

# JavaFX

<b>Introducción</b>	<b>1</b>
<b>JavaFX e IntelliJ IDEA</b>	<b>1</b>
Configuración de IntelliJ	4
<b>Características de JavaFX</b>	<b>4</b>
Escenario - Stage	5
Escena - Scene	5
Grafo de escena - Scene Graph	5
Nodos	6
<b>Aplicación JavaFX</b>	<b>7</b>
<b>Paneis de Layout</b>	<b>10</b>
<b>Controis</b>	<b>14</b>
<b>Eventos</b>	<b>16</b>
<b>Novo proxecto JavaFX FXML</b>	<b>18</b>
<b>JavaFX FXML</b>	<b>23</b>
Obxectos en FXML	23
Espazo de nomes	24
Clase controladora FXML	25
FXML estilos CSS	26
FXML e IntelliJ	26
<b>Diálogos</b>	<b>27</b>
Alert	28
<b>Propiedades</b>	<b>30</b>
<b>Propiedades e Clases</b>	<b>32</b>
<b>JavaFX Collections</b>	<b>34</b>
<b>TableView</b>	<b>35</b>
<b>Scene Builder</b>	<b>37</b>
<b>CSS</b>	<b>41</b>
<b>Referencias</b>	<b>44</b>

# Introdución

JavaFX é unha API deseñada para crear interfaces gráficas de usuario (GUI) que vén a substituír a tecnoloxía Swing. Proporciona unha interface moderna e altamente portable, que pode incluír audio, vídeo, gráficos e animación.

Foi deseñado tendo en conta o patrón **MVC**, que mantén separado o código que xestiona os datos do código da interface gráfica. O controlador é unha especie de intermediario entre os datos e a interface e controla o que sucede cando unha persoa interacciona coa interface.

Ao traballar con JavaFX, non é obrigatorio seguir o patrón MVC, mais é unha practica recomentable a seguir. No patrón MVC, o modelo está composto os datos do modelo da aplicación, a vista está formada polos ficheiros fxml e o controlador é o código que determina que pasa cando unha persoa interacciona coa interface.

Entre as principais características de JavaFX están: soporta estilos mediante CSS, proporciona aceleración gráfica por hardware, permite aplicar efectos e animacións de forma fácil, integra gráficos 2D e 3D, a interface pode ser construída usando Java ou arquivos FXML, provee soporte multimedia para a reprodución de audio e vídeo, etc.

Hai varias razóns polas que JavaFX é unha plataforma boa para crear aplicacións gráficas. Primeiramente porque Java aínda é unha das linguaxes de programación máis populares. Ademais, as aplicacións JavaFX poderán executarse en diferentes sistemas operativos e dispositivos: Windows, Linux, Mac, iOS, Android / Chromebook, Raspberry Pi, móbiles, televisores, navegadores, etc.

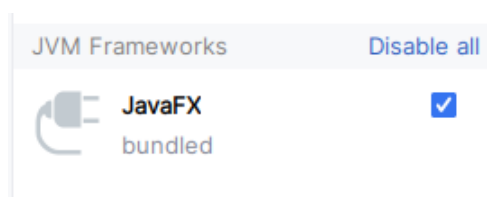
JavaFX permite deseñar a interface gráfica usando código Java ou usando a linguaxe FXML, que é unha linguaxe de marcas baseada en XML na que se definirán os obxectos gráficos e compoñentes da interface. Para isto Oracle proporciona a ferramenta Scene Builder que é un editor visual para FXML.

## JavaFX e IntelliJ IDEA

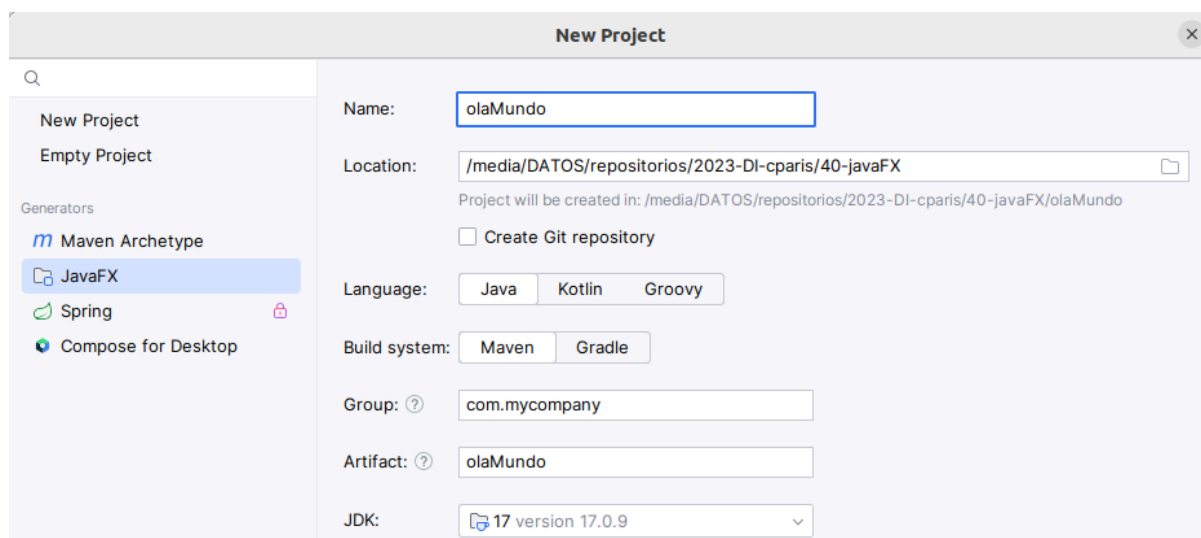
Neste apartado descríbense os pasos para crear unha aplicación JavaFX usando IntelliJ IDEA.

IntelliJ IDEA inclúe soporte para JavaFX, o que implica que proporcionará axuda para completar código, buscar, navegar e refactorizar ficheiros, integración con JavaFX Scene builder, empaquetado de aplicacións, etc.

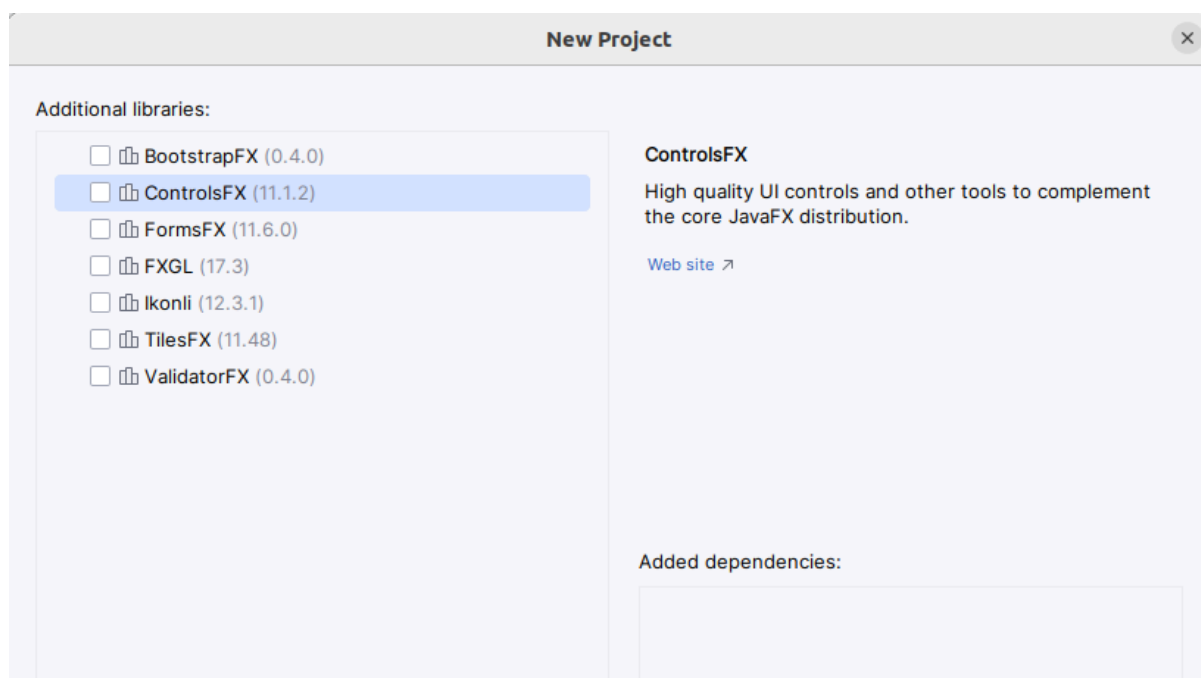
Antes de nada hai que comprobar que o plugin de JavaFX estea habilitado. Acceder a **Settings (Ctrl + Alt + S)** e seleccionar **Plugins**. Comprobar, na pestana **Installed** que o plugin JavaFX está habilitado.



Unha vez comprobado que o plugin de JavaFX está instalado crearase un novo proxecto dende o menú **File -> New -> Project**. Da lista de xeradores de proxectos escollerase **JavaFX** e completaranse os datos do nome do proxecto, ruta, Group (nome do paquete que se creará co proxecto), JDK e pulsarase **Next**.




No seguinte paso seleccionaranse as librerías a usar na aplicación. De momento non se seleccionará ningunha e pulsarase o botón **Create**.

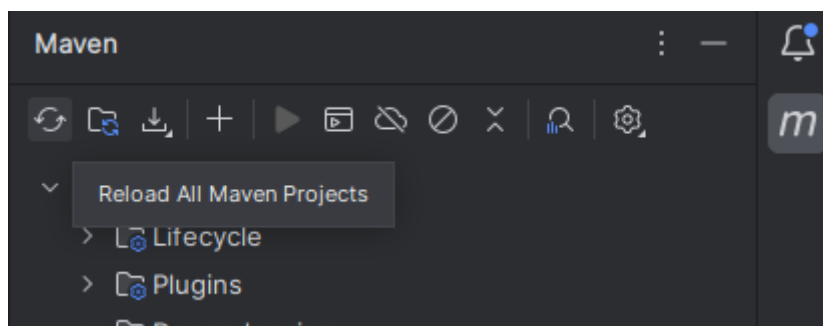


**NOTA:** recordar actualizar o ficheiro **.gitignore** para que só se suban ao repositorio os ficheiros necesarios.

Despois de crear o proxecto, verificar o ficheiro **pom.xml** e comprobar que inclúe a dependencia **javafx.controls**. Comprobar tamén que inclúe o **javafx-maven-plugin**. Unha vez feito isto xa se pode executar o proxecto.

Para executar a aplicación haberá que abrir o ficheiro **HelloApplication.java** e pulsar na icona . Cando a compilación do código remate, mostrarase a ventá da aplicación.

**NOTA:** se o proxecto dá algún erro pode ser necesario recargalo para que se descarguen e actualicen todas as dependencias pulsando no primeiro botón da barra de ferramentas de Maven.



Observar que o proxecto creado ten unha clase denominada **HelloApplication** que estende **Application**. **Application** é a clase básica que permite crear unha aplicación JavaFX e xestiona o ciclo de vida da aplicación.

O método **main** é o punto de inicio do proxecto de NetBeans. Para iniciar a aplicación JavaFX dende o main, hai que chamar ao método **Application.launch()**, que é o encargado de lanzar a aplicación JavaFX.

```
public class HelloApplication extends Application {
    @Override
    public void start(Stage stage) throws IOException {
        FXMLLoader fxmlLoader =
            new FXMLLoader(HelloApplication.class.getResource("hello-view.fxml"));
        Scene scene = new Scene(fxmlLoader.load(), 320, 240);
        stage.setTitle("Hello!");
        stage.setScene(scene);
        stage.show();
    }

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

A clase **HelloApplication**, que estende de **Application**, debe ter un construtor baleiro (sen parámetros) para poder iniciarse correctamente. Recordar que cando non se inclúe un construtor, o compilador engade un construtor por defecto sen parámetros automaticamente.

A clase **Application** xestiona o ciclo de vida da aplicación, para o que define algúns métodos como **init**, **start** e **stop**, que se poderían sobrescribir para programar algunhas accións.

Cando se lanza a aplicación (**launch**) primeiro execútase o método **init** e despois o método **start**. Dado que **start** é un método abstracto é obrigatorio sobreescribilo, e será aquí onde se cree a interface. O método **stop** execútase cando a aplicación termina e serve para executar tarefas de finalización e liberación de recursos.

Ao método **start** pásaselle como parámetro un obxecto **Stage**, que é un contedor de nivel superior de JavaFX e estende a clase `Window`. Este obxecto `Stage` é creado automaticamente e representa a **ventá principal**, equivalente ao `JFrame` de Swing. Aínda que é posible crear máis `Stages`, a maioría das aplicacións só terán unha ventá principal.

O método **start** carga a interface gráfica definida no ficheiro **hello-view.fxml**, o que creará todos os obxectos definidos. Este ficheiro está escrito en FXML, que é o formato XML específico de JavaFX. O código deste ficheiro será explicado en detalle ao longo deste documento.

Por último, establécese o título do stage (título da ventá principal), establécese a escena e fai visible o stage.

## Configuración de IntelliJ

É interesante ter configurado o IDE adecuadamente para desenvolver código da forma máis rápida posible.

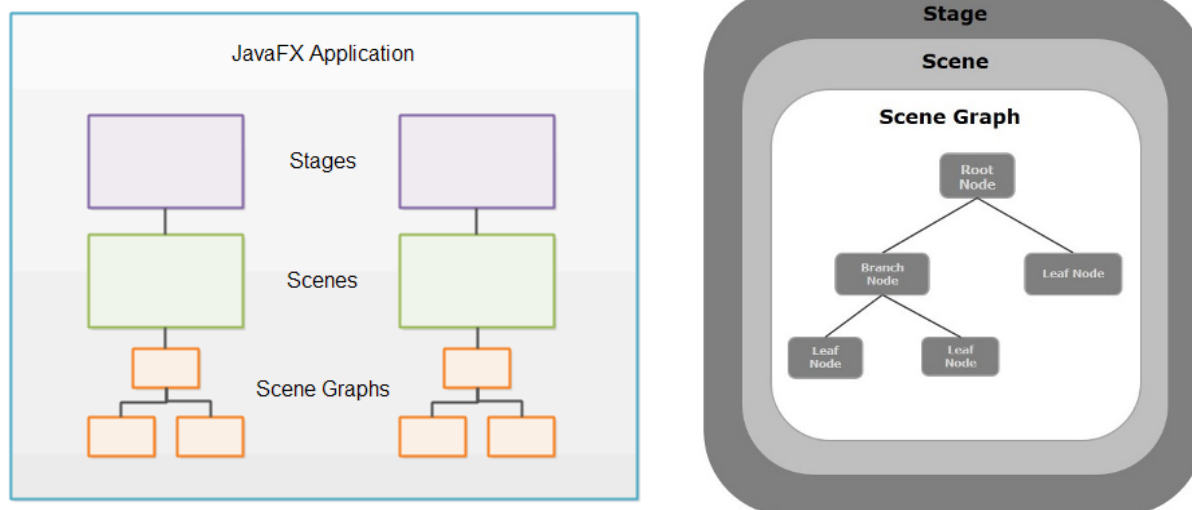
Algunhas configuracións interesantes son:

- Engadir imports automaticamente: Settings -> Editor -> General -> Auto Import -> Add unambiguous imports on the fly
- Accións a realizar cando se garda un ficheiro: Settings -> Tools -> Actions on Save. Poden activarse opcións como “Reformat code”, “Optimize imports”, “Rearrange code”, ...
- Modificar os espazos da tabulación: Settings -> Editor -> Code Style -> Java -> Tabs and Indents. Especificar o tamaño da tabulación.
- Permitir cambiar o tamaño da fonte utilizando Ctrl+Mouse Wheel: Settings -> Editor -> General -> Change font size with Ctrl+Mouse Wheel.

## Características de JavaFX

En xeral, unha aplicación JavaFX contén un escenario (**stage**) que se corresponde coa ventá principal. Cada **stage** ten unha escena (**scene**) asociada. E cada **scene** ten un **grafo de escena** ou árbore, onde cada nodo se corresponde con un control ou área da interface gráfica.

Utilízase o símil do teatro, onde hai un escenario no que se representan varias escenas ao longo de toda a obra.



## Escenario - Stage

O **stage** é o contedor de nivel superior en JavaFX. É o frame externo dunha aplicación, equivalente ao JFrame de Swing. Normalmente correspóndese con unha ventá. Se unha aplicación ten varias ventás, cada ventá terá o seu propio **stage**.

No caso de haber varias ventás, un escenario pode ter, de forma opcional, unha ventá/stage **proprietaria**. Neste caso dise que é a ventá pai. A ventá propietaria debe ser inicializada antes de que o escenario se mostre.

As ventás descendentes están ligadas á ventá pai e sempre se situarán por encima desta. Cando a ventá pai se cerra ou minimiza, todas as ventás descendentes veranse afectadas.

## Escena - Scene

Para poder mostrar algo nun escenario (**stage**) é necesario ter unha escena (**scene**).

Un **escenario** só pode mostrar unha **escena** ao mesmo tempo, sen embargo é posible cambiar de escena en tempo de execución.

Unha escena é representada polo obxecto **Scene** nunha aplicación JavaFX.

O escenario é o contedor de nivel superior, a ventá principal da aplicación. A escena é o contedor para o contido visual do escenario.

## Grafo de escena - Scene Graph

Todos os compoñentes visuais (controis, layouts, etc.) deben estar asociados a unha escena para poder ser mostrados, e a escena debe estar asociada con un escenario para ser visible.

Un **grafo de escena** é unha estrutura de datos en **árbore** que contén todos os obxectos (controis, layouts, etc.) da escena.

En calquera momento, o grafo sabe que obxectos debe mostrar, que áreas da pantalla necesitan repintarse e como renderizar todo da maneira máis eficiente.

En lugar de invocar métodos de debuxo de baixo nivel, as persoas programadoras usan a API do grafo de escena e o sistema xestiona automaticamente os detalles de renderización, o que diminúe a cantidade de código necesario na aplicación.

## Nodos

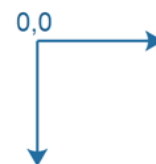
Os elementos individuais agregados a un grafo de escena son denominados **nodos**. Todos os nodos son subclases de [javafx.scene.Node](#).

Hai dous tipos de nodos: **rama** ou **folla**. Unha nodo rama é un nodo que pode conter outros nodos descendentes. Un nodo folla é un nodo que non pode conter outros nodos.

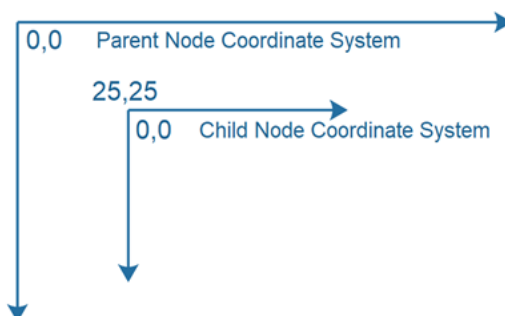
O primeiro nodo da árbore é chamado o **nodo raíz** e non ten ascendentes.

Cada nodo só pode ser engadido unha vez ao grafo, é dicir, só pode aparecer nun único sitio do grafo. Se se intenta engadir a mesma instancia dun nodo varias veces ao grafo, JavaFX lanzará unha excepción.

Cada nodo ten o seu propio sistema de coordenadas. A diferenza do sistema de coordenadas cartesianas estándar, o eixe Y está invertido. Isto significa que a orixe de coordenadas (0,0) está situada na esquina superior esquerda do nodo.



Este sistema de coordenadas tamén é utilizado para distribuír e posicionar os descendentes en relación co nodo pai. É dicir, cada nodo fillo terá o seu propio sistema de coordenadas e terá unha posición relativa dentro do sistema de coordenadas do nodo pai.



As implementacións concretas de nodos inclúen controis (botóns, táboas, caixas de texto, ...), layout managers (xestores de contido), imaxes, paneis, etc.

# Aplicación JavaFX

Para crear unha aplicación gráfica JavaFX pode facerse de varias formas:

- Usar un arquivo FXML que define os elementos que compoñen a interface. Pode usarse a ferramenta SceneBuilder para crear o arquivo FXML
- Programar a interface usando código da API de JavaFX.

Empezaremos usando a última opción.

- En JavaFX un **Stage** representa unha ventá.
- Esta ventá pode conter só un escenario (**Scene**), que é un contedor para os elementos que forman a interface e están organizados nunha árbore xerárquica (**Scene Graph**). Coma no teatro, un escenario só pode conter unha escena á vez. Cando a historia avanza, a escena pode cambiar.
- Os paneis, como VBox, GridPane, StackPane, etc. poden usarse como **nodo raíz** do grafo de escena e serven de contedores para o resto dos elementos da escena.
- Os elementos que contén un Scene, deben estender da clase Node.

Como exemplo, reescribírase a aplicación creada no apartado [JavaFX e IntelliJ IDEA](#) programando o código usando JavaFX:

```
@Override
public void start(Stage stage) throws IOException {
    VBox vbox = new VBox(20);
    vbox.setAlignment(Pos.CENTER);

    Label label = new Label();
    vbox.getChildren().add(label);

    Button boton = new Button("Hello!");
    boton.setOnAction((e) -> label.setText("Welcome to JavaFX Application!"));
    vbox.getChildren().add(boton);

    Scene scene = new Scene(vbox, 320, 240);
    stage.setTitle("Hello!");
    stage.setScene(scene);
    stage.show();
}
```

Observar que o código anterior é equivalente á creación da interface usando FXML.



Outro exemplo:

```
public class HelloWorld extends Application {

    @Override
    public void start(Stage stage) {

        Button btn = new Button("Saudar");
        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent t) {
                System.out.println("Ola mundo!");
            }
        });

        // ROOT NODE
        StackPane root = new StackPane();
        root.getChildren().add(btn);

        // o scene contém o ROOT NODE
        Scene scene = new Scene(root, 300, 250);

        // configurar e mostrar o stage
        stage.setTitle("Ola mundo!");
        stage.setScene(scene);
        stage.show();
    }

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

Outro exemplo:

```
public class App extends Application {

    @Override
    public void start(Stage stage) {
        TextField txt = new TextField();
        txt.setPromptText("Escribe o teu nome");

        Button btn = new Button("Saudar");
        btn.setOnAction(e -> System.out.println("Ola: " + txt.getText()));

        // ROOT NODE, este contém e organiza o botón e o cadro de texto
        VBox root = new VBox(10.0);
        root.setPadding(new Insets(10.0));
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(btn);
        root.getChildren().add(txt);
    }
}
```

```

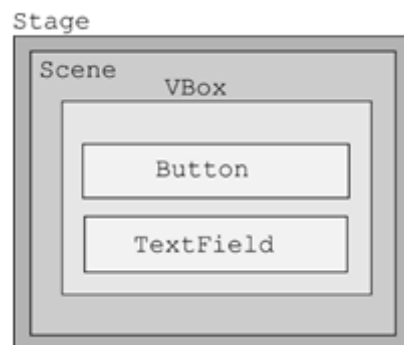
// o scene contén o ROOT NODE
Scene scene = new Scene(root, 300, 250);

// configurar e mostrar o stage
stage.setTitle("Ola JavaFX");
stage.setScene(scene);
stage.show();
}

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

```

O exemplo anterior crea unha aplicación cos seguintes obxectos:



Unha aplicación JavaFX remata chamando ao método `Platform.exit()` ou cando se cerra a última ventá visible.

**NOTA:** fixarse que se importan os controis de `javafx` e non os de `awt`.

Cando se utiliza un manexador de eventos, é posible acceder ao elemento onde se orixinou o evento a partir o obxecto `Event`:

```

btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent t) {
        Button b = (Button) e.getTarget();
        ....
    }
});

```

### Exercicios:

1. Deseña unha aplicación JavaFX que teña un botón e unha etiqueta. Decide ti o deseño visual.

Inicialmente a etiqueta non ten ningún texto asignado. Cada vez que se pulse o botón, a etiqueta debe mostrar unha mensaxe informando do número de clicks realizados.

Configura a etiqueta para que o tipo de letra sexa Arial, cor azul e grosa.

- Engade á túa aplicación un botón Saír. Para saír da aplicación é necesario executar o seguinte código:

**Platform.exit();**

- Tooltip:** Engade a un dos botóns da aplicación anterior un Tooltip con unha frase explicativa do seu funcionamento.
- Mnemonics:** Engade a un dos botóns da aplicación anterior un mnemonic, de tal forma que pulsar a combinación de teclas Alt+"Letra inicial texto" sexa equivalente a pulsar o botón.

## Paneis de Layout

Os paneis de layout son contedores usados para xestionar a posición dos controis do grafo. Cando a ventá é redimensionada, o panel automaticamente recoloca e redimensiona os nodos que contén.

A principal ventaxa de usar paneis de layout é que o tamaño e aliñamento dos nodos é xestionado polo panel. Se o panel se redimensiona, os nodos tamén se redimensionan, tendo en conta que non todos os nodos son redimensionables

En JavaFX todos os controis teñen un **tamaño preferido** (*preferred size*) que se calcula en función do seu contido. Por exemplo, para mostrar un botón en pantalla o seu tamaño calcularase en función do seu contido (texto e iconas).

Os controis tamén proporcionan un **tamaño mínimo e máximo** que están baseados no uso típico do control. Por exemplo, o tamaño máximo dun botón é igual ao seu tamaño preferido, xa que normalmente non se quere que os botóns se redimensionen. Pode cambiarse este comportamento configurando o tamaño mínimo e máximo do compoñente a valores diferentes aos establecidos por defecto.

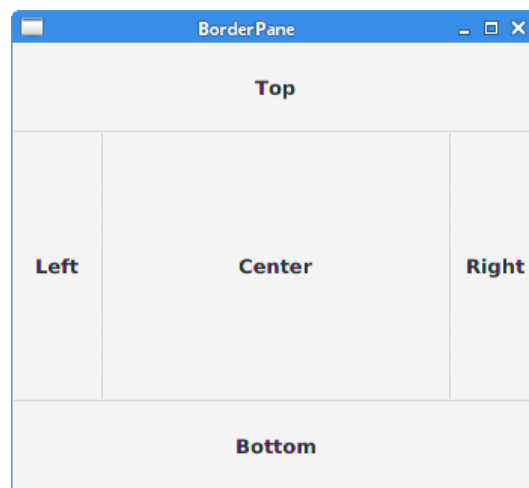
Cando se coloca un control nun layout, o control pasará a ser **fillo** do layout, que se encargará de visualizalo adecuadamente.

JavaFX ten, entre outros, os seguintes paneis de layout:

- **[StackPane](#):** coloca os nodos nunha pila de atrás cara adiante. O último nodo engadido é o da cima. [Exemplo](#).
- **[HBox](#)** e **[VBox](#):** os elementos colócanse nunha fila horizontal ou nunha columna vertical. [Exemplo](#).
- **[FlowPane](#):** posiciona os nodos nunha fila ou columna. Un nodo pasará a unha nova fila ou columna cando non collan máis elementos na fila ou columna. A orientación por defecto dun flowPane é horizontal, aínda que pode modificarse.
- **[AnchorPane](#):** permite ancorar os bordes dos fillos a unha distancia predefinida do borde do panel contedor.

- **[TilePane](#)**: coloca os nodos de forma que todos teñen o mesmo tamaño. Por defecto, os nodos son colocados horizontalmente. Tamén pode configurarse para que se coloquen nunha columna e incluso nunha matriz. [Exemplo](#)
- **[BorderPane](#)**: coloca os elementos nas posicións superior, inferior, dereita, esquerda ou centro.

Os elementos do panel superior e inferior ocupan o alto necesario e todo o ancho. Os elementos do panel esquerdo e dereito ocupan o ancho necesario e todo o alto. O elemento central ocupará o resto do espazo dispoñible.



- **[GridPane](#)**: posiciona os elementos nunha malla de filas e columnas. Os nodos poden estenderse por múltiples filas e columnas.

Pode engadirse un elemento ao grid indicando a columna e a fila onde se colocará e o número de filas/columnas polos que se expandirá:

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

GridPane gridPane = new GridPane();

// add(Node child, int columnIndex, int rowIndex, int colspan, int rowspan)
gridPane.add(button1, 0, 0, 1, 1);
gridPane.add(button2, 1, 0, 1, 1);
```

Tamén pode establecerse o espacio horizontal e vertical entre compoñentes dun GridPane usando os métodos [setHGap\(\)](#) e [setVGap\(\)](#):

```
gridPane.setHgap(10);
gridPane.setVgap(10);
```

Por defecto, o tamaño das filas e columnas axústase ao contido. Pode configurarse o grid para que as filas/columnas se redimensionen. Por exemplo, se é necesario que unha columna poida medrar, pode usarse o seguinte código:

```
// establecer o tamaño mínimo, preferido e máximo.
ColumnConstraints column1 = new ColumnConstraints(100,100,Double.MAX_VALUE);
column1.setHgrow(Priority.ALWAYS);
ColumnConstraints column2 = new ColumnConstraints(100);
// first column gets any extra width
gridPane.getColumnConstraints().addAll(column1, column2);
```

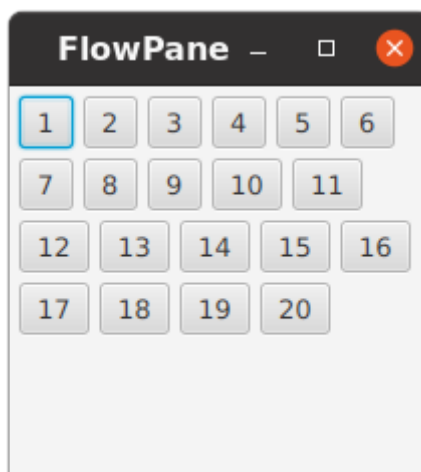
Tamén é posible configurar o tamaño das columnas con unha porcentaxe:

```
GridPane gridpane = new GridPane();
ColumnConstraints column1 = new ColumnConstraints();
column1.setPercentWidth(50);
ColumnConstraints column2 = new ColumnConstraints();
column2.setPercentWidth(50);
// each get 50% of width
gridpane.getColumnConstraints().addAll(column1, column2);
```

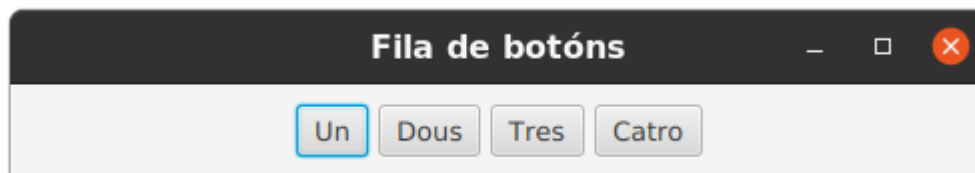
Para crear un layout máis complexo, é posible aniñar diferentes contedores nunha aplicación. Ademais, tamén é posible colocar os compoñentes utilizando coordenadas absolutas.

### Exercicios:

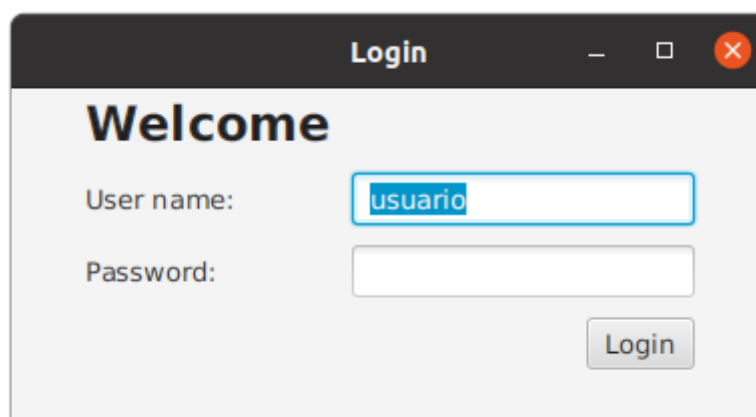
1. Utilizando un FlowPane, crea unha aplicación similar á seguinte imaxe. Nunha mesma fila colocaranse tantos botóns como collan. Se a ventá se redimensiona, os compoñentes recolócanse:



- Utilizando un HBox, crea unha aplicación de forma que os botóns estean sempre centrados horizontalmente, coma na seguinte imaxe:



- Fai un deseño con un AnchorPane que conteña un botón ancorado a unha distancia de 10 do borde inferior e 10 do borde dereito. Proba a redimensionar a ventá e ver o que pasa.
- Deseña unha aplicación JavaFX que simule un formulario de login como o da imaxe:



O formulario está deseñado usando un GridPane de 2 columnas que ten as seguintes propiedades:

- Hgap e Vgap: 10
- Padding: 25, 25, 25, 25
- Ancho: 400
- Alto: 275
- Aliñamento: centro

O texto “Welcome” ten tamaño 24 e está con estilo negra.

O botón está dentro dun contedor HBox que ocupa as 2 columnas e ten unha aliñación á esquina inferior dereita.

Engade unha acción ao botón de tal forma que ao pulsar nel apareza un texto informativo na imaxe (“Pulsaches o botón”). Esta mensaxe debe aparecer centrada.

Para aprender como funciona o GridPane podes mostrar as liñas internas executando a instrución **setGridLinesVisible(true)**.

## Controis

Os controis son os elementos básicos para construír unha aplicación. Un Control é un nodo do grafo de escena que pode ser manipulado polas persoas usuarias.

JavaFX ten un amplo rango de controis predeseñados: etiquetas, checkBox, choiceBox, slider, etc.

- **Botón:** permite executar unha acción cando unha persoa pulsa o botón

```
btn.setOnAction(e -> System.out.println("Ola: " + txt.getText()));
```

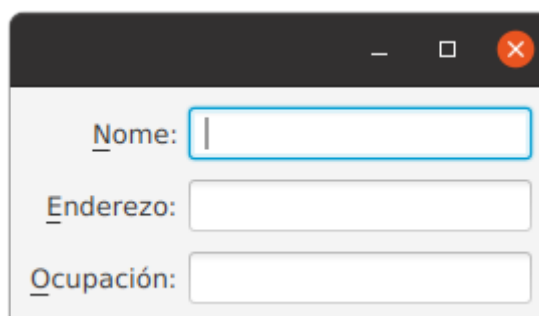
- **Etiqueta:** é un control de texto non editable. Ten unha propiedade **labelFor** que permite especificar o nodo ao que se transferirá o foco cando o mnemonic da etiqueta é pulsado.

```
Label etiqueta = new Label ("Etiqueta");  
etiqueta.setLabelFor(caixaTexto);
```

- **Caixa de texto:** control para permitir á persoa usuaria introducir texto.

```
TextField textField = new TextField();  
textField.setPromptText("Introduce o teu nome");  
System.out.println(textField.getText());
```

**Exercicio:** deseña unha aplicación JavaFX que simule un formulario coma o da seguinte imaxe, feito cun GridPane. Asigna os mnemonics ás etiquetas e fai que ao activar o mnemonic se transfira o foco á caixa de texto asociada.



- **CheckBox:** control con dous estados por defecto (seleccionado ou non seleccionado). Para saber se o checkbox está seleccionado usarase o método [isSelected\(\)](#).

**Exercicio:** deseña unha aplicación JavaFX que teña un checkBox e unha etiqueta que informe do seu estado. Cada vez que se cambia o estado, o texto da etiqueta actualízase. Escolle ti o deseño.

- **ChoiceBox**: control para presentar unha lista de valores predefinidos para escoller.

**Exercicio:** deseña unha aplicación JavaFX que teña un choiceBox e unha etiqueta que informe do valor seleccionado. Cada vez que se selecciona un elemento distinto, o texto da etiqueta actualízase. Escolle ti o deseño.

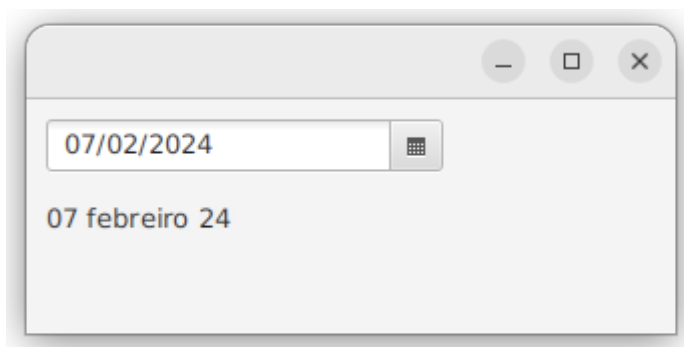
- **Barra de progreso**: control que permite mostrar o progreso dunha tarefa. O progreso establécese como un número de tipo *double* entre 0 e 1.

**Exercicio:** Crea unha aplicación JavaFX que inclúa unha barra de progreso e dous botóns “Aumentar” e “Disminuír”. Fai que ao pulsar os botóns, o valor da barra de progreso aumente ou diminúa, en función do botón pulsado.

Engade á aplicación anterior un control “**ProgressIndicator**”. O seu valor tamén debe cambiar como o da barra de progreso ao pulsar os botón.

- **DatePicker**: control que permite seleccionar unha data.

**Exercicio:** crea unha aplicación JavaFX que conteña un control DatePicker, para seleccionar unha data, e unha etiqueta. Cando se seleccione unha data, esta debe aparecer no texto da etiqueta co formato “día mes ano”, indicando o mes con letras. Por exemplo: “16 febreiro 2022”.



- **ColorPicker**: control para seleccionar unha cor.

**Exercicio:** Crea unha aplicación JavaFX que conteña un control ColorPicker e unha etiqueta. Cada vez que se seleccione unha cor diferente no ColorPicker cambiarase a cor do texto da etiqueta pola cor seleccionada.

- **TabPane**: contedor que pode conter múltiples pestanas. As diferentes pestanas móstranse facendo click no título.

**Exercicio:** Crea unha aplicación JavaFX que conteña un TabPane con dúas pestanas mínimo. Inclúe en cada unha das pestanas diferentes controis.

- **MenuBar**: control composto dunha serie de obxectos Menu, que á súa vez conteñen obxectos MenuItem.

**Exercicio:** Crea unha aplicación JavaFX que inclúa unha barra de menús. Proba a engadir diferentes elementos: submenús, separadores, etc. Decide ti a acción a realizar cando se pulse un menú.



- **Botóns de radio:** os botóns de radio úsanse, normalmente, para crear unha serie de elementos excluíntes entre sí, dos que só pode estar seleccionado un. Para implementar esta funcionalidade en JavaFX, os RadioButtons teñen que pertencer ao mesmo **ToggleGroup**.

**Exercicio:** Crea unha aplicación JavaFX que inclúa 3 botóns de radio dos cales só poida seleccionarse un. Inclúe unha etiqueta na aplicación, de tal forma que o texto da etiqueta mostre sempre información do botón de radio seleccionado actualmente.

- **ToggleButton:** é un botón que pode estar seleccionado ou non seleccionado. Mentres está seleccionado móstrase como se estivese pulsado.

Poden combinarse nun grupo (**ToggleGroup**) dous ou máis botóns de tal forma que só un poida estar seleccionado.

**Exercicio:** Crea unha aplicación JavaFX que conteña 3 TogglesButtons, dos cales só un poida estar seleccionado ao mesmo tempo. Estes botóns fan referencia a 3 cores diferentes, por exemplo vermello, verde e azul. Engade un rectángulo á aplicación e fai que estea pintado da cor do botón seleccionado. Se non hai ningún botón seleccionado o rectángulo debe ter cor branca.

## Eventos

As aplicacións gráficas son aplicacións dirixidas por eventos. É dicir, executarán diferente código en función dos eventos que se produzan. Estes eventos poden ser xerados polas persoas usuarias (pulsar o rato ou o teclado, por exemplo), unha aplicación (un timer) ou o sistema (un reloxo, por exemplo).

Un **evento** é un cambio no estado dun dispositivo (como un rato ou un teclado), unha acción da persoa usuaria (ActionEvent) ou unha tarefa en segundo plano. Os eventos tamén poden ser lanzados como resultado de facer scroll ou ao editar nodos complexos como un TableView ou un ListView.

Nas aplicacións JavaFX existe un thread que escoita os eventos que se producen. Cando se lanza un evento, o thread da interface gráfica comproba se hai algún manexador de eventos rexistrado para ese evento e se é así, execútao.

A clase que representa os eventos é [javafx.event.Event](#). Esta clase ten moitas subclases que representan aos distintos eventos que poden suceder nunha aplicación: MouseEvent, KeyEvent, WindowEvent, etc.

Cada evento ten as seguintes propiedades:

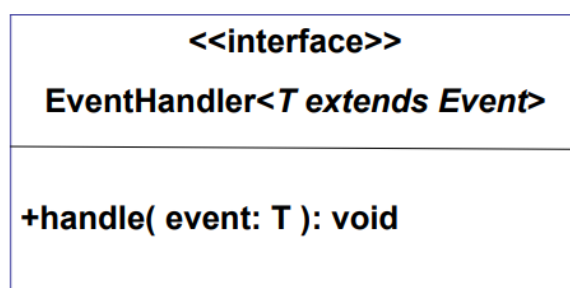
- **Source:** é o obxecto onde se rexistrou o manexador do evento.
- **Target:** é o nodo onde se produciu o evento. Por exemplo un botón, unha ventá, etc.
- **Type:** é o tipo de evento producido. Por exemplo presionar un botón do rato, presionar unha tecla, mover o rato, etc.

O obxecto onde se xerou o evento delega a tarefa de manexo de evento no manexador de eventos. Cando o evento ocorre, o obxecto source crea un obxecto Event e envía ao manexador de evento rexistrado.

Cando se produce un evento, determínase a ruta do evento mediante a **Event Dispatch chain**, que non é máis que o camiño dende o Stage ata o nodo fonte do evento. Unha vez construído o evento, este viaxa dende o nodo raíz ata o nodo fonte do evento (**fase de captura**). Se calquera nodo ten un [filtro rexistrado para o evento](#), executarase. Durante a **fase de burbulla (bubbling phase)** o evento viaxa no sentido inverso, dende o nodo á raíz da árbore. Se calquera nodo ten rexistrado un manexador para o evento, este executarase.

**NOTA:** algúns nodos, como o botón, consumen o evento, polo que non se propaga pola cadea de nodos.

JavaFX ten unha única interface para todos os manexadores de eventos: [EventHandler](#). Observar que está parametrizada co tipo de evento



Para responder a un evento é necesario implementar un [EventHandler](#). Para iso haberá que indicar o [tipo de evento](#) que manexará ([MouseEvent](#) no exemplo). O manexador de eventos pode ser creado usando unha clase anónima ou unha expresión lambda:

```
EventHandler<MouseEvent> handler1 = new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent event) {
        System.out.println("handler1...");
    }
};

EventHandler<MouseEvent> handler2 = (MouseEvent event) -> {
    System.out.println("handler2...");
};
```

Para agregar estes manexadores de eventos pode usarse o método **addEventHandler()**. Como parámetros hai que indicar o tipo de evento que se quere manexar e o manexador de eventos.

```
btn.addEventHandler(MouseEvent.MOUSE_CLICKED, handler1);
```

Algunhas clases en JavaFX definen propiedades para o manexo de eventos, que proporcionan unha forma de rexistrar manexadores de eventos. Os setters destas propiedades permitirán rexistrar manexadores de eventos. A súa sintaxe é:

### **setOnEvent-Type(EventHandler<super event-class>)**

Así, por exemplo, para engadir un evento do rato a un botón, pódese usar o método **setOnMouseClicked** para eventos `MOUSE_CLICKED`. Estes métodos teñen a forma **setOnXXX()** onde XXX corresponde ao tipo de evento para o que se desexa agregar o controlador:

```
btn.setMouseClicked(event -> System.out.println("clic!"));
```

Outros exemplos:

- `setOnAction( EventHandler<ActionEvent> e )`
- `setOnKeyTyped( EventHandler<KeyEvent> e )`
- `setMouseClicked( EventHandler<MouseEvent> e )`
- `setOnMouseMoved( EventHandler<MouseEvent> e )`

### **Exercicios:**

1. Crea unha aplicación JavaFX que conteña un panel e un rectángulo da cor que ti queiras dentro. Rexistra un manexador de eventos para o rectángulo e o panel. Fai que cada vez que se pulse co rato sobre estes elementos apareza por pantalla (serve na consola) información do target, source e type do evento. Mostra tamén a posición X e Y de onde se lanzou o evento. ¿Estas coordenadas son relativas á ventá da aplicación ou ao monitor?
2. Crea unha aplicación Java FX con un panel HBox. Crea un menú contextual con dúas opcións. Fai que ao pulsar co botón dereito do rato sobre o panel apareza o menú contextual (show) na posición actual do rato. Configura as opcións do menú coa acción que queiras.
3. Crea unha aplicación Java FX con 4 botóns e unha etiqueta. Crea un único manexador de eventos que debes asignar aos 4 botóns. O manexador de eventos debe mostrar na etiqueta información de cal foi o botón pulsado.

## **Novo proxecto JavaFX FXML**

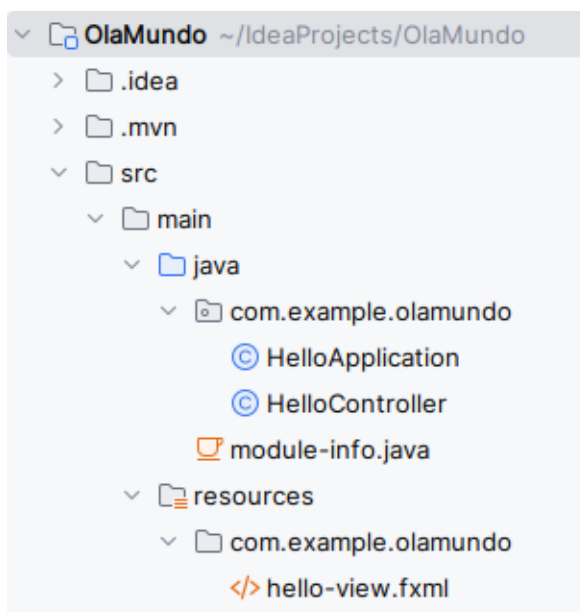
Ata agora creamos aplicacións sinxelas en JavaFX onde todo o código está nun único ficheiro. Son exemplos pequenos, mais cando as aplicacións empezan a medrar é útil ter o código estruturado en diferentes ficheiros para que sexa máis fácil de entender e manexar.

Dado que JavaFX foi deseñado tendo en conta o patrón MVC, é moi fácil crear unha aplicación que siga este patrón (Model View Controller), garantizando que o código da interface (FXML) estea separado do código que manipula os datos da aplicación. O controlador será un intermediario entre a interface e os datos, é dicir, é a clase encargada de xestionar os eventos que suceden na aplicación.

En JavaFX a estrutura xerárquica dos compoñentes da interface pode definirse usando a linguaxe XML. O formato XML específico de JavaFX chámase **FXML**.

No ficheiro FXML dunha aplicación JavaFX defínense todos os compoñentes gráficos e as súas propiedades e enlázanse con unha clase Controladora, que será a responsable de manexar a lóxica de control do programa.

Cando dende IntelliJ se crea un novo proxecto baseado en JavaFX, créase unha aplicación base que utiliza FXML. Observar a estrutura de arquivos e carpetas creadas:



O ficheiro **HelloApplication.java** é a clase principal da aplicación e a que inicia o programa:

```
public class HelloApplication extends Application {
    public static void main(String[] args) {
        launch();
    }

    @Override
    public void start(Stage stage) throws IOException {
        FXMLLoader fxmlLoader =
            new FXMLLoader(HelloApplication.class.getResource("hello-view.fxml"));
        Scene scene = new Scene(fxmlLoader.load(), 320, 240);
        stage.setTitle("Hello!");
        stage.setScene(scene);
        stage.show();
    }
}
```

Fixarse como a clase anterior carga o ficheiro **.fxml** utilizando o método **FXMLLoader.load()**. Este método crea todos os obxectos definidos en dito ficheiro e devolve o grafo de escena.

O ficheiro **fxml** ten un VBox como nodo de nivel superior, que será o nodo raíz do grafo de escena. Cando se constrúe a escena (`new Scene(...)`) pásase o nodo raíz do grafo de escena, que é o devolto polo método **FXMLLoader.load()**.

E finalmente, para mostrar a escena, haberá que invocar o método **stage.show()**.

Contido do ficheiro fxml:

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

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>

<?import javafx.scene.control.Button?>
<VBox alignment="CENTER" spacing="20.0" xmlns:fx="http://javafx.com/fxml"
    fx:controller="com.example.olamundo.HelloController">
    <padding>
        <Insets bottom="20.0" left="20.0" right="20.0" top="20.0"/>
    </padding>

    <Label fx:id="welcomeText"/>
    <Button text="Hello!" onAction="#onHelloButtonClick"/>
</VBox>
```

A primeira liña é a declaración de XML:

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

Despois veñen unha serie de instrucións **import**. Igual que en Java, antes de usar un compoñente, hai que importalo.

A continuación veñen os compoñentes da interface coas súas propiedades. Fixarse que:

- O atributo **xmlns:fx** é necesario e define o espacio de nomes **fx**.
- A clase controladora está especificada coa seguinte instrución:

```
fx:controller="com.example.olamundo.HelloController"
```

- O atributo **fx:id** nun elemento crea unha variable no espacio de nomes do documento, á que se poderá facer referencia dende outros ficheiros do código. No caso da etiqueta correspóndese co nome da variable na clase **HelloController**
- O método a executar cando se pulsa o botón está especificado no atributo **onAction**:

```
onAction="#onHelloButtonClick"
```

O método a executar é **onHelloButtonClick**, que estará definido na clase **HelloController**.

**NOTA:** cando se cree un novo ficheiro \*.fxml IntelliJ proporciona axuda automaticamente para crear a clase controladora.

Observar que o botón ten un atributo “text” tal que o seu valor será asignado a unha propiedade definida na clase do botón. É dicir, cando o FXMLLoader cree o botón, establecerá o valor do atributo de texto. Noutras palabras, é como se o FXMLLoader invocase o seguinte código:

```
Button boton = new Button();
button.setText("Hello!");
```

Cabe destacar que o tipo de node debe definir as propiedades que conincidan coa convención dos JavaBeans, é dicir, deben ter os métodos get e set. Se falta algún, a propiedade volverase invisible para o FXMLLoader.

Código da clase controladora:

```
public class HelloController {
    @FXML
    private Label welcomeText;

    @FXML
    protected void onHelloButtonClick() {
        welcomeText.setText("Welcome to JavaFX Application!");
    }
}
```

Para asociar os elementos da clase controladora cos usados en FXML é necesario engadir a **anotación @FXML** antes da súa declaración. Isto permite que un membro privado ou protexido dunha clase sexa accesible dende FXML.

A clase controladora do exemplo ten declarado o método **onHelloButtonClick**, que se enlaza dende o FXML para indicar o método a executar cando se pulsa o botón.

Tamén sería interesante que se declarase o botón e se asociase co elemento da interface usando a **anotación @FXML**. O nome da variable do botón e o identificador usado en FXML teñen que ser idénticos.

No seguinte exemplo utilízanse propiedades de GridPane nos nodos que se engaden ao panel, para o que se fai uso do espacio de nomes (GridPane.). Estas propiedades están definidas na clase GridPane e non nos nodos de destino. Esta é a razón pola que o nome da clase aparece como espacio de nomes.

```
<GridPane>
  <Label text="Username:" GridPane.columnIndex="0" GridPane.rowIndex="0"/>
  <TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
  <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>
  <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="1"/>
  <Button text="Cancel" GridPane.columnIndex="0" GridPane.rowIndex="2"/>
  <Button text="Login" GridPane.columnIndex="1" GridPane.rowIndex="2"/>
</GridPane>
```

Cada propiedade debe ter os seus métodos get e set. Neste caso ademais, deben ter o modificador estático. Isto dá como resultado as seguintes definicións de métodos en GridPane:

```
public static void setColumnIndex(Node node, int index);
public static int getColumnIndex(Node node);
public static void setRowIndex(Node node, int index);
public static int getRowIndex(Node node);
```

Os nodos poden definir unha propiedade por defecto, para o que teñen que engadir a anotación **@javafx.beans.DefaultProperty** na clase do compoñente destino. Por exemplo, o tipo **javafx.scene.layout.Pane** ten a anotación **@DefaultProperty("children")** o que permite escribir o código anterior en lugar de ter que escribir o seguinte:

```
<GridPane>
  <children>
    <Label text="Username:" GridPane.columnIndex="0" GridPane.rowIndex="0"/>
    <!-- elementos adicionales -->
  </children>
</GridPane>
```

Se é necesario inicializar algo na interface, pode engadirse o método **initialize()** ao controlador coas instrucións a executar cada vez que se cargue a interface gráfica durante a execución da aplicación. Exemplo:

```
<VBox xmlns:fx="http://javafx.com/fxml" alignment="CENTER" spacing="20.0"
  fx:controller="com.example.olamundo.HelloController">

  <Label fx:id="welcomeText"/>
  <Button text="Hello!" onAction="#onHelloButtonClick"/>
  <Button fx:id="button" text="Click Me!"/>
</VBox>
```

```
public class HelloController {
  @FXML
  private Button button;
  ...

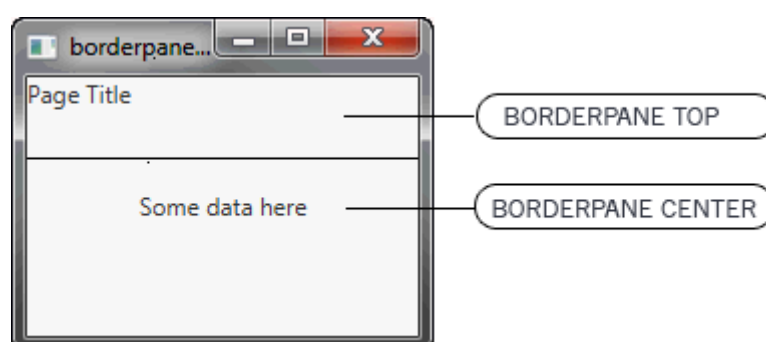
  public void initialize() {
    button.setOnAction(new EventHandler<ActionEvent>() {
      @Override
      public void handle(ActionEvent event) {
        System.out.println("You clicked me!");
      }
    });
  }
}
```

## JavaFX FXML

O formato JavaFX FXML é un formato baseado en XML para realizar deseños de interfaces gráficas de forma similar a como se crean interfaces gráficas en HTML. Ademais, FXML permite separar o deseño da interface do resto do código.

FXML non ten un esquema, mais ten unha estrutura predefinida. A forma de construír o FXML depende da API dos obxectos construídos, polo que se pode consultar a documentación de dita API para entender que elementos e atributos se poden engadir. En xeral, a maioría das clases JavaFX poden ser usadas como elementos e a maioría das súas propiedades poden ser usadas como atributos (tamén se inclúen as propiedades herdadas).

O seguinte código mostra un exemplo de creación dunha interface gráfica en Java en en FXML. A interface deseñada correspóndese coa da seguinte imaxe:



```
BorderPane border = new BorderPane();
Label toppanetext = new Label("Page Title");
border.setTop(toppanetext);
Label centerpanetext = new Label ("Some data here");
border.setCenter(centerpanetext);
```

```
<BorderPane>
  <top>
    <Label text="Page Title"/>
  </top>
  <center>
    <Label text="Some data here"/>
  </center>
</BorderPane>
```

## Obxectos en FXML

Para crear obxectos en FXML úsase o elemento FXML apropiado. O nome do elemento usado en FXML correspóndese co nome da clase Java.

Hai que ter en conta que para poder usar un elemento, primeiro hai que importalo, igual que no código Java.



No seguinte exemplo, os nomes VBox e Label son válidos porque as dúas clases están declaradas usando as sentencias import.

A maioría dos obxectos de JavaFX teñen **propiedades**. Exemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox spacing="20">
  <children>
    <Label text="Line 1"/>
    <Label text="Line 2"/>
  </children>
</VBox>
```

Hai varias formas de establecer os valores dunha propiedade dos obxectos Java:

- Usar un atributo XML. Por exemplo, o valor establecido ao atributo **spacing** (20) pasarase como parámetro ao método **setSpacing()** do obxecto VBox cando este sexa creado. Tamén o valor do atributo **text** das etiquetas, será pasado como parámetro ao método **setText()** da etiqueta.
- Usar un elemento aniñado. O elemento **children** aniñado no VBox correspóndese co método **getChildren** do obxecto VBox. Os elementos aniñados no seu interior serán convertidos en compoñentes JavaFX e engadidos á colección de descendentes do obxecto VBox obtidos con **getChildren()**.

Observar que as propiedades son accesibles mediante os métodos **get/set**.

En JavaFX é posible asignar IDs a elementos. Estes IDs poden ser usados para referenciar elementos noutra parte do ficheiro FXML, dende CSS ou dende a clase controladora.

Os IDs asígnanse usando o atributo **id** do espazo de nomes de FXML:

```
<VBox xmlns:fx="http://javafx.com/fxml/1">
  <Label fx:id="label1" text="Line 1"/>
</VBox>
```

## Espazo de nomes

FXML ten un espazo de nomes que pode establecerse no elemento raíz. Este espazo de nomes é necesario para algúns atributos como o atributo **fx:id**.

Exemplo de declaración do espazo de nomes:

```
<VBox xmlns:fx="http://javafx.com/fxml">
```

## Clase controladora FXML

Os documentos FXML teñen asociada unha clase controladora. Unha clase controladora permite acceder aos compoñentes declarados no ficheiro FXML e facer que o obxecto controlador actúe de mediador.

Hai varias formas de establecer un controladora para un ficheiro FXML. Unha forma é especificalo no ficheiro.

Exemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Button?>

<VBox xmlns:fx="http://javafx.com/fxml"
fx:controller="com.jenkov.javaafx.MyFXMLController" >
  <Button text="Click me!" onAction="reactToClick()"/>
</VBox>
```

Cando se carga o ficheiro FXML créase unha instancia da clase controladora. Para que funcione, a clase controladora debe ter un construtor sen argumentos.

Poden asociarse compoñentes no ficheiro FXML con campos da clase controladora. Para que sexa posible, hai que engadir o atributo **fx:id** ao elemento FXML co mesmo nome que o campo da clase controladora e na clase controladora engadir a anotación **@FXML**. Exemplo:

```
public class MyFXMLController {

    @FXML
    public Label label1 = null;

}

<VBox xmlns:fx="http://javafx.com/fxml/1">
  <Label fx:id="label1" text="Line 1"/>
</VBox>
```

Tamén é posible referenciar métodos da clase controladora dende FXML. Exemplo:

```
<VBox xmlns:fx="http://javafx.com/fxml"
fx:controller="com.jenkov.javaafx.MyFXMLController" spacing="20">
<children>
  <Label fx:id="label1" text="Line 1"/>
  <Label fx:id="label2" text="Line 2"/>
  <Button fx:id="button1" text="Click me!" onAction="#buttonClicked"/>
</children>
</VBox>
```

```
public class MyFXMLController {

    @FXML
    public void buttonClicked(Event e){
        System.out.println("Button clicked");
    }
}
```

## FXML estilos CSS

É posible aplicar estilos a compoñentes JavaFX no ficheiro FXML utilizando o elemento `<style>`.

As propiedades usadas en JavaFX son moi similares ás da web con CSS, aínda que comezan por `-fx-`. Exemplos:

- `-fx-background-color`: cor de fondo.
- `-fx-text-fill`: cor do texto.
- `-fx-font-size`: tamaño do texto.

Máis información na [guía de referencia de CSS](#).

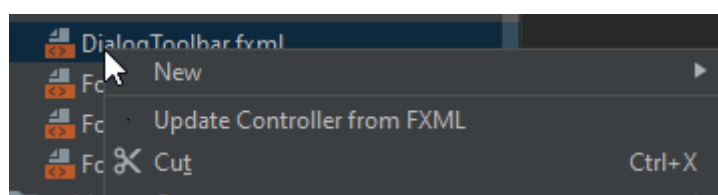
Exemplo:

```
<VBox xmlns:fx="http://javafx.com/fxml/1">
  <Button text="Click me!" onAction="reactToClick()">
    <style>
      -fx-padding: 10;
      -fx-border-width: 3;
    </style>
  </Button>
</VBox>
```

## FXML e IntelliJ

Para poder actualizar automaticamente a clase controladora cos cambios introducidos no ficheiro fxml (creación automática de ids e de métodos), haberá que instalar o plugin [FXMLManager](#).

Con este plugin pode pulsarse co botón dereito no ficheiro fxml e actualizar a clase controladora:

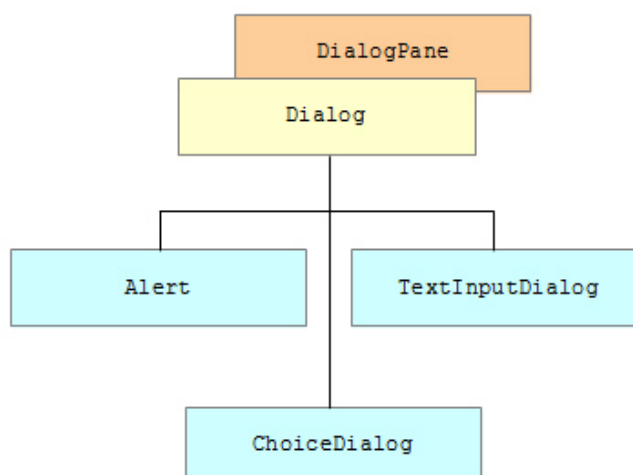


**Exercicio:**

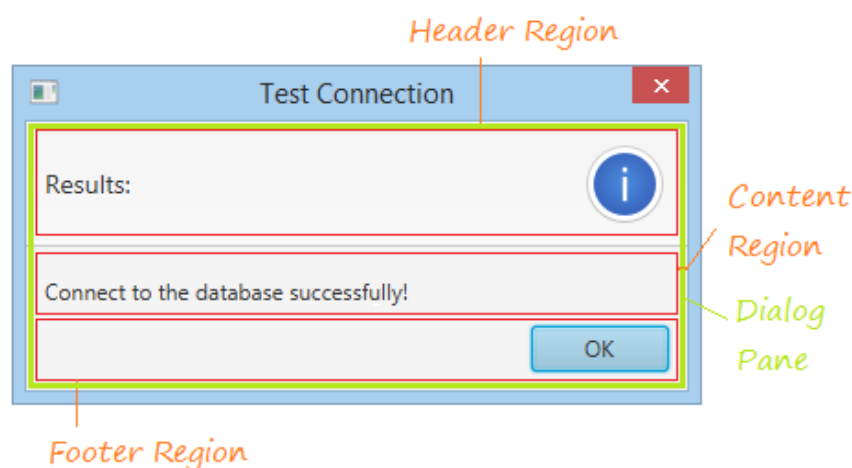
1. Utilizando **JavaFX FXML** repite algúns dos exercicios creados ata agora. Proba a utilizar diferentes layouts e diferentes controis.
2. Crea unha aplicación que conteña unha área de texto e un menú contextual con dúas opcións. A primeira opción permitirá incrementar o tamaño da letra da área de texto e a segunda opción permitirá pasar a maiúsculas o texto que estea seleccionado da área de texto.

## Diálogos

En JavaFX [Dialog](#) é unha clase base para crear diálogos. Ten tres subclases especializadas: [Alert](#), [ChoiceDialog](#) and [TextInputDialog](#), que son as que se usan habitualmente.



Un Dialog está formado por un [DialogPane](#), nodo raíz do diálogo, e ofrece a funcionalidade necesaria para mostralo no Stage. O DialogPane é o responsable da colocación da cabeceira, gráficos, contido e botóns. A seguinte imaxe mostra as diferentes rexións do panel, podendo configurar cales se mostran/ocultan.



A clase [Dialog<R>](#) ten un tipo xenérico, R, que representa o tipo do resultado devolto polo diálogo a través da propiedade [result](#). Por defecto o tipo do resultado é [ButtonType](#).

Despois de instanciar un diálogo hai que configuralo. As únicas propiedades relacionadas con contido son [contentText](#), [headerText](#) e [graphic](#), que se corresponden coas propiedades equivalentes do `DialogPane`. Dado que o habitual é mostrar unha cadea de texto, tanto na cabeceira como no contido do diálogo, estas propiedades son moi útiles xa que se lles pode asignar directamente un string sen necesidade de crear unha etiqueta.

Tamén sería posible establecer un nodo na área de contido ou cabeceira. Hai que ter en conta que cando nunha rexión se establece ao mesmo tempo un nodo e un texto, terá prioridade o nodo.

A continuación haberá que establecer os botóns creados usando [ButtonType](#). Exemplo de diálogo:

```
Dialog<String> dialog = new Dialog<>();
dialog.setTitle("Login Dialog");
dialog.setContentText("Would you like to log in?");
ButtonType loginButtonType = new ButtonType("Login", ButtonData.OK_DONE);
dialog.getDialogPane().getButtonTypes().add(loginButtonType);
dialog.showAndWait();
```

Unha vez que o diálogo está configurado, hai que mostralo. A maioría das veces os diálogos son mostrados de forma “modal” e bloquean o código. “Modal” significa que o diálogo evita calquera interacción da persoa usuaria coa aplicación mentres o diálogo é visible. O bloqueo do código significa que a execución da aplicación se detén no punto onde se mostra o diálogo. Isto quere dicir que se mostra o diálogo, espérase a resposta da persoa usuaria e despois continúaase a execución da aplicación no punto onde parara, permitindo á persoa desenvolvedora utilizar inmediatamente a resposta ao diálogo. Un diálogo que non bloquee o código implicaría escribir o código necesario para asegurarse de obter a resposta antes de que esta sexa usada. Para especificar se se quere o comportamento que bloquee o código ou non, utilízanse respectivamente os métodos [showAndWait\(\)](#) or [show\(\)](#).

Os diálogos de JavaFX son modais por defecto. É recomendable establecer a ventá antecesora a través do método [initOwner\(Window owner\)](#).

## Alert

A clase [Alert](#) é subclase de [Dialog](#) e proporciona soporte para un número de diálogos predeseñados para solicitar información nunha interface gráfica. Moitas persoas consideran que a clase `Alert` é a apropiada na maioría dos casos. Tamén están dispoñibles as clases [TextInputDialog](#) and [ChoiceDialog](#) para facer unha pregunta tipo texto ou de escolla, respectivamente.

Cando se crea unha nova instancia da clase **Alert**, hai que pasar como parámetro un valor de entre os dispoñibles en [Alert.AlertType](#) (CONFIRMATION, ERROR, INFORMATION,

NONE ou WARNING). A partir deste valor configúranse apropiadamente a ventá incluíndo o título, cabeceira, gráfico, así como os botóns por defecto.

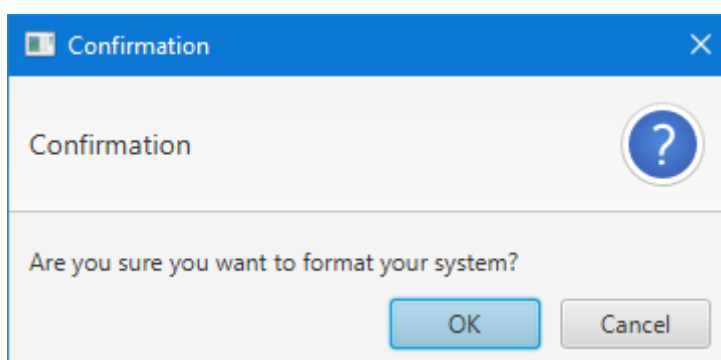
Igual que un diálogo, unha alerta está composta por:

- A **cabeceira**, que soe usarse para mostrar unha pequena notificación e unha icona.
- Na **rexión de contido** normalmente amósase un texto utilizando o método [setContentText\(String\)](#). Tamén é posible mostrar un Nodo neste rexión a través do método [alert.getDialogPane\(\).setContent\(Node\)](#).
- Na **rexión ao pé** da ventá é onde soen mostrarse os botóns, podendo personalizalos.

Unha vez que se crea un obxecto **alert**, hai que mostralo. Igual que os diálogos, as alertas son mostradas de forma modal e bloquean a execución do código. Isto significa que as persoas usuarias non poden interactuar con ningunha outra ventá da aplicación e o código detén a súa execución esperando a resposta das persoas usuarias. Para especificar se o código se bloquea ou non esperando unha resposta hai que escoller entre usar os métodos [showAndWait\(\)](#) or [show\(\)](#), respectivamente.

Exemplo de creación dunha alerta:

```
Alert alert = new Alert(AlertType.CONFIRMATION, "Are you sure you want to format your system?");
Optional<ButtonType> result = alert.showAndWait();
if (result.isPresent() && result.get() == ButtonType.OK) {
    formatSystem();
}
```



Exercicios:

1. Crea unha aplicación con varios botóns, cada un dos cales debe crear un diálogo distinto:
  - a. Un ChoiceDialog que permita escoller entre varias opcións e mostre por consola a opción seleccionada ao cerrar o diálogo.
  - b. Unha alerta de tipo ERROR. Personaliza o título, a cabeceira (header) e o contido.
  - c. Unha alerta de tipo INFORMATION.

- d. Unha alerta de tipo WARNING.
- e. Un `TextInputDialog` que pida un nome que se mostrará por consola ao pechar o diálogo.
- f. Proba a personalizar a icona dun diálogo.

## Propiedades

Unha propiedade JavaFX é un tipo especial de variable dun obxecto JavaFX. As propiedades JavaFX son usadas para almacenar información do obxecto. Ademais, as propiedades teñen a capacidade de notificar cambios nos seus valores e tamén permiten facer asociacións con propiedades doutros obxectos.

Por exemplo, [consulta na API as propiedades do obxecto Slider](#). Entre estas variables están as tres fundamentais: **max**, **min** e **value**, que almacena o valor actual representado polo Slider.

Os obxectos tamén teñen propiedades herdadas dos seus antecesoros. Por exemplo, consulta na [API de TextField](#) que ten unha propiedade **text** herdada de **TextInputControl**.

Pode engadirse un listener a unha propiedade JavaFX para detectar cambios na mesma. No caso de que a propiedade cambie, executarase o manexador do evento definido no listener.

Por exemplo, nun slider unha propiedade á que se pode asociar un listener para detectar cambios é a propiedade **value**, que representa o valor actual seleccionado no slider. Pódese asociar un listener a esta propiedade co seguinte código:

```
slider.valueProperty().addListener(...);
```

O exemplo anterior, executará o código programado cada vez que o valor do slider cambie.

Tamén é posible asociar (bind) propiedades entre sí, de tal forma que cando se produza un cambio nunha propiedade, automaticamente se cambie a propiedade asociada. É dicir, cando dúas variables están asociadas, un cambio nunha delas reflectirase na outra.

Exemplo de asociación de dúas variables de tal forma que un cambio nunha produza automaticamente un cambio na outra:

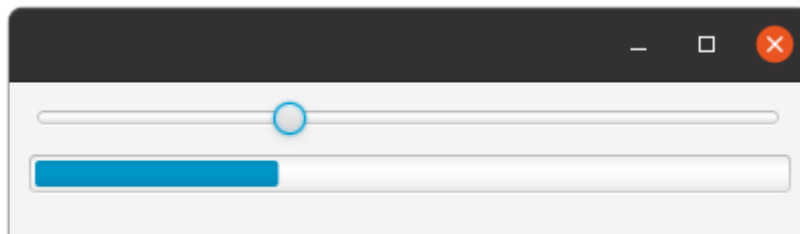
```
Slider slider = new Slider(0, 1, 0);

ProgressBar bar = new ProgressBar(0);
bar.setMaxWidth(Double.MAX_VALUE);
bar.progressProperty().bind(slider.valueProperty());
```

O exemplo anterior é unidireccional. É dicir, un cambio no slider producirá un cambio na barra de progreso e non ao revés. Se quixeramos que a asociación fose bidireccional usárase o método **bindBidirectional**.

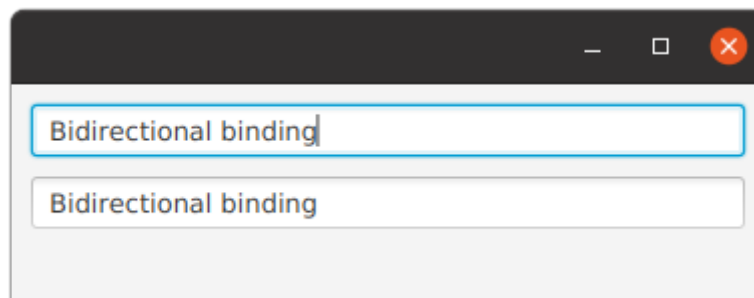
**Exercicios:**

1. Crea unha aplicación JavaFX que conteña un Slider e unha barra de progreso. Asocia a propiedade do Slider coa propiedade progress da barra de progreso de tal forma que cando o slider cambie tamén o faga a barra de progreso. Debe verse algo similar á seguinte imaxe:



Engade ao exercicio anterior dous botóns: un para aumentar e outro para diminuir o progreso da barra de progreso. Fai as modificacións necesarias para que a asociación entre o slider e a barra de progreso sexa bidireccional.

2. Crea unha aplicación JavaFX que conteña dúas caixas de texto. O contido das caixas de texto debe estar asociado, de tal forma que o que se escriba nunha aparecerá automaticamente na outra e viceversa.

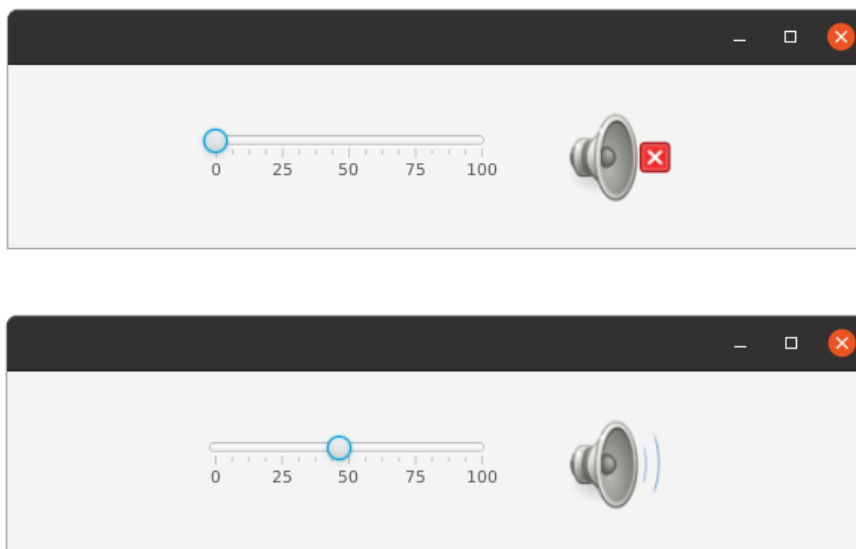


3. Crea unha aplicación JavaFX que inclúa un Slider e un ImageView, de tal forma que ao modificar o valor do Slider, a imaxe que se amosa cambie. O slider simula o volume, de tal forma que a medida que este cambia, a imaxe tamén debe variar.

Inicialmente amósase a icona de silencio e o valor de slider está a 0.



Utilizando un listener, fai que cando o volume suba se amose unha icona apropiada: volume baixo, medio ou alto.



**NOTA:** recoméndase colocar as imaxes en `src/main/resources/imagenes`, así a ruta de acceso dende código será `/imagenes/ficheiroImaxe`.

## Propiedades e Clases

É común en JavaFX usar **Properties** para todos os campos dunha clase do modelo de datos. Unha **Property** permite, ser notificado automaticamente cando unha variable cambia. Isto axuda a ter a interface sincronizada co modelo de datos.

Una property é, en realidade, unha clase envoltorio que engade funcionalidades extra aos campos dunha clase. Estas funcionalidades extra permiten notificar de cambios nos seus valores e facer asociacións entre propiedades.

Para cada tipo simple como `String`, `Double`, etc. hai a clase envoltorio como `StringProperty`, `DoubleProperty`, etc. Para tipos complexos existe `ObjectProperty<T>`.

Para crear as propiedades hai implementacións distintas para cada tipo de datos. Por exemplo, para unha propiedade de tipo `Double` contamos con `SimpleDoubleProperty` e `ReadOnlyDoubleWrapper`. A primeira crea unha propiedade de lectura/escritura e a segunda de só lectura.

[Máis información sobre as propiedades.](#)

Exemplo de clase con unha propiedade:

```
class Bill {
    // Define a variable to store the property
    private DoubleProperty amountDue = new SimpleDoubleProperty();

    // Define a getter for the property's value
    public final double getAmountDue() {return amountDue.get();}

    // Define a setter for the property's value
    public final void setAmountDue(double value) {amountDue.set(value);}

    // Define a getter for the property itself
    public DoubleProperty amountDueProperty() {return amountDue;}
}
```

Observar que as propiedades teñen os métodos `get()` e `set()` para ler e modificar o tipo primitivo, no caso anterior `double`.

A propiedade **amountDue** non ten un tipo estándar, senón que é unha clase (`DoubleProperty`) que encapsula un tipo primitivo e engade nova funcionalidade. Todas as clases do paquete [javafx.beans.property](#) conteñen soporte para observabilidade e asociación (observability and binding).

Exemplo de asociación dun listener á propiedade **amountDue**:

```
public class Main {

    public static void main(String[] args) {
        Bill electricBill = new Bill();

        electricBill.amountDueProperty().addListener(new ChangeListener() {
            @Override public void changed(ObservableValue o, Object oldVal,
                Object newVal) {
                System.out.println("Electric bill has changed!");
            }
        });

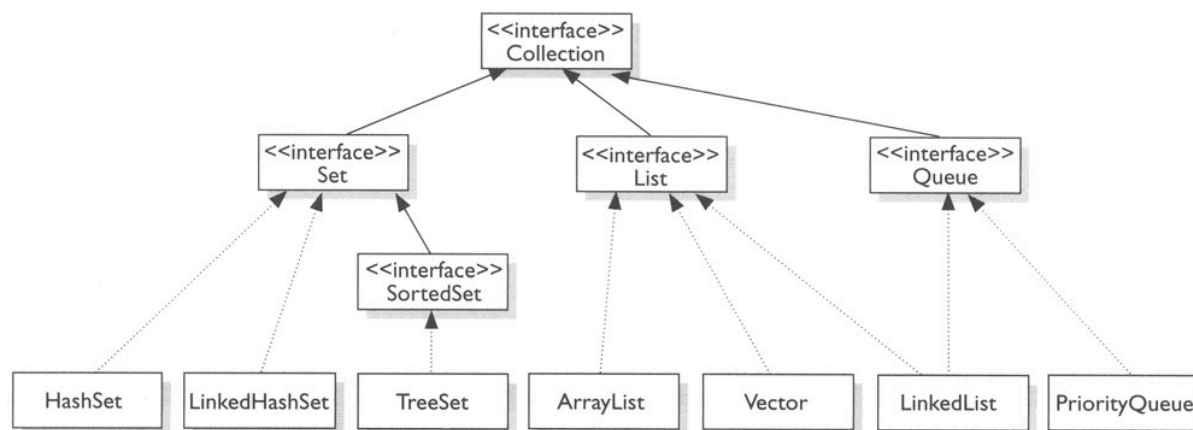
        electricBill.setAmountDue(100.00);
    }
}
```

O listener anterior tamén pode implementarse como unha expresión lambda:

```
electricBill.amountDueProperty().addListener((ov, oldVal, newVal) -> {
    System.out.println("Electric bill has changed!");
});
```

# JavaFX Collections

A interface [java.util.Collection](#) é a interface raíz da xerarquía de Collections. Serve de base para representar un grupo de obxectos. A interface [java.util.List](#) é unha das súas subinterfaces.



A clase [java.util.Collections](#) ten exclusivamente métodos estáticos de utilidade para traballar con coleccións, como por exemplo **reverse**, **sort**, **swap**, etc.

As coleccións en JavaFX están definidas no paquete [javafx.collections](#), composto, entre outras, polas seguintes interfaces e clases:

- Interface [ObservableList](#): lista que permite aos listeners detectar os cambios que se producen.
- Interface [ListChangeListener](#): interface que recibe notificacións de cambios nun ObservableList.
- Clase [FXCollections](#): clase utilidade con métodos estáticos que son copias dos métodos de [java.util.Collections](#).
- Clase [ListChangeListener.Change](#): representa un cambio feito nun [ObservableList](#).

O seguinte exemplo crea un ArrayList que posteriormente se envolve nun ObservableList. Despois rexístrase un **ListChangeListener** que recibirá notificacións cada vez que se fai un cambio no ObservableList:

```

// Use Java Collections to create the List.
List<String> list = new ArrayList<String>();

// Now add observability by wrapping it with ObservableList.
ObservableList<String> observableList = FXCollections.observableList(list);
observableList.addListener(new ListChangeListener() {
    @Override
    public void onChanged(ListChangeListener.Change change) {
        System.out.println("Detected a change! ");
    }
});
  
```

```
// Changes to the observableList WILL be reported.
// This line will print out "Detected a change!"
observableList.add("item one");

// Changes to the underlying list will NOT be reported
// Nothing will be printed as a result of the next line.
list.add("item two");

System.out.println("Size: " + observableList.size());
```

## TableView

O control TableView permite mostrar datos en forma de táboa nunha aplicación JavaFX.

Un TableView mostrará datos dun obxecto. Neste caso utilizarase a clase Persoa seguinte:

```
public class Person {
    private StringProperty firstName;
    private StringProperty lastName;

    public Person(String firstName, String lastName) {
        this.firstName = new SimpleStringProperty(firstName);
        this.lastName = new SimpleStringProperty(lastName);
    }

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

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

    public StringProperty firstNameProperty() {
        return firstName;
    }

    public String getLastName() {
        return lastName.get();
    }

    public void setLastName(String lastName) {
        this.lastName.set(lastName);
    }

    public StringProperty lastNameProperty() {
        return lastName;
    }
}
```

Para mostrar nunha táboa os datos de varias Persoas, haberá que crear un TableView similar ao seguinte código. [Exemplo de uso dun TableView](#):

```
TableView tableView = new TableView();

TableColumn<Person, String> column1 = new TableColumn<>("First Name");
column1.setCellValueFactory(new PropertyValueFactory<>("firstName"));

TableColumn<Person, String> column2 = new TableColumn<>("Last Name");
column2.setCellValueFactory(new PropertyValueFactory<>("lastName"));

tableView.getColumns().add(column1);
tableView.getColumns().add(column2);
```

**NOTA:** o código anterior está creado con Java. Tamén é posible crear o obxecto TableView e as columnas de forma equivalente usando FXML.

Unha táboa necesita unha ou máis columnas. O valor pasado como parámetro ao construtor de TableColumn é a cabeceira que aparecerá na columna da táboa.

```
TableColumn<Person, String> column1 = new TableColumn<>("First Name");
```

A clase TableColumn necesita ter asignado un **cell value factory**. O cell value factory extraerá o valor a ser mostrado en cada cela de cada fila. No exemplo seguinte úsase un PropertyValueFactory, que pode extraer o valor dunha propiedade (pasada como parámetro) dun obxecto java.

```
column1.setCellValueFactory(new PropertyValueFactory<>("firstName"));
```

PropertyValueFactory utiliza o argumento "firstName" no construtor asumindo que o obxecto **Person** ten un método público **firstNameProperty**, sen parámetros. Se dito método existe, é invocado e o valor devolto será colocado na cela da táboa. Ademais, **TableView** engade un observador ao valor devolto tal que, calquera cambio fará que a táboa o detecte e se actualice.

A forma recomendada de engadir elementos ao control TableView é usar un **ObservableList**. Este **ObservableList** é observado por defecto polo TableView, polo que calquera cambio que se produza no observable provocará que a táboa se actualice automaticamente.

No exemplo que estamos usando utilizarase o modelo de datos da seguinte lista:

```
final ObservableList<Person> data = FXCollections.observableArrayList(
    new Person("Jacob", "Smith"),
    new Person("Isabella", "Johnson"),
    new Person("Ethan", "Williams"),
    new Person("Emma", "Jones"),
    new Person("Michael", "Brown")
);
```

Para asignar os datos ao TableView utilizarase a instrución:

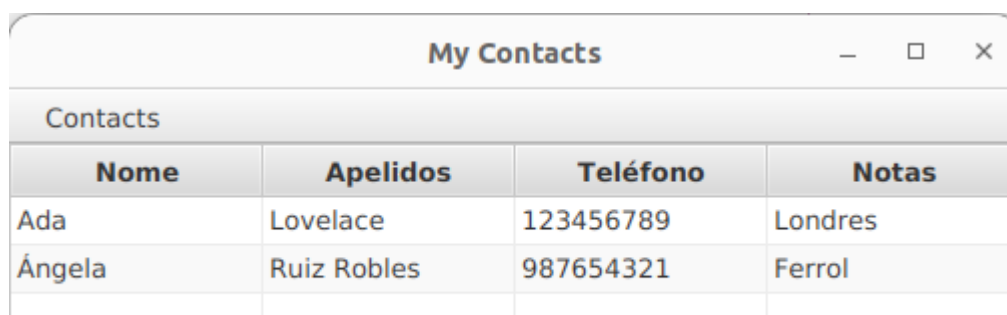
```
tableView.setItems(data);
```

Pode comprobarse que se se actualiza a lista (engadir ou eliminar elementos, por exemplo) o TableView actualízase automaticamente.

### Exercicio:

1. Crea unha aplicación de xestión de contactos que permita engadir, editar e eliminar contactos coas seguintes características:
  - a. a ventá principal debe mostrar todos os contactos usando un control TableView.
  - b. A aplicación terá un menú para engadir, editar e borrar contactos.
  - c. Os contactos deben almacenarse e lerse de ficheiro.
  - d. Crea un obxecto para almacenar os contactos coa seguinte información: nome, apelidos, teléfono e notas.
  - e. Para enlazar os datos dos contactos co TableView, fai que as propiedades do obxecto anterior sexan de tipo SimpleStringProperty.

A aplicación terá un aspecto coma o da seguinte imaxe:



Contacts			
Nome	Apelidos	Teléfono	Notas
Ada	Lovelace	123456789	Londres
Ángela	Ruiz Robles	987654321	Ferrol

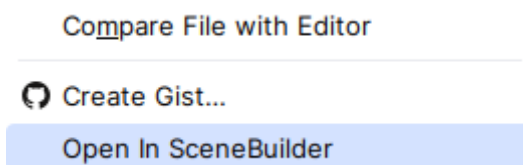
## Scene Builder

A aplicación Scene Builder permite deseñar de forma visual, a interface gráfica dunha aplicación JavaFX. É dicir, ofrece unha interface WYSIWYG para deseñar aplicacións JavaFX.

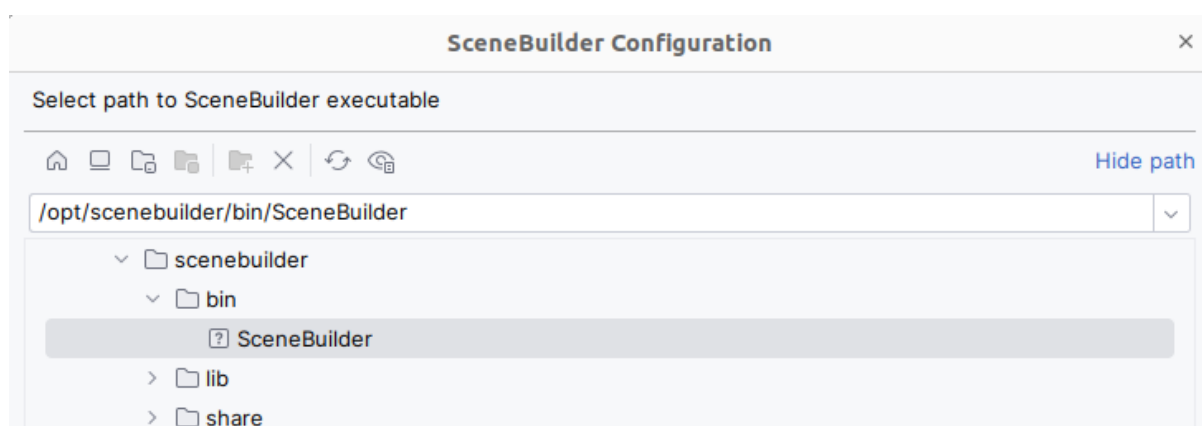
Para usar [Scene Builder](#) hai que descargalo e instalalo no sistema.

Scene Builder permite deseñar de forma visual as interfaces, almacenando a configuración da mesma en ficheiros con extensión **fxml**.

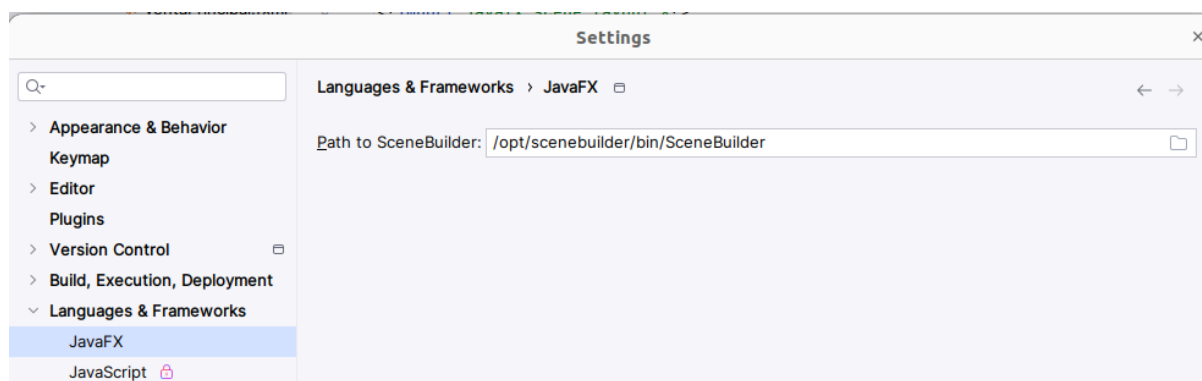
Os ficheiros FXML imos modificalos a través do Scene Builder. Para iso pulsamos co botón dereito -> Open in SceneBuilder:



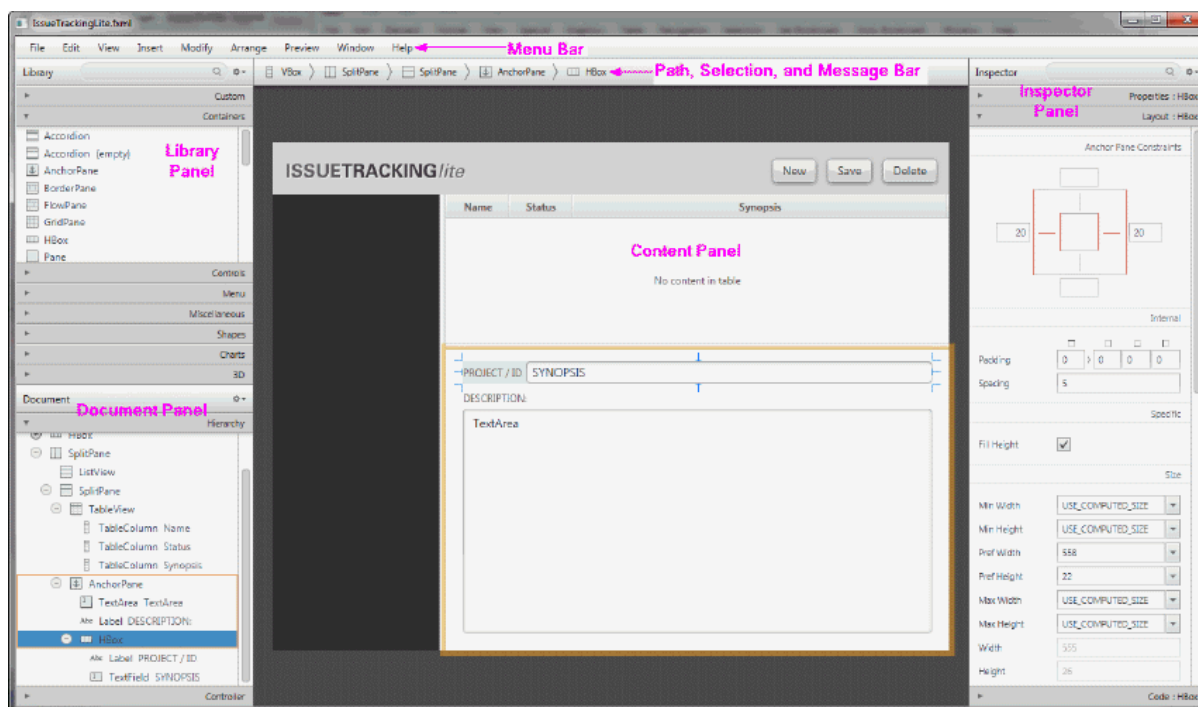
Ao pulsar o botón anterior pode ser necesario configurar no editor a ruta do directorio de instalación do Scene Builder:



A ruta de SceneBuilder tamén pode ser configurada dende o menú Plugins que aparece ao pulsar sobre a roda dentada na esquina superior dereita do editor:



Cando se abra Scene Builder, aparecerá unha ventá similar á seguinte:



Seccións incluídas por defecto na ventá de Scene Builder:

- **Manu Bar:** barra de menús.

Son interesantes algunhas das opcións que ofrece o menú:

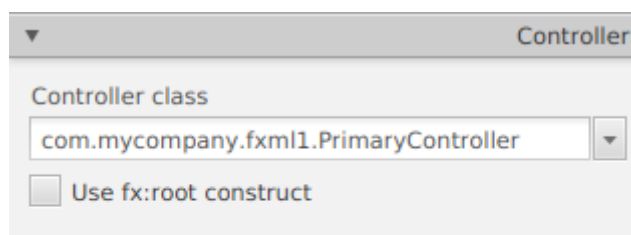
- [Explicación detallada das opcións do menú Modify.](#)
- [Explicación detallada das opcións do menú Preview.](#)
- **Path, Selection and Message Bar:** mostra a ruta ao elemento seleccionado e permite seleccionar outro elemento para cambiar o foco. Tamén mostra calquera erro ou mensaxes.
- **Content Panel:** contedor para os elementos gráficos.
- **Library Panel:** lista os elementos e controis dispoñibles de JavaFX que poden ser usados para deseñar unha interface gráfica. Pode seleccionarse calquera elemento e engadilo ao panel de contido.

Fixarse que hai un campo que permite buscar compoñentes polo nome.

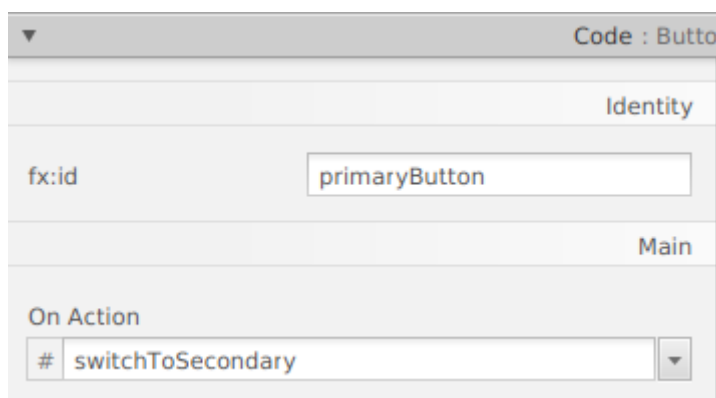
- **Document Panel:** contén as seccións *Hierarchy* e *Controller*. A primeira mostra uha representación en forma de árbore dos elementos da interface. A sección *Controller* permite xestionar información do ficheiro controlador e tamén dá información sobre os valores fx:id asignados.
- **Inspector Panel:** contén as seccións *Properties*, *Layout* e *Code*. As seccións *Properties* e *Layout* mostran as propiedades do elemento seleccionado. A sección *Code* permite xestionar as accións de manexo de eventos do elemento seleccionado. Este panel tamén ten unha caixa de busca para filtrar propiedades.



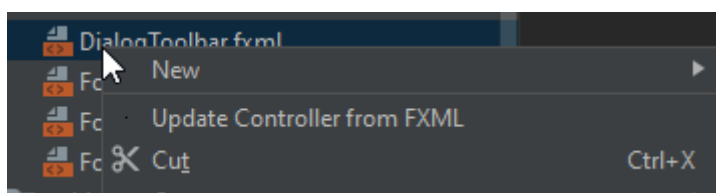
Dende Scene Builder pódese configurar a clase controladora do ficheiro .fxml. Para iso hai que acceder ao menú **Controller** na esquina inferior esquerda:



Para personalizar a acción a realizar cando sucede un evento, seleccionar o control e acceder á zona **Code** na esquina inferior dereita:



Para poder actualizar automaticamente a clase controladora cos cambios introducidos no ficheiro fxml (creación automática de ids e de métodos), haberá que instalar o plugin [FXMLManager](#). Con este plugin pode pulsarse co botón dereito no ficheiro fxml e actualizar a clase controladora:



A funcionalidade **Update Controller from FXML** permite sincronizar as modificacións feitas no ficheiro XML dende o Scene Builder e o ficheiro controlador. Para exemplificalo imos facer o seguinte:

- No Scene Builder, arrastra un botón dende a librería de controis ao panel.
- No panel de Código (Code), na parte dereita da ventá, asigna a **fx:id** o valor **button2** e a **On Action** o método **handleButtonAction2**.
- Garda o ficheiro en Scene Builder.
- En IntelliJ, pulsa co botón dereito do rato sobre o ficheiro fxml e selecciona **Update Controller from FXML**. Observa que se engadiu o código asociado aos cambios realizados de forma gráfica. Observa que o ficheiro **PrimaryController** foi actualizado cos cambios realizados no ficheiro fxml. Creouse a variable para o botón e o método **handleButtonAction2**.

**NOTA:** Recoméndase actualizar esta clase despois de realizar calquera cambio no ficheiro FXML.

### Exercicio:

1. Elabora unha aplicación para a xestión dunha libreta de direccións. A pantalla principal terá un aspecto coma o da seguinte imaxe, composta por unha barra de menús, un split pane con un TableView do lado esquerdo:



Implementa a funcionalidade dos botóns Novo, Editar e Borrar. Para os botóns editar e borrar hai que comprobar que hai un elemento na táboa seleccionado.

Ademais, cando se selecciona un elemento da táboa, mostraranse os seus detalles no lado dereito do splitPane.

## CSS

JavaFX permite aplicar estilos CSS aos compoñentes, igual que se foran elementos HTML, utilizando a mesma sintaxe que a que se usa nas páxinas web. A única diferenza é que JavaFX ten propiedades específicas, polo que os nomes son lixeiramente diferentes ás propiedades equivalentes web.

Aplicar estilos usando CSS axuda a separar o deseño do código facendo máis fácil os cambios de cada unha das partes por separado. Ademais, permite cambiar o estilo de múltiples compoñentes nun só paso ao usar follas de estilo compartidas.

Existen diferentes métodos para aplicar estilos CSS a un compoñente JavaFX. Estes métodos son:

- **Folla de estilos por defecto de JavaFX:** as aplicacións JavaFX teñen unha folla de estilos por defecto que se aplica a todos os compoñentes. Se non se indica un estilo específico, úsase a folla de estilos por defecto.
- **Folla de estilos específica da escena (Scene):** pode establecerse unha folla de estilos a un obxecto **Scene**. Esta folla de estilos será aplicada a todos os compoñentes engadidos ao grafo de escena dese obxecto. Cando se aplican estilos a unha escena, sobrescríbense os estilos por defecto do apartado anterior.

Para establecer unha folla de estilos a un obxecto Scene utilízase a seguinte instrución:

```
scene.getStylesheets().add("style1/button-styles.css");
```

Observar que o ficheiro **button-styles.css** está no directorio chamado **style1**. JavaFX busca este ficheiro no classpath, polo que o directorio debe estar nun directorio raíz (ou JAR) incluído no classpath da aplicación.

- **Folla de estilos específica do compoñente pai:** é posible establecer unha folla de estilos a todas as subclases de **JavaFX Parent** (clase base de todos os compoñentes que teñen fillos).

Normalmente os estilos establecidos nunha subclase de Parent terán precedencia sobre as regras CSS establecidas para a escena e as da folla de estilos por defecto.

Os compoñentes Layout son típicos exemplos de subclases de Parent. Se se establecen estilos para un layout, os estilos aplicaranse a todos os compoñentes que estean nese layout.

Para establecer estilos a unha subclase de Parent utilízase unha instrución similar á seguinte en Java:

```
vbox.getStylesheets().add("style1/button-styles.css");
```

Tamén é posible establecer estilos dende FXML:

```
// o símbolo @ indica que o css está no mesmo directorio que o ficheiro FXML
<VBox stylesheets="@styles.css">
  <Button fx:id="button1" text="Click me!" id="botonNovo"/>
</VBox>
```

```
/*styles.css*/
/* Aplícase a todos os botóns*/
.button {
  -fx-background-color: #0000ff;
}
```

```
/* aplícase ao elemento con id=botonNovo*/
#botonNovo {
    -fx-background-color: #00ff00;
}
```

- **Aplicar estilos a unha propiedade do compoñente:** poden establecerse estilos a un compoñente modificando directamente as propiedades CSS do compoñente.

Os estilos establecidos a unha propiedade teñen prioridade sobre o resto dos estilos: os da subclase Parent, da escena e dos estilos por defecto.

Exemplo en Java e en FXML:

```
button.setStyle("-fx-background-color: #0000ff");
<Button fx:id="button1" text="Click me!" style="-fx-background-color: #0000ff"/>
```

O exemplo anterior establece a cor de fondo do botón a azul.

Poden establecerse múltiples propiedades CSS na mesma cadea:

```
String styles =
    "-fx-background-color: #0000ff;" +
    "-fx-border-color: #ff0000;" ;

Button button = new Button("Button 2");
button.setStyle(styles);
```

Cada nodo nun grafo de escena ten unha clase **styleClass**, de forma análoga ao atributo class dos elementos HTML. Un nodo pode ter máis dunha clase, igual que en HTML.

Cada nodo ten unha variable **id**, análoga á HTML.

As propiedades usadas en JavaFX son moi similares ás da web con CSS, aínda que comezan por -fx-. Exemplos:

- -fx-background-color: cor de fondo.
- -fx-text-fill: cor do texto.
- -fx-font-size: tamaño do texto.

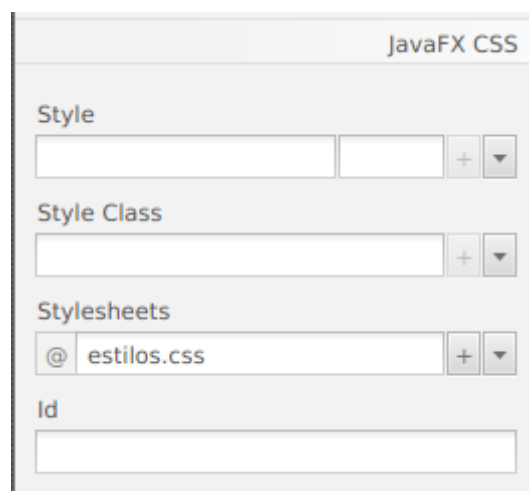
Máis información na [guía de referencia de CSS](#).

Exemplo de ficheiro **estilos.css**:

```
.button {
    -fx-text-fill: white;
    -fx-font-family: "Arial Narrow";
    -fx-font-weight: bold;
    -fx-background-color: linear-gradient(#61a2b1, #2A5058);
    -fx-effect: dropshadow( three-pass-box , rgba(0,0,0,0.6) , 5, 0.0 , 0 , 1 );
}

.button:hover {
    -fx-background-color: linear-gradient(#2A5058, #61a2b1);
}
```

Asignar a folia de estilos ao panel:



Na imaxe anterior:

- Style: permite modificar as propiedades CSS dun nodo particular.
- Style Class: permite asignar unha clase CSS a un nodo.
- Id: permite asignar un Id a un nodo.

## Referencias

Para a elaboración deste material utilizáronse, entre outros, os recursos que se enumeran a continuación:

- [Getting Started with JavaFX](#)
- [Overview \(JavaFX 19\)](#)
- <http://tutorials.jenkov.com/javafx/index.html>
- [JavaFX - Sitio Web de Javier García Escobedo \(javiergarciaescobedo.es\)](#)
- [JavaFX Tutorial](#)
- [Part 1: Scene Builder | JavaFX Tutorial](#)
- [JavaFX Dialogs \(official\)](#)
- [Getting Started with JavaFX: About This Tutorial](#)

- [Getting Started with JavaFX Scene Builder Release 2 - Contents](#)
- [Client Technologies: Java Platform, Standard Edition \(Java SE\) 8 Release 8](#)
- [JavaFX with Gradle, Eclipse, Scene Builder and OpenJDK 11: Refactor with FXML and Scene Builder | The Coding Interface](#)
- [13 Table View \(Release 8\)](#)
- [JavaFX tutorial - learn Java GUI programming in JavaFX](#)
- [JavaFX Tutorial: Getting started | Vojtech Ruzicka's Programming Blog](#)
- [JavaFX Tutorials | o7planning.org](#)
- [JavaFX Tutorial de Introducción](#)
- [Java](#)
- [Introducción a JavaFX y Prerrequisitos | Jairo García Rincón](#)
- [Building a JavaFX Application Using Scene Builder](#)
- [GitHub - mhrimaz/AwesomeJavaFX: A curated list of awesome JavaFX libraries, books, frameworks, etc...](#)
- [JavaFX Documentation Project](#)
- <https://andresalmiray.com/articles/utilizando-fxmlloader/>
- [1 Why Use FXML \(Release 8\)](#)
- [3 Creating an Address Book with FXML - JavaFX](#)
- [Java and OOP](#)