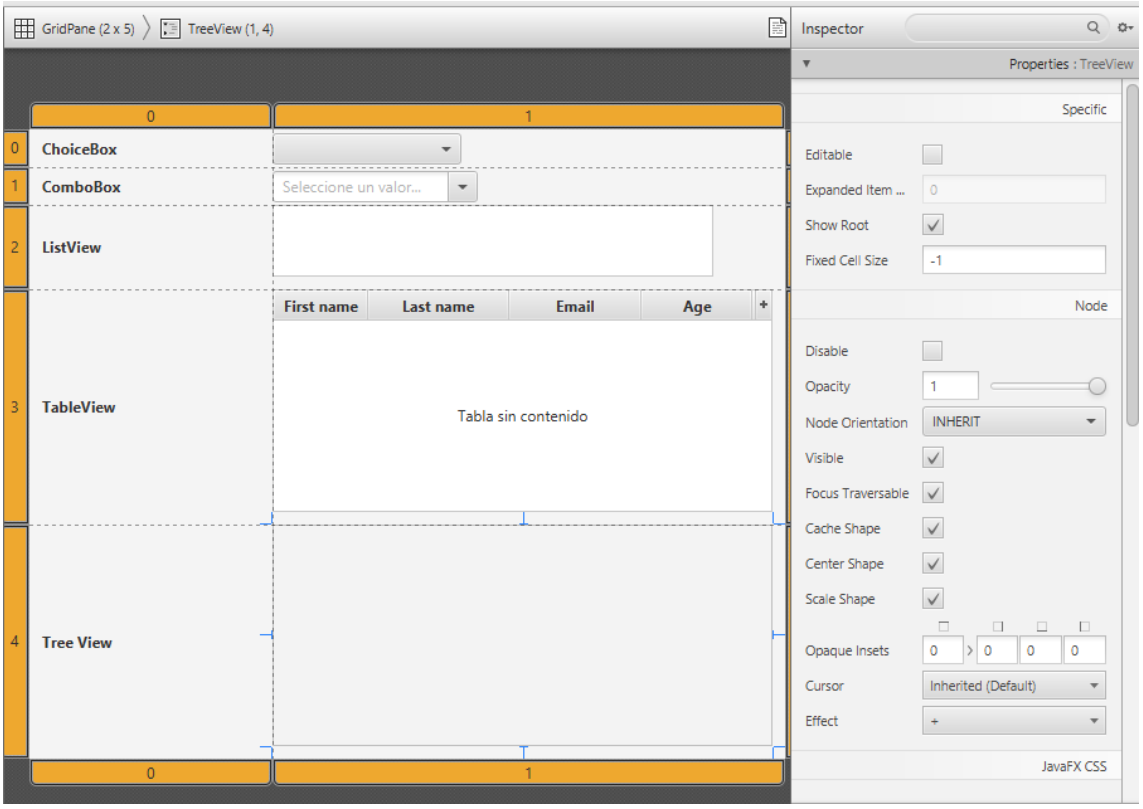
3.5. TreeView

Como su nombre indica, permite crear estructuras jerárquicas en forma de árbol con diferentes nodos denominados padres que contienen otros nodos hijos. Es decir, una estructura similar a la que se muestra a continuación:

Tree View



Las propiedades específicas de **TreeView** son las que se indican a continuación y se explican debajo:



- **Editable:** Permite modificar el contenido del TreeView dinámicamente. Esto solo es posible si cada ítem es a su vez editable (y solo se pueden crear por código Java). Además, es necesario añadir la línea de código Java: `tree1.setCellFactory(TextFieldTreeCell.forTreeView());` (siendo tree1 una propiedad que representa un TreeView).
- **Expanded Item Count:** Es un valor de solo lectura que permite obtener el número de nodos que están expandidos en cada momento. Por ejemplo, si solo aparece el ítem raíz entonces el valor será 1, pero si se expande y tiene tres nodos más será un total de 4.
- **Show Root:** Todos los TreeView deben tener un ítem raíz. Esta propiedad permite indicar si se muestra o no este ítem.
- **Fixed Cell Size:** El alto fijo de cada ítem que contiene datos. Si el valor es igual o menor que 0, entonces el alto se adaptará al contenido.

Por otro lado, para añadir ítems a un **TreeView** es necesario crear un objeto de la clase **TreeItem** por cada nodo que queramos añadir siguiendo el código que se indica debajo.

```
public class ListController {
    @FXML
    private TreeView<String> tree1;

    @FXML
    private void initialize() {
        // Ítems para el TreeView
        // Ítem raíz
        TreeItem<String> rootItem = new TreeItem<String>("Tutorials");

        // Ítem de primer nivel
        TreeItem<String> webItem = new TreeItem<String>("Web Tutorials");
        webItem.getChildren().add(new TreeItem<String>("HTML  Tutorial"));
        webItem.getChildren().add(new TreeItem<String>("HTML5 Tutorial"));
        webItem.getChildren().add(new TreeItem<String>("CSS Tutorial"));
        webItem.getChildren().add(new TreeItem<String>("SVG Tutorial"));
        rootItem.getChildren().add(webItem);

        // Otro ítem de primer nivel
        TreeItem<String> javaItem = new TreeItem<String>("Java
Tutorials");
        javaItem.getChildren().add(new TreeItem<String>("Java Language"));
        javaItem.getChildren().add(new TreeItem<String>("Java
Collections"));
        javaItem.getChildren().add(new TreeItem<String>("Java
Concurrency"));
        rootItem.getChildren().add(javaItem);

        tree1.setCellFactory(TextFieldTreeCell.forTreeView());
        // Expadimos por defecto el ítem raíz
        rootItem.setExpanded(true);
        tree1.setRoot(rootItem);
    }
}
```

Si echamos un vistazo al código, al **TreeView** con nombre tree1 se le asigna un ítem denominado **rootItem.** Siempre es necesario incluir un elemento raíz a partir del cual se crean los demás (en la imagen del comienzo del apartado sería “Tutorials”).

Los ítems se crean con la clase **TreeItem** y se pueden parametrizar con cualquier clase y especificar el texto que se va a mostrar en el constructor (en este caso empleamos String). Por cada ítem, se pueden añadir tantos hijos como queramos con los métodos **getChildren** y **add**. Por ejemplo, estamos añadiendo **webItem** y **javaItem** a **rootItem,** y además webItem y javaItem tienen a su vez otros nodos que creamos con los métodos mencionados anteriormente.

Finalmente, se añade **rootItem** a tree1, un ítem al que le han ido añadiendo diferentes hijos que a su vez tienen varios nodos.

Por último, si echamos un vistazo al código estamos haciendo que los ítems sean editables con la línea `tree1.setCellFactory(TextFieldTreeCell.forTreeView());` (y la propiedad correspondiente del Scene Builder) y el ítem raíz aparece expandido por defecto con el método **setExpanded**

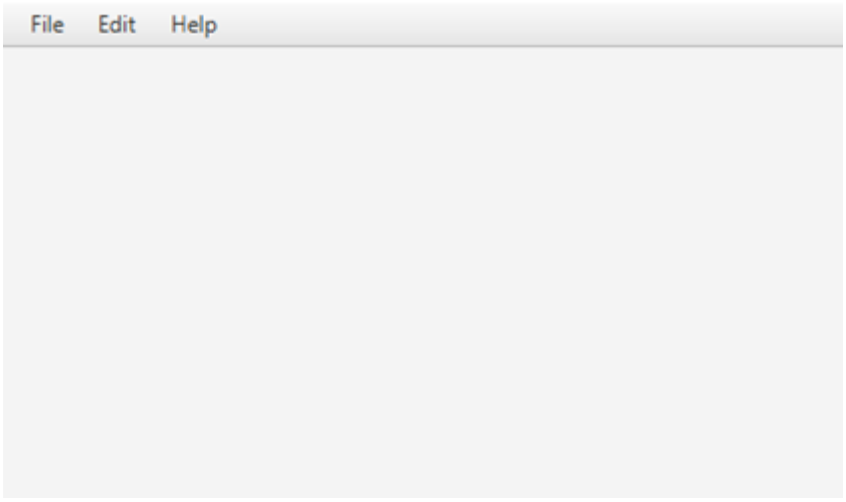
4. Controles gráficos avanzados

En esta sección se expondrán otros controles gráficos más avanzados, que en muchos casos sirven para manejar la navegación en una aplicación o como contenedor para otros controles.

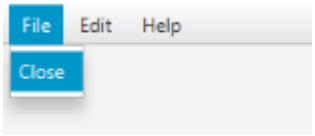
Solamente se explicarán las propiedades específicas de cada uno, ya que los estilos comunes se han descrito en la sección anterior.

4.1. MenuBar

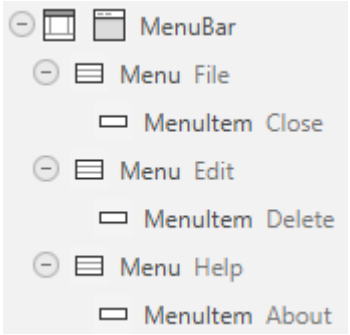
Se trata de un menú horizontal que normalmente se suele incluir en la parte superior de una pantalla, tal y como se muestra en la imagen:



Cada una de las opciones despliega una serie de ítems, por ejemplo:



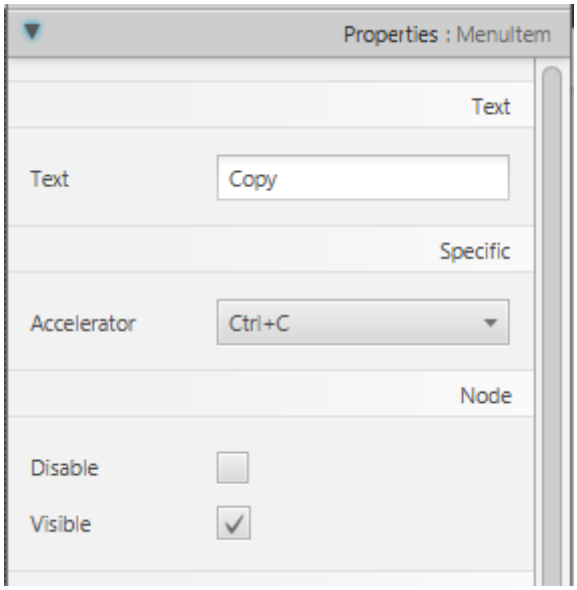
Para añadir estos ítems, se emplean fundamentalmente los controles **Menu** y **MenuItem** creando una estructura similar a la de debajo:



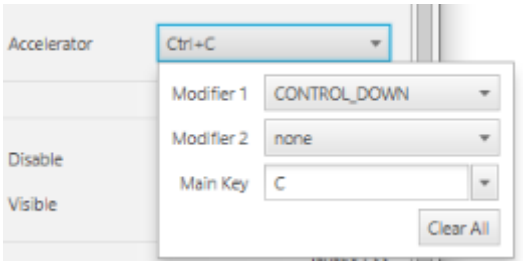
Es decir, cada categoría del menú corresponde a un control **Menu,** mientras que cada ítem dentro de cada categoría es un **MenuItem**. Por ejemplo, en la imagen anterior teníamos las categorías File, Edit y About que se corresponden a los diferentes controles de la clase **Menu.**

Podemos arrastrar tantas categorías e ítems como consideremos.

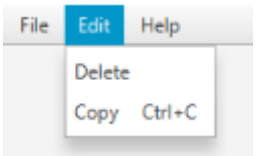
Por otro lado, cada **Menu** y **MenuItem** tienen una serie de propiedades específicas comunes a ambas clases:



- **Text:** Se refiere al texto que se mostrará en cada categoría o menú.
- **Accelerator:** Sirve para indicar combinaciones de teclas para llevar a cabo directamente las acciones asociadas a la categoría o ítem. Por cada combinación de teclas necesitamos un modificador (normalmente presionar CONTROL, ALT o SHIFT) y otra tecla.

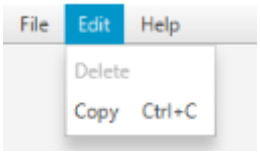


Por ejemplo, en este caso hemos creado un ítem para copiar y pegar. Además, la combinación de teclas se muestra automáticamente en el ítem cuando se visualiza el menú.

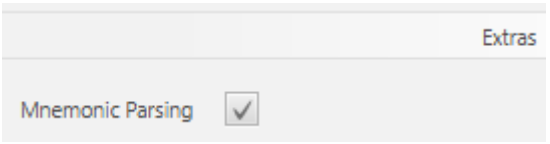


- **Visible:** Permite ocultar una opción de menú por defecto.

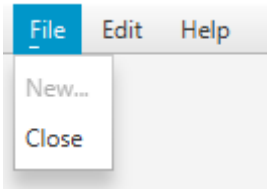
- **Disable:** Permite desactivar una opción de menú por defecto. Por ejemplo:



- **Mnemonic parsing:** Otro valor que aparece al final del todo en la pestaña Properties es "Mnemonic parsing".

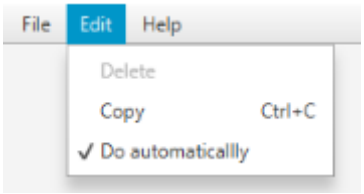


Se emplea especialmente para seleccionar una categoría del menú pulsando "Alt" y la letra por la que empieza. Para ello, además tenemos que añadir el caracter "_" en el texto de cada opción. Por ejemplo, en la categoría "File" tendríamos que cambiar el texto a "_File" y activando esta propiedad se selecciona automáticamente al teclear "Alt+F". Véase el guión bajo cuando se selecciona tecleando Alt.

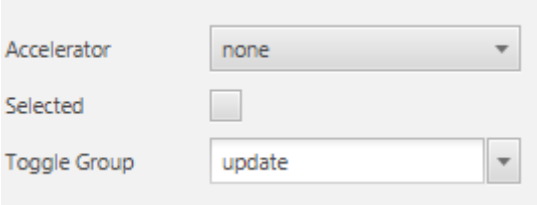


Existen una serie de subclases de **MenuItem** que permiten añadir una serie de funcionalidades. Las propiedades específicas son similares al propio MenuItem y la forma de agregar estos ítems es exactamente igual arrastrando dentro de la categoría creada con **Menu**.

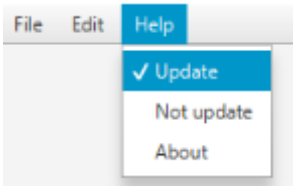
- **CheckMenuItem:** Añade un icono que indica si el ítem está seleccionado o no.



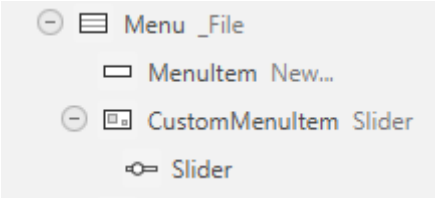
- **RadioMenuItem:** Similar al anterior, pero en este caso se añade la propiedad ToggleGroup.



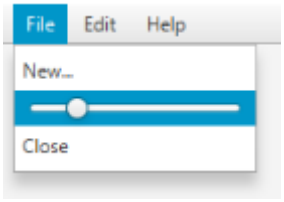
Al igual que en un RadioButton, esto permite que las opciones de menú asociadas al mismo ToggleGroup sean excluyentes. Es decir, no se puede seleccionar a la ver "Update" y "Not update".



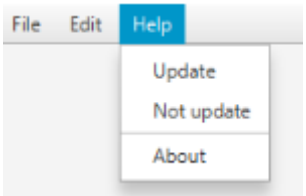
- **CustomMenuItem:** Permite sustituir el contenido textual por otro control. Por defecto se agrega un checkbox, pero podemos reemplazarlo por otro control arrastrándolo en el correspondiente CustomMenuItem.



Por ejemplo, en este caso la opción de menú es un slider que permite directamente asignar el valor que corresponda desde dicho menú.



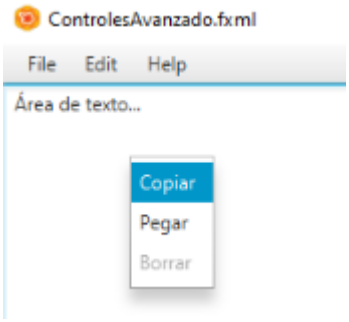
- **SeparatorMenuItem:** Permite añadir una línea separadora entre varias opciones de menú. Basta con arrastrar el componente entre las dos opciones que queremos separar y se obtiene un resultado similar al siguiente:



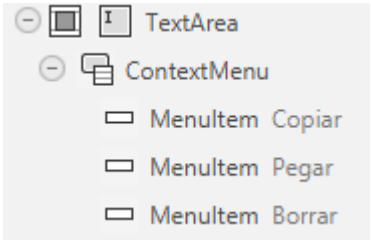
4.2. ContextMenu

IMPORTANTE: Este componente suele presentar problemas con SceneBuilder. Una posible solución es iniciar esta herramienta como Administrador desde el sistema operativo en lugar de emplear el acceso directo de Eclipse. Si el error persiste, la solución sería crear un menú contextual desde el código Java con las clases **ContextMenu** y **MenuItem**. Este menú se añade en el componente que lo queramos insertar con el método **setContextMenu**.

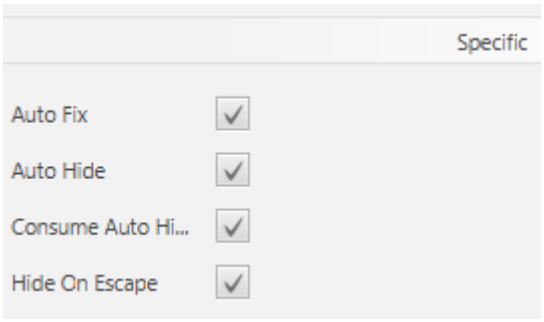
Se trata de un menú contextual típico cuando hacemos click al botón derecho. Hay componentes que ya tienen el menú del sistema operativo por defecto, pero lo podemos sobrescribir.



El funcionamiento es similar a **MenuBar**, aunque en este caso añadimos directamente los ítems de menú sin necesidad de emplear un contenedor de la clase **Menu**. Cualquier ítem de menú estudiado en el apartado anterior sería válido y con las mismas propiedades. Por ejemplo, en la imagen de debajo se incluye en un **TextArea** con el resultado que se indicaba en la captura de arriba.



Las propiedades específicas de **ContextMenu** son:

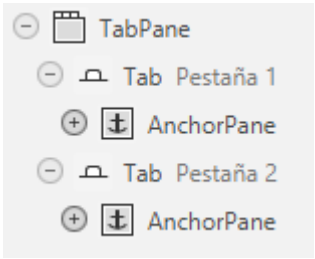


- **Auto Fix:** Permite ajustar el menú contextual para que no se muestre fuera de la aplicación, aunque el sistema operativo ya tiene por defecto funcionalidades que permiten realizar dicho ajuste.
- **Auto Hide:** Si la propiedad se activa, el menú contextual desaparece en cuanto pierde el foco. Si no está activa, el menú contextual no desaparece hasta que no marquemos ninguna opción de este o se pulse la tecla ESC.
- **Consume Auto Hide:** Relacionado con eventos que se estudiará más adelante. Permite especificar si se deben llevar a cabo las acciones asociadas del evento que ha provocado que se ocultara el menú.
- **Hide On Escape:** En principio se utiliza para que un menú contextual desaparezca al teclear ESC. No obstante, en muchas ocasiones la funcionalidad del sistema operativo sobrescribe “Hide On Escape” para ocultar igualmente el menú con ESC aunque no se haya asignado esta propiedad y así evitar que se bloquee la ventana.

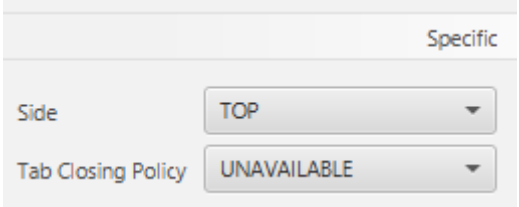
4.3. TabPane

Es un tipo de contenedor que permite dividir la pantalla en pestañas como se muestra a continuación:

Cada pestaña es un elemento de la clase **Tab**, que contiene por defecto un **AnchorPane**, pero se puede modificar por otro layout o contenedor. Podemos añadir tantos elementos de la clase Tab como pestañas necesitemos.



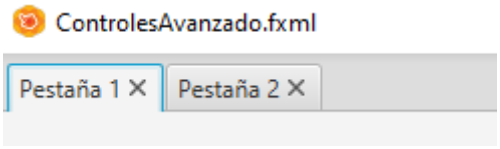
A continuación se indican las propiedades específicas más comunes de la clase **TabPane**



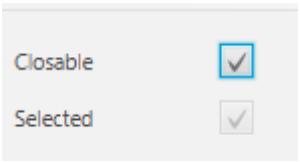
- Side: Permite indicar la posición del TabPane dentro del contenedor al que pertenece. Por defecto de muestra arriba con el valor **TOP**, pero podemos indicar que se sitúe a la

izquierda, derecha o debajo del área de visualización. Por ejemplo, si indicamos **LEFT** se mostraría de la siguiente manera:

- Tab Closing Policy: Permite especificar si se pueden cerrar o no las pestañas. La opción por defecto es **UNAVAILABLE** que no permite cerrar ninguna pestaña. Por otro lado, las opciones **SELECTED_TAB** y **ALL_TABS** sí que lo permiten. SELECTED_TAB permite solo cerrar la pestaña actual, mientras que ALL_TABS muestra el icono de cerrar las pestañas en todas aunque no se hayan seleccionado.

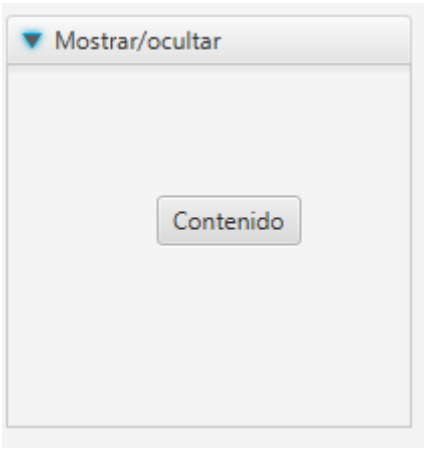


Para poder cerrar una pestaña es necesario activar la propiedad **Closable** de la clase **Tab**.

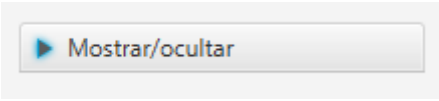


4.4. TitledPane

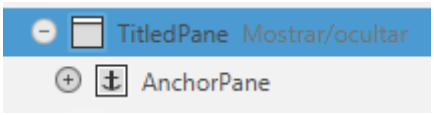
Se trata de un contenedor que muestra un título y que permite visualizar u ocultar el contenido tal y como se muestra en el ejemplo que se indica a continuación.



Si hacemos click en el icono con la flecha a la izquierda del texto, deja de aparecer el botón que hemos incluido en el contenido.

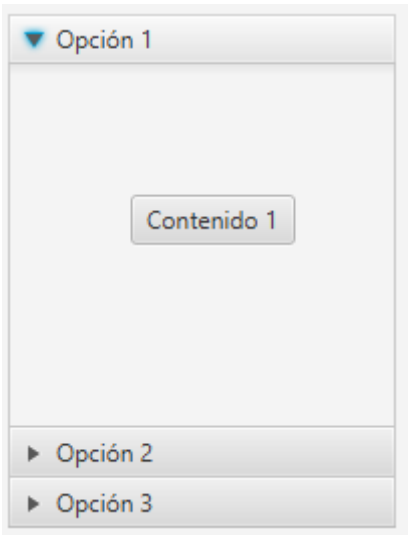


Un **TitledPane** contiene por defecto un **AnchorPane**, pero se puede configurar con cualquier otro layout o contenedor.

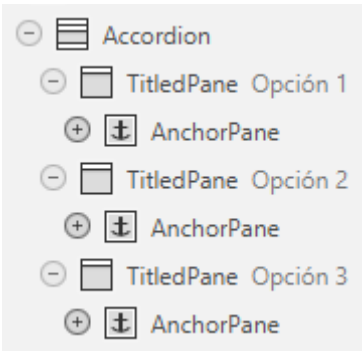


4.5. Accordion

Se trata de un contenedor que puede contener múltiples secciones que se pueden expandir u ocultar, solamente mostrando una de ellas al mismo tiempo. El formato es similar a la imagen de debajo:



Si hacemos click en “Opción 2” y “Opción 3” se oculta el contenido de “Opción 1” y se muestra el que hayamos seleccionado en cada momento. Como se puede observar, cada opción es un **TitledPane** con el mismo formato que el apartado anterior.



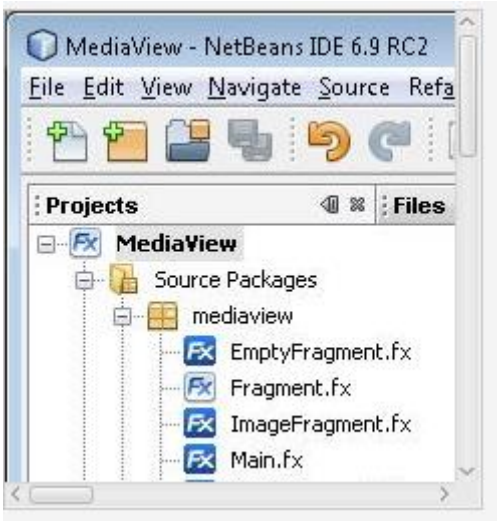
Podemos añadir un **TitledPane** por cada sección nueva que sea necesaria. En este caso no tenemos propiedades específicas y cada **TitledPane** se añade debajo del anterior.

Sí que es interesante tener en cuenta que un **Accordion** consta de varios **TitledPane** que a su vez contienen un **AnchorPane** (u otro contenedor). Las dimensiones del contenido se adaptarán al ancho y alto establecido en el **Accordion** y aunque cambiemos, por ejemplo, el alto de un **AnchorPane,** no afectará y se adapta al tamaño del propio acordeón.

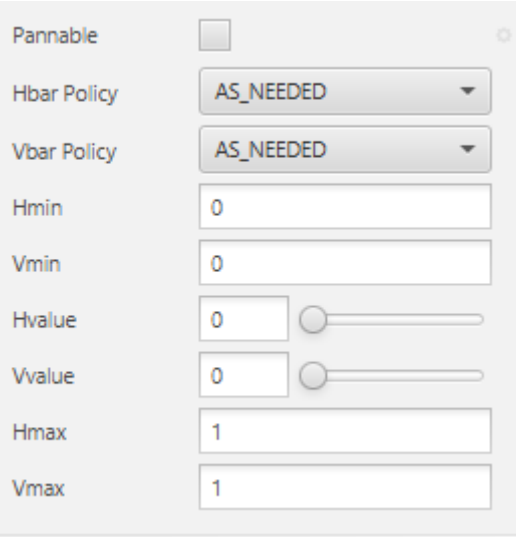
4.6. ScrollPane

Se trata de un contenedor que permite añadir un scroll cuando el tamaño del contenido desborda las dimensiones de dicho contenedor.

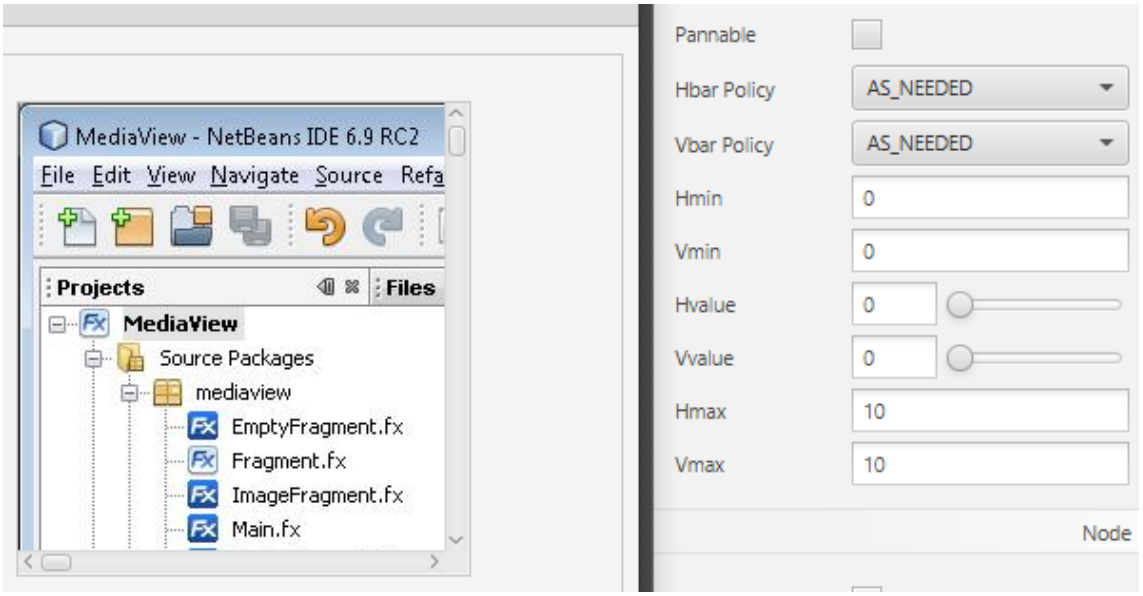
Por ejemplo, a continuación se añade una imagen dentro de un **ScrollPane** más grande que el contenedor, por lo que aparecen las barras verticales y horizontales.



Las propiedades específicas de un **ScrollPane** son las siguientes:



- **Pannable:** Por defecto no está activa y solo permite al usuario acceder al contenido oculto del ScrollPane con la barra horizontal o vertical. Si se activa esta propiedad, el usuario también puede arrastrar el contenido con el ratón.
- **Hbar Policy y Vbar Policy:** Permite especificar cuándo queremos que se muestran las barras del scroll. Tenemos las siguientes opciones que se pueden configurar por separado para la barra horizontal o vertical (Hbar o Vbar):
    - NEVER: No se muestran nunca, aunque el contenido exceda el contenedor.
    - ALWAYS: Se muestran siempre, aunque no sea necesario porque el contenido es más pequeño.
    - AS_NEEDED: Solo se muestran las barras cuando se detecta que el contenido sea más grande que el contenedor.
- **Hmin, Vmin, Hmax y Vmax:** Permite controlar el rango de valores del scroll horizontal y vertical por separado. Por defecto el rango es de 0 a 1, pero si ampliamos el rango, por ejemplo cambiando Hmax y Vmax, el scroll será más pequeño que en la imagen del comienzo del apartado y tendremos más margen de movimiento.

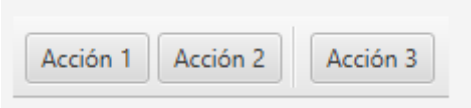


- **Hvalue y Vvalue:** Podemos inicializar el scroll dentro de una posición determinada, siempre que sea en el rango de las propiedades anteriores.

El elemento **ScrollPane** tiene otras propiedades específicas dentro de la pestaña **Layout.**

4.7. ToolBar

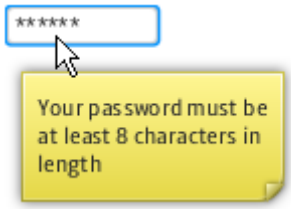
Permite crear la típica barra de tareas como de un sistema operativo agrupando diferentes controles de izquierda a derecha o de arriba a abajo (según la propiedad ORIENTATION).



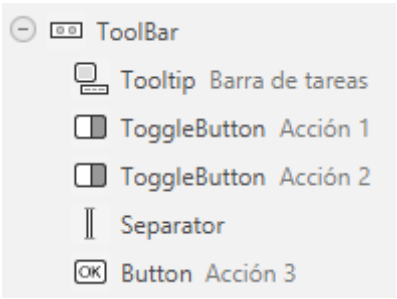
En cierto modo es muy similar a **HBox** o **VBox**, salvo por algunos estilos que se añaden por defecto y otras propiedades específicas (por ejemplo el borde inferior que aparece en la imagen).

4.8. ToolTip

Se trata del texto informativo adicional que suele aparecer cuando nos posicionamos por encima de algún elemento.



Para crear un **ToolTip** se puede arrastrar prácticamente dentro de cualquier otro control de JavaFX. Por ejemplo, en la imagen de debajo se crea un **ToolTip** con el nombre “Barra de tareas” dentro de un control ToolBar (que contiene otros elementos).



El resultado sería el siguiente con el texto explicativo correspondiente: