by Nacho Iborra and Arturo Bernal

Development Environments Block 3 Unit 3: Introduction to GUI. JavaFX

3.3.1. What is JavaFX?

JavaFX is a set of Java packages that lets us create a wide variety of graphical user interfaces (GUI), from the classical ones with typical controls such as labels, buttons, text boxes, menus, and so on, to some advanced and modern applications, with some interesting options such as animations or perspective.

If we look backwards, we can see JavaFX as an evolution of a previous Java library, called Swing, that is still included in the official JDK, although it is becoming quite obsolete, and the possibilities that it offers are much more reduced. That is why now most of the Java desktop applications are being developed with JavaFX. At the beginning, it was distributed as an additional library that we needed to add to our projects. In Java version 8 it was included in Java core, but from version 11 it is, again, a separate library So we need to download it and link it to our projects. However, it can be integrated with some of the most popular Java IDEs, such as Eclipse, NetBeans or IntelliJ. This allows us to:

- Create JavaFX applications directly from our preferred IDE.
- Run our JavaFX programs on any device that runs Java 8+ applications (desktops, laptops, tablets, mobile phones...)

3.3.1.1. JavaFX download and installation

In order to download JavaFX library, we must download it here. In our case, we are going to use version 11 (the one associated to our current operating system). We just need to unzip the file and place it at a concrete, easy-to-access folder (for instance, `C:\openjfx`). You will see that it contains a set of *.jar* files, which will be added later to our projects.

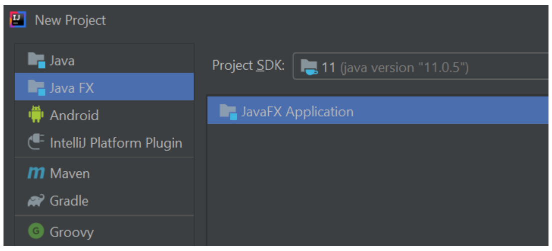
3.3.1.2. What else do I need to install before going on?

It's also recommended that you install **Scene Builder** http://gluonhq.com/labs/scene-builder/. This is an external program we'll use to create our JavaFX graphical user interfaces (GUI). This tool generates a FXML file that defines the list of componentes (buttons, labels and so on) to be added to the JavaFX view. We could create the view manually using Java code, but this method lets us see the immediate result of what we're doing, makes interface design much faster, and also keeps the view separated from the rest of the code.

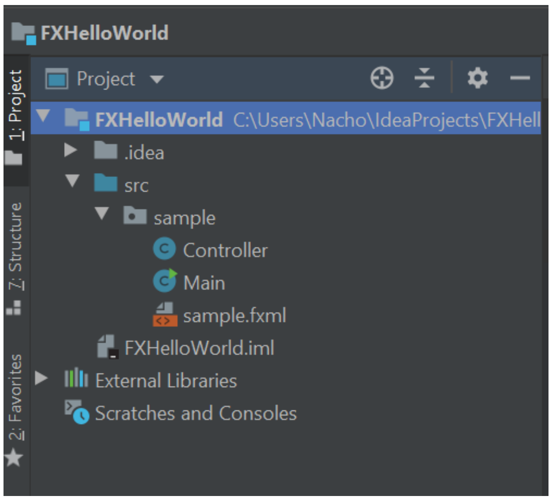
> **IMPORTANT**: make sure you download and install a version of Scene Builder according to your current JDK version. In other words, if you installed JDK 11, you need to install SceneBuilder 11.

3.3.2. Creating our first JavaFX application

In order to use Scene Builder to create our interfaces (views) we must create a new "**JavaFX application**" project, so the IDE creates the necessary files following the Model-View-Controller pattern.



As you can see in the left panel, 3 source files will be created inside `src` folder:



- `sample.fxml` is the FXML file that will be edited from SceneBuilder as we add components to our application.
- `Controller` class will be used to specify the code associated to the FXML file. Every variable name or event that we define to respond to a user action, will be added to this file.
- `Main` class is the main application. You can have a look at the code inside this file. It just launches the application itselt, but all the logic needs to be implemented in the `Controller` class.

Notice that main application is a subtype of `Application` class. This class will hold the main method that will start our application. In this example the file `Main.java` contains our application class (and the main method) that will initialize everything.

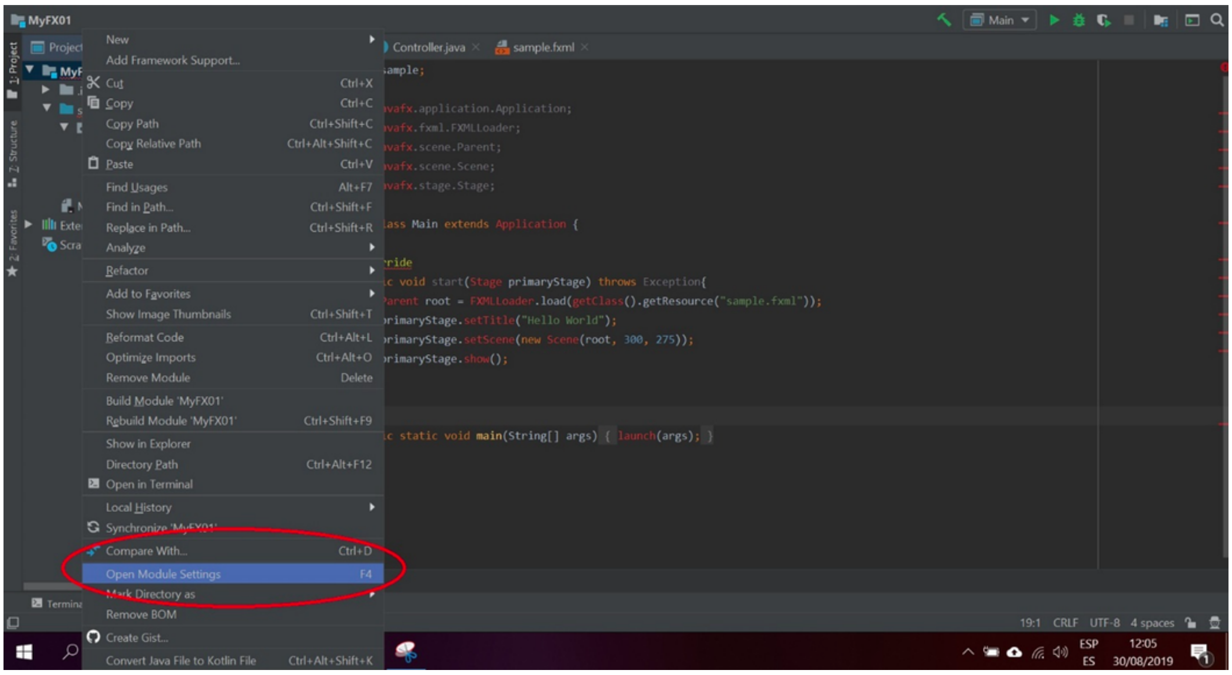
```java
public class Main extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        Parent root =
            FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }

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

As you can see, the application class inherits from `javafx.application.Application`, transforms our FXML view into a Java object (the main scene node containing all the other nodes from the scene), and puts that into a Scene object which will be shown by the Stage object (main window).

3.3.2.1. Adding JavaFX library

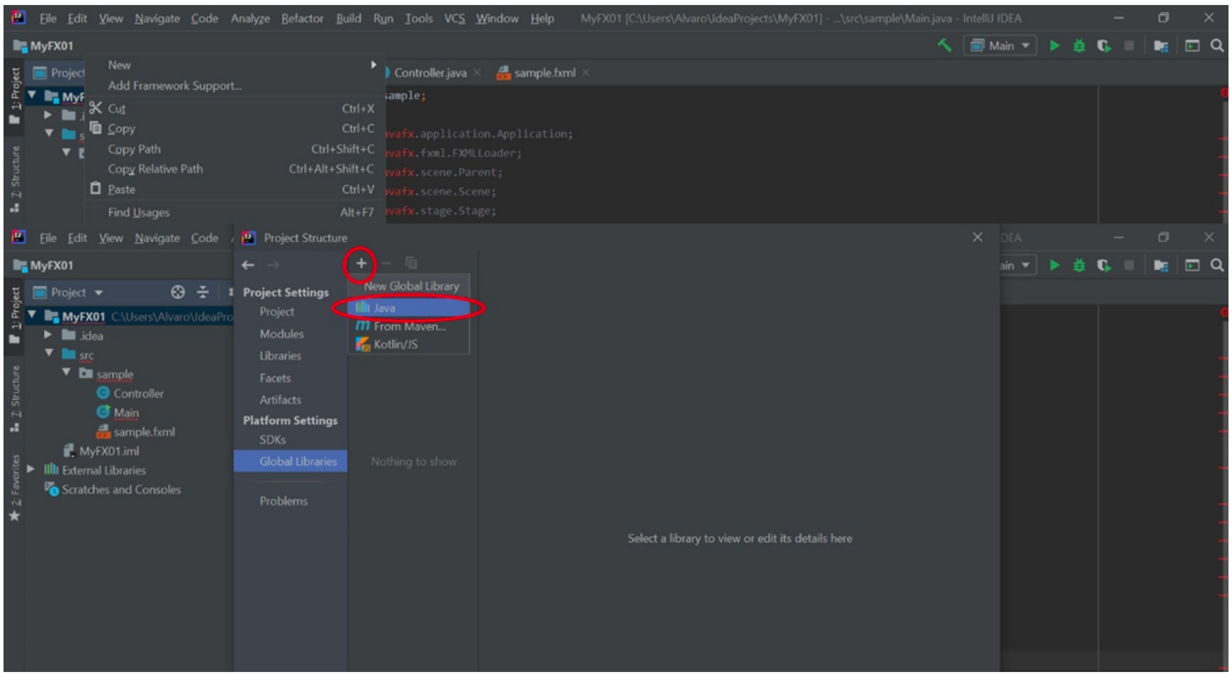
If your project does not compile or run once you have created it, you may need to add JavaFX libraries to IntelliJ global settings. Follow these steps:

- Right click on the project's name in the left panel, and choose *Open Module Settings* option.

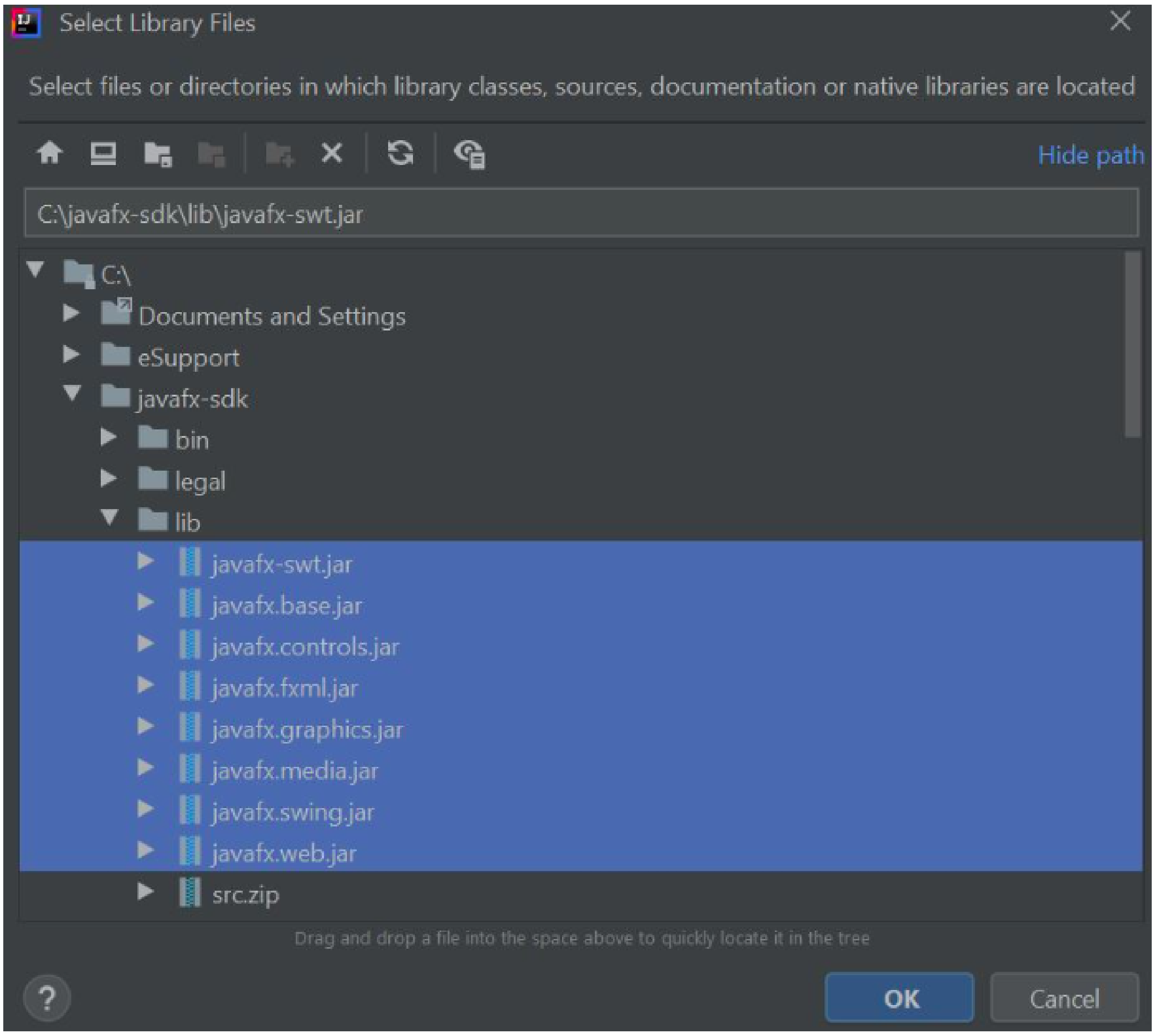


```
sample;

vafx.application.Application;
vafx.fxml.FXMLLoader;
vafx.scene.Parent;
vafx.scene.Scene;
vafx.stage.Stage;

lass Main extends Application {

ride
c void start(Stage primaryStage) throws Exception{
Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
primaryStage.setTitle("Hello World");
primaryStage.setScene(new Scene(root, 300, 275));
primaryStage.show();


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

- Choose *Global Libraries* option, then click on the + button and choose *Java*



- Choose the folder in which you copied the JavaFX files, and choose every JAR file:



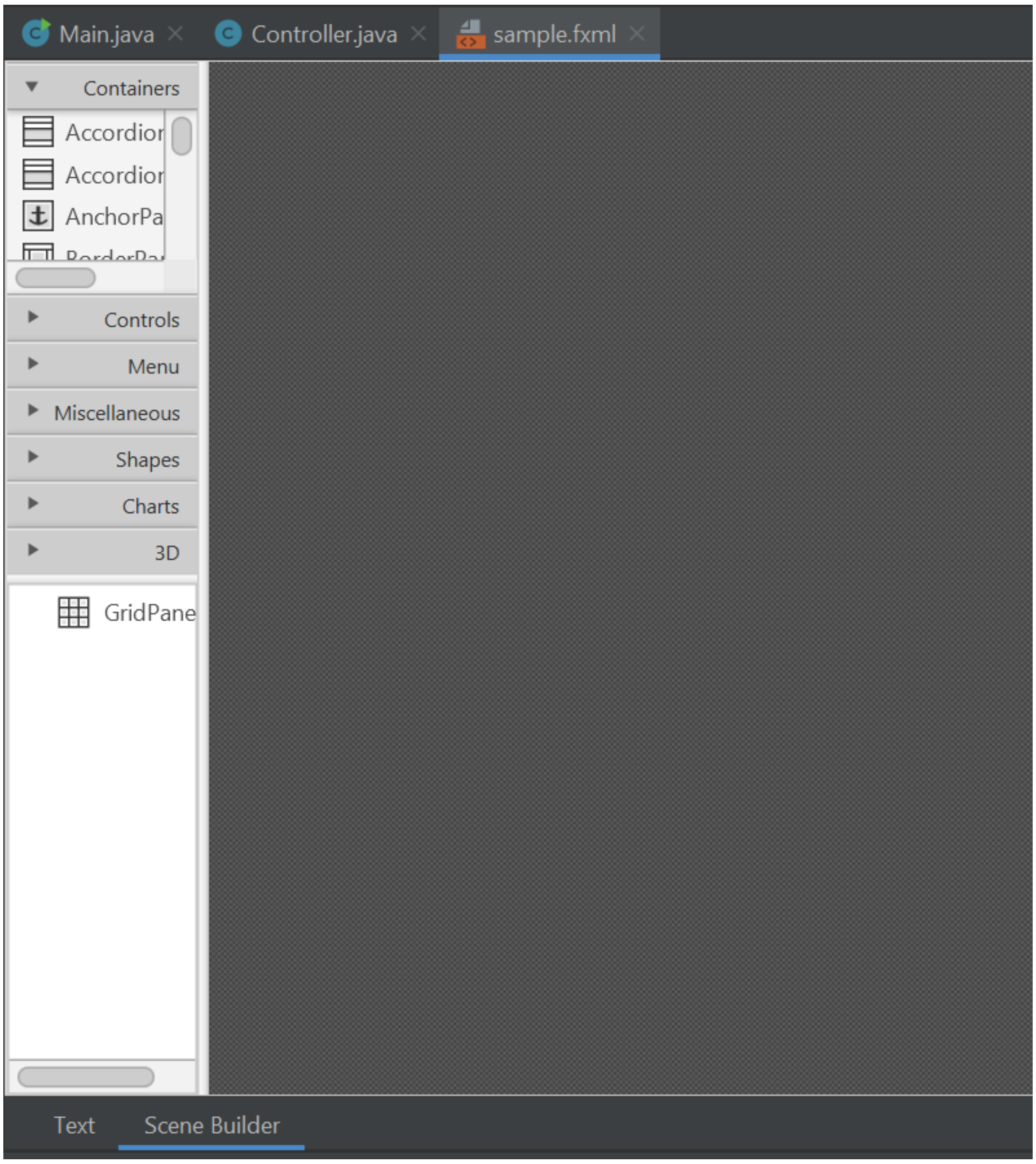
Now you should be ready to run your JavaFX project and see an empty window.

3.3.2.2. Editing the FXML view with SceneBuilder

To open the FXML file with Scene Builder (as long as you already have Scene Builder installed in your system), you just have to double click on it. The FXML source file will be opened in the main area, but you can see two tabs at the bottom, to swap between *Text* and *Scene Builder*. This is what you see from the *Text* tab:

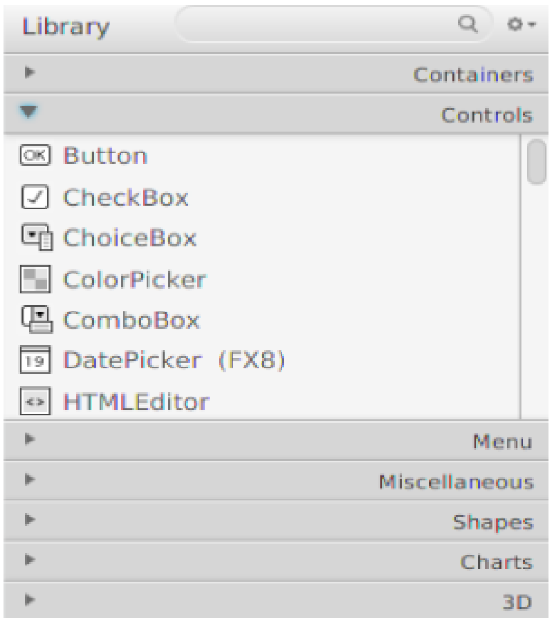
```
<?import javafx.geometry.Insets?>
<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<GridPane fx:controller="sample.Controller"
    xmlns:fx="http://javafx.com/fxml" alignment="center"
    hgap="10" vgap="10">
</GridPane>
```

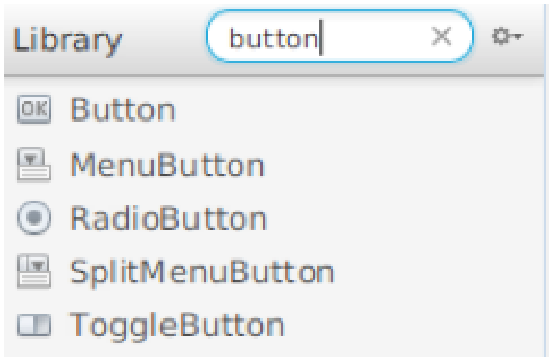
If you choose the *Scene Builder* tab (or if you right click on the FXML file and choose *Open In SceneBuilder* option), Scene Builder editor will be opened with a default, empty view.



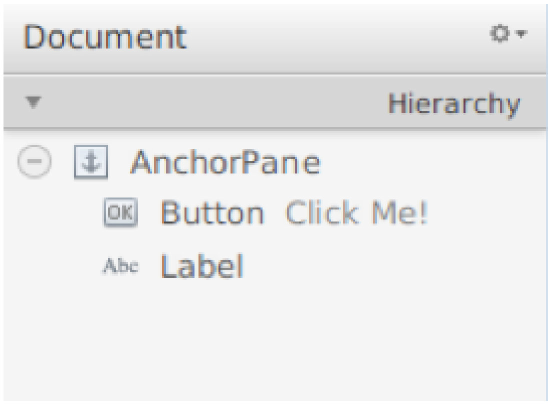
3.3.2.2.1. Understanding Scene Builder main window

At the top left part of the application, we have the JavaFX Object's library, from which we can select the widget we need and drag it to the view (scene). You can also use the search box to find what you are looking for more quickly.



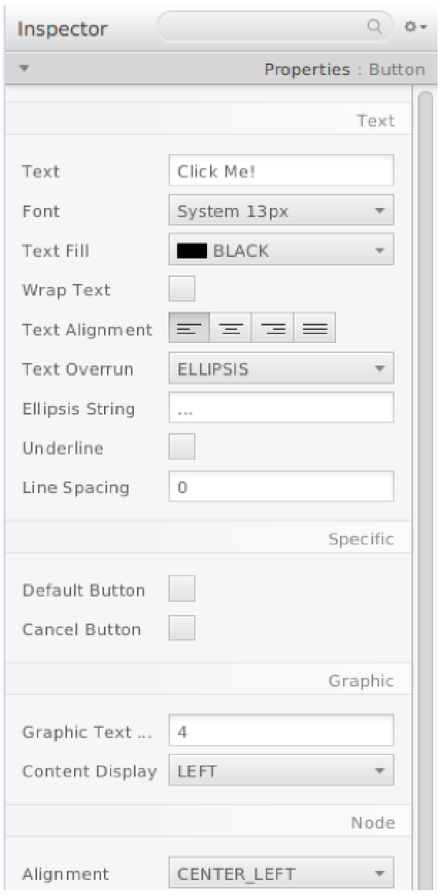


At the bottom left, you'll see the scene's object's hierarchy. There you can also drag the elements and control which elements are inside other elements.



At the right part of the application you'll find the current selected object's inspector from which you'll be able to change its properties (visual and code).

- From the **Properties** tab you can change the text inside the element (for instance, the text of a label or a button), colors, text alignment, font type and size and so on.
- From the **Layout** tab you can change the padding and margin of the elment, along with the anchor point, this is, the point of the main container to which the element is anchored, so that if we resize the window, the element will keep the same distance with the chosen anchor point(s).
- From the **Code** tab you can specify the *id* of the object (*fx:id*) to be used in the controller's code, and the method which will be called when an event (example: action on a button) is triggered for that object.



Inspector

Properties : Button

Text

Text	Click Me!
Font	System 13px
Text Fill	BLACK
Wrap Text	☐
Text Alignment	
Text Overrun	ELLIPSIS
Ellipsis String	...
Underline	☐
Line Spacing	0

Specific

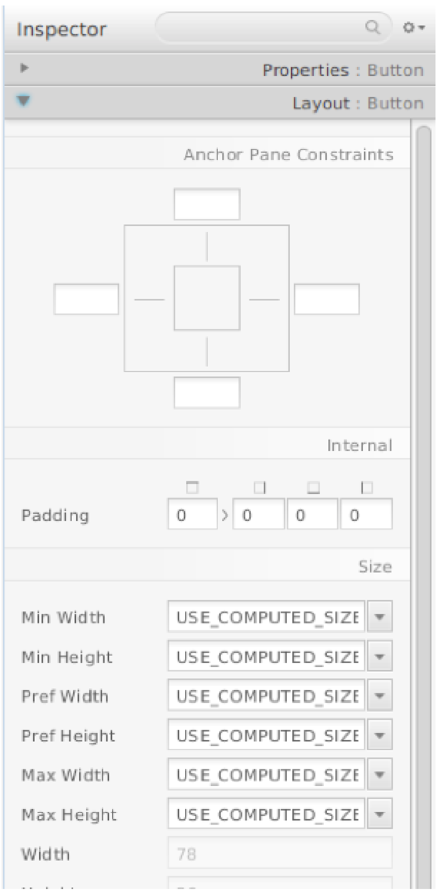
Default Button	☐
Cancel Button	☐

Graphic

Graphic Text ...	4
Content Display	LEFT

Node

Alignment	CENTER_LEFT



Inspector

Properties : Button

Layout : Button

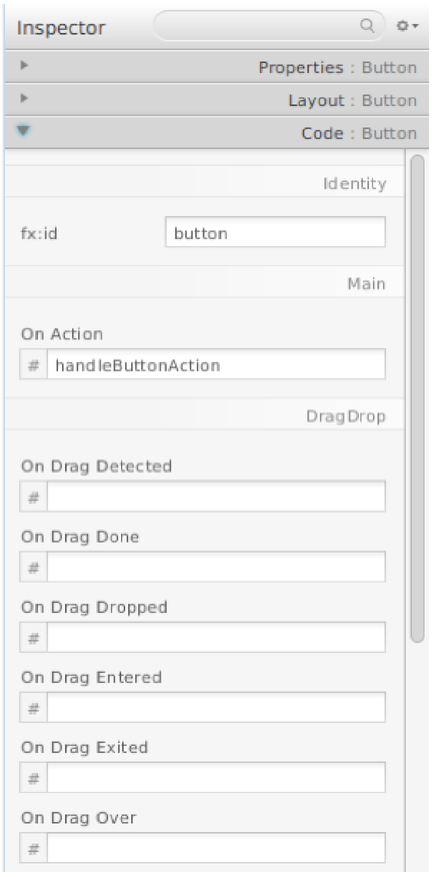
Anchor Pane Constraints

Internal

Padding	0	0	0	0

Size

Min Width	USE_COMPUTED_SIZE
Min Height	USE_COMPUTED_SIZE
Pref Width	USE_COMPUTED_SIZE
Pref Height	USE_COMPUTED_SIZE
Max Width	USE_COMPUTED_SIZE
Max Height	USE_COMPUTED_SIZE
Width	78



Inspector

Properties : Button

Layout : Button

Code : Button

Identity

fx:id	button

Main

On Action	# handleButtonAction

DragDrop

On Drag Detected	#
On Drag Done	#
On Drag Dropped	#
On Drag Entered	#
On Drag Exited	#
On Drag Over	#

If you want to add any component to the application, you just need to drag it over the main Scene Builder area, and place it at your desired position. But before doing this, let's see what kind of components we can use.

3.3.2.3. JavaFX containers and controls

Now that we have learnt the basics about Scene Builder, let's see the main elements that we can include in these applications. These components can be found in the upper left panel, inside the *Containers* or *Controls* tab, respectively.

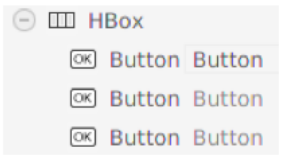
3.3.2.3.1. JavaFX containers

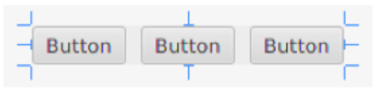
Every control that we can place in an application, such as buttons, labels and so on, must be placed inside a **container**, also known as *layout managers*. These components let us arrange the controls in a given way, so that we don't need to take care of placing the components in their positions manually.

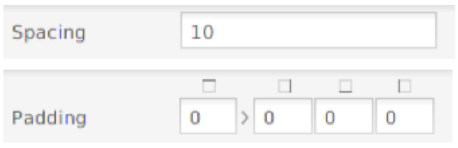
Some of the most common containers in JavaFX are:

- **HBox**: arranges controls horizontally, one next to the other.
- **VBox**: arranges controls vertically, one above/below the other.

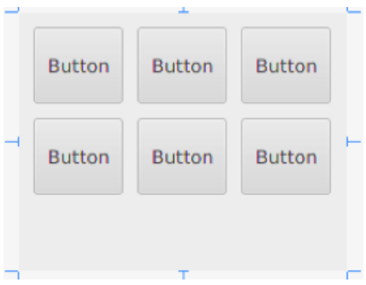
These two containers have some interesting properties in the right *Properties* tab, such as *Spacing* to automatically separate each control from the rest.

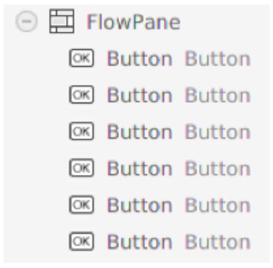


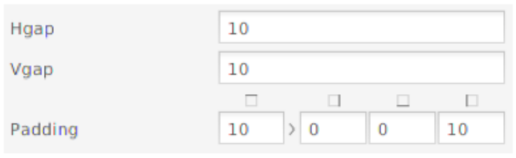



Spacing 10
Padding 0 0 0 0

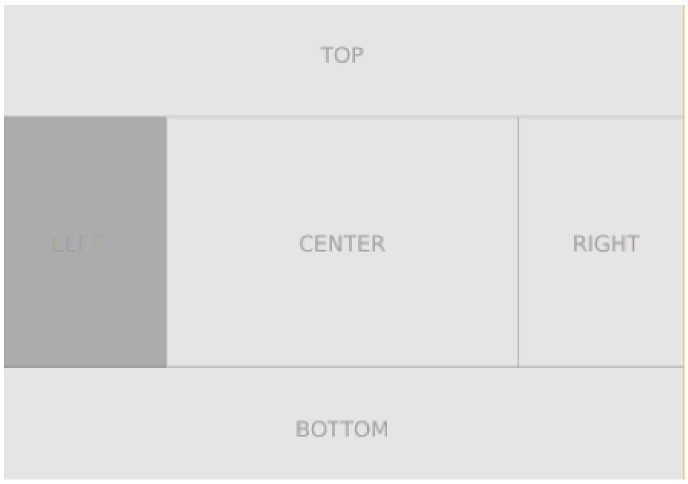
- **FlowPane**: arranges controls next to each other until there is no more space (vertically or horizontally). Then, it goes to next row (or column, depending on its configuration) to keep on arranging more controls. In a *FlowPane* we can control the space between elements horizontally (Hgap) and vertically (Vgap).

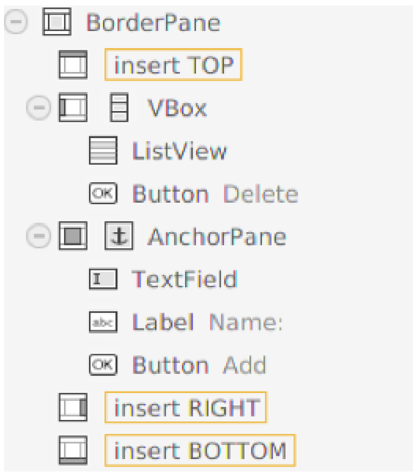




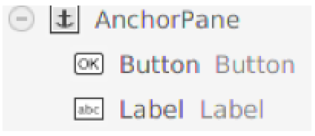

Hgap 10
Vgap 10
Padding 10 0 0 10

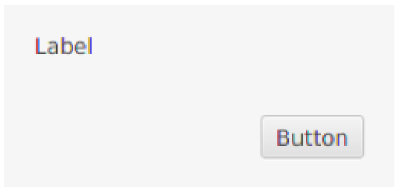
- **BorderPane**: this layout divides the pane into five regions: top, bottom, left, right and center, and we can add a control (or a container with some controls) in each region.

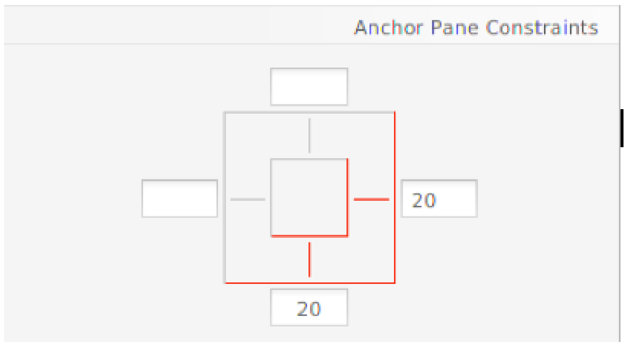


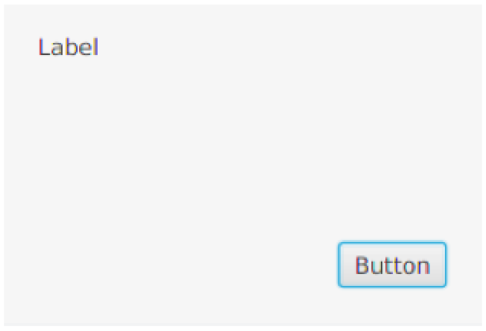


- **AnchorPane**: this layout enables you to anchor nodes to the top, bottom, left side, right side, or center of the pane. As the window is resized, the nodes maintain their position relative to their anchor point. Nodes can be anchored to more than one position and more than one node can be anchored to the same position.









There are other layout panes, such as *GridPane* (it creates some kind of table in the pane to arrange the controls), *TilePane* (similar to FlowPane, but leaving the same space for each control), and so on.

3.3.2.3.2. JavaFX controls

Once we have chosen the appropriate container(s) for our application, we need to place the controls inside it/them. It is important to assign an id (*fx:id* from the *Code* tab) to each control that need to be accessed from the Java code, so that a variable will be created in the corresponding controller.

The most common controls that we can find in many JavaFX applications are:

- **Labels**: they display some text in the scene. Once we have put the label inside a container, there are some useful methods in the `Label` class, such as `getText` or `setText`, to get/set the text of the label.
- **Buttons**: they let us click on them to fire some action. We can specify the text of the button either in the constructor or with the corresponding `setText` method, as we do with labels.
- **RadioButtons**: a set of buttons where only one of them can be selected at the same time. We need to define a group (`ToggleGroup` class), and add the radio buttons to it. For instance, if we have three radio buttons to choose three different colors, associated to three variables called `white`, `gray` and `black`, we can add them all to a toggle group and leave one of them selected by default with the following piece of code:

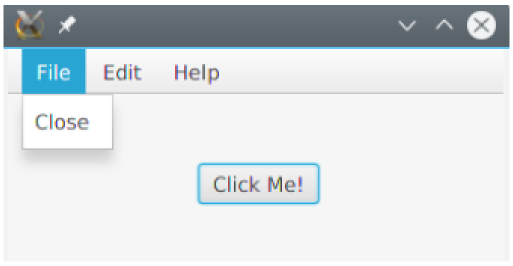
```
ToggleGroup colorGroup = new ToggleGroup();

white.setToggleGroup(colorGroup);
gray.setToggleGroup(colorGroup);
black.setToggleGroup(colorGroup);

colorGroup.selectToggle(white);
```

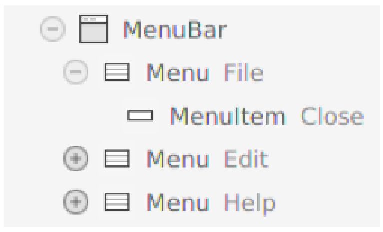
- **Checkboxes**: a control that we can check and uncheck, alternatively, every time we click on it. It has the methods `setSelected` and `isSelected` to check/uncheck it, and determine if it is currently checked.
- **Text fields**: Regarding text fields, the most common controls that we can use in our applications are `TextField` (for short text inputs, with a single line), and `TextArea` (for longer texts, with several rows and columns). There are some methods such as `getText` or `setText` to get and set the text of the control, respectively.
- **Lists**: There are two main types of lists that we can use in any application: `ListView`, with a fixed size, where some elements are shown, and we can scroll the list to look for the element(s) that we want. We also have `ChoiceBox` or `ComboBox` to work with dropdown lists, where only one element is shown, and we can choose any other element by spreading out the list. Anyway, we usually use an ObservableList of items to add elements to these type of lists. There are also some useful methods to get the currently selected item(s), or their indexes. It we have a `ListView` variable called `list`, this is how we would add elements to the list, and then check the currently selected one:

```
list.setItems(
    FXCollections.observableArrayList(
        "Windows", "Linux", "Mac OS X"));
...
String element = myList.getSelectionModel().getSelectedItem();
```

If we are using a dropdown list (such as `ChoiceBox` or `ComboBox`), then we can just call `getValue` method to get the currently selected value, instead of calling `getSelectionModel().getSelectedItem()`, which is more appropriate for fixed lists.

- **Menus**: As in many desktop applications, we can add a menu to our JavaFX application (this is not usual when we are developing a mobile application). The pattern that we usually follow is to put a menu bar (with default menus inside that we can edit), define the categories (Menu), and add menu items to the categories.





As you can see, we need to use three different elements:

- **MenuBar** to define the bar where all menus and menu items will be placed
- **Menu** to define the categories for the menu items. A category is an item that can be displayed, but it has no action (if we click on it, nothing else happens but showing the items that this category contains).
- **MenuItem** defines each item of our menus. If we click on an item, we can define some code associated to that action, as we will see when talking about events. In the example above we have defined a menu

item called "Close" inside File menu. There are also some `MenuItem` subtypes, such as `CheckMenuItem` (items that can be checked/unchecked, like checkboxes), or `RadioMenuItem` (groups of items where only one of them can be checked at the same time, like radio buttons). We can also use a `SeparatorMenuItem` to create a separation line between groups of menu items.

3.3.2.4. Synchronizing the controller class

A controller class is associated to a view (FXML) and binds those view's controls and methods to its Java code. You can see which controller class is binded to a view looking at the `fx:controller` field in the first FXML tag.

```
<GridPane fx:controller="sample.Controller" ...
...
```

If we change the main container of the application (for instance, if we remove the default `GridController` and add an `AnchorPane`), we need to manually specify the controller associated to this new main container:

```
<AnchorPane fx:controller="sample.Controller" ...
```

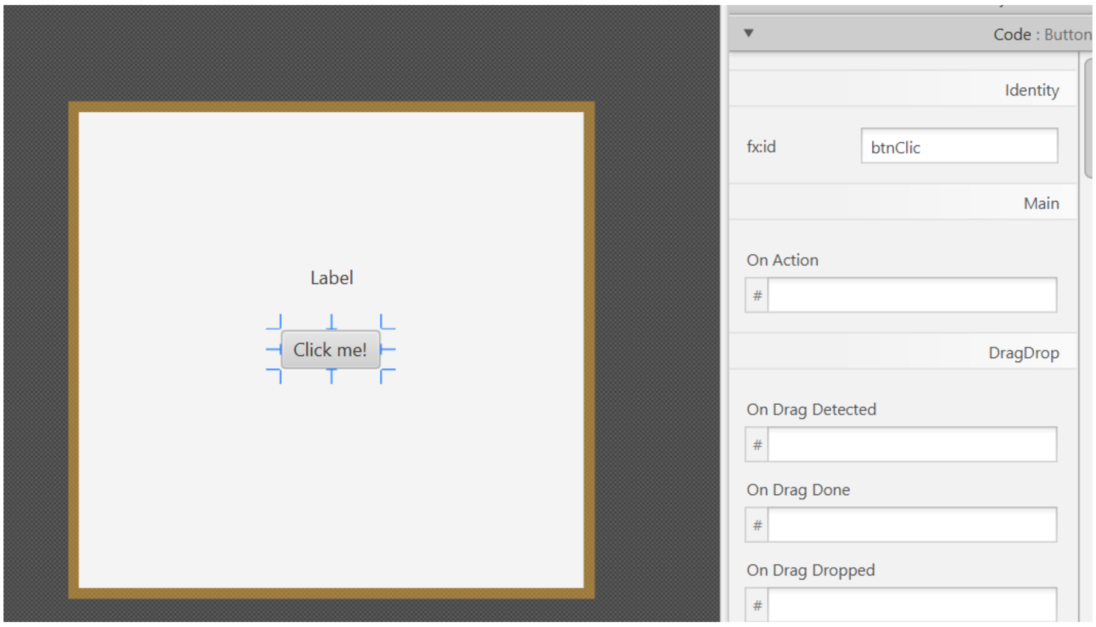
3.3.2.4.1. Adding some changes to the controller class

We need to add some manual changes to the controller before getting it linked to the FXML file. To be more precise, we need to implement `Initializable` interface, so that `initialize` method will be added. This method is automatically called when showing the application, at the beginning, so all the code related to control initialization, file loading and so on must be started from here.

```
package sample;

import javafx.fxml.Initializable;
import java.net.URL;
import java.util.ResourceBundle;

public class Controller implements Initializable
{
    @Override
    public void initialize(URL url,
        ResourceBundle resourceBundle)
    {

    }
}
```

3.3.2.4.2. Updating controller class from FXML file

As we add controls and containers to the application in SceneBuilder, they are automatically added to the FXML file but, in order to have them accessible from the controller Java code, we need to assign them an id (*fx:id* property in the *Code* tab of Scene Builder).

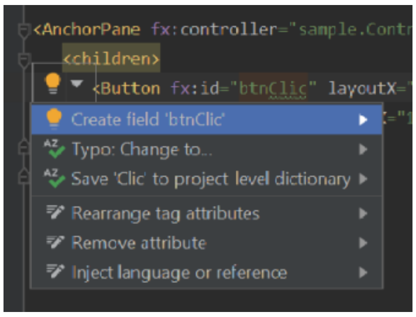


fx:id btnClic

This *id* will be automatically added to the FXML file, as a property of the corresponding control:

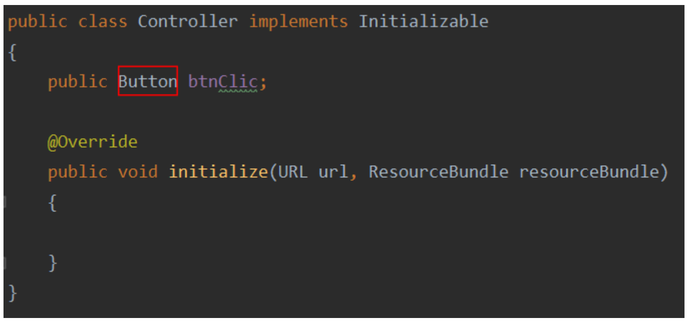
```
<Button fx:id="btnClic" layoutX="131.0" layoutY="141.0"
mnemonicParsing="false" text="Click me!" />
```

Then, we need to manually add this property to the controller class (as long as the controller is properly attached to the FXML file, as we have explained before). To do this, we can just move the mouse over the corresponding attribute in the FXML file, then go to the "bulb" that will be shown and choose *Create field XXX*, where XXX will be the *id* given to our control:



```
<AnchorPane fx:controller="sample.Contr
    <children>
        <Button fx:id="btnClic" layoutX=
```

This is how our controller should look like after this change:



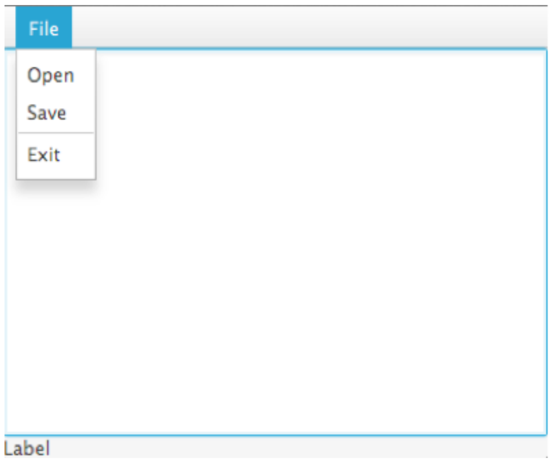
```
public class Controller implements Initializable
{
    public Button btnClic;

    @Override
    public void initialize(URL url, ResourceBundle resourceBundle)
    {

    }
}
```

You can try to launch the project now. It will show a small window with a button, but nothing will happen when pressing it. You may also need to add some changes to `Main.start` method, in order to resize the main window:

```
@Override
public void start(Stage primaryStage) throws Exception{
    ...
    primaryStage.setScene(new Scene(root, 600, 400));
    primaryStage.show();
}
```

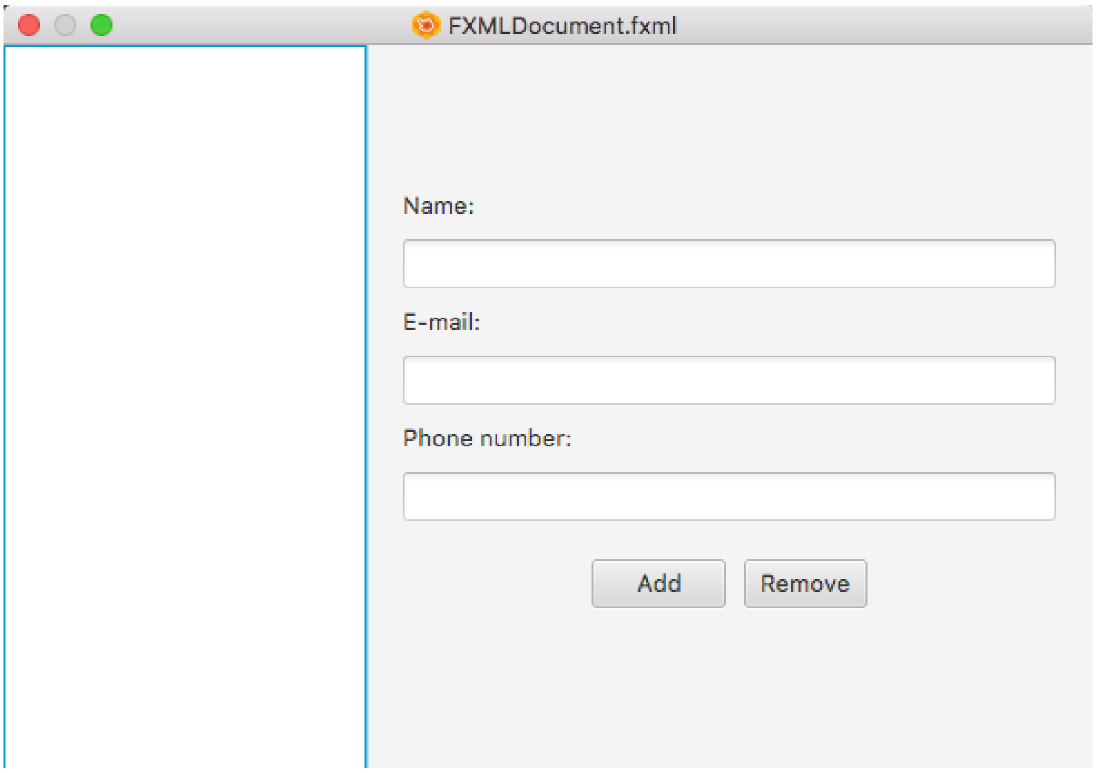
Proposed exercises:

3.3.2.1. Create a new project (remember, a *JavaFX Application*) called *NotepadFX* with the following appearance:



You must choose an appropriate container, and place a menu bar at the top, with the *File* menu containing the specified menu items (*Open*, *Save* and *Exit*). Then, place a *TextArea* in the center and a label at the bottom. Once you have finished placing the container(s) and controls, save your project. We will go on with it in later sections.

3.3.2.2. Create a JavaFX FXML project called *ContactsFX* with the following main window:



> There is a list view on the left, and some labels, text fields and buttons on the right. Choose the appropriate containers and controls to get a similar appearance. Save your project when you are done, we will go on with it later.

3.3.3. Events

If we only add controls to our JavaFX application (buttons, labels, text fields...) we will not be able to do anything but clicking and typing with it. There will be no file loading, data saving, or any operation with the data that we type or add to the application.

In order to allow our application to respond to our clicks and typings, we need to define event handlers. An **event** is something that happens in our application. Clicking the mouse, pressing a key, or even passing a mouse over the application window, are examples of events. An **event handler** is a method (or object with a method) that responds to a given event by executing some instructions. For instance, we can define a handler that, when a user clicks a given button, takes the numerical values from some text boxes, adds them and shows the result.

3.3.3.1. Main event types

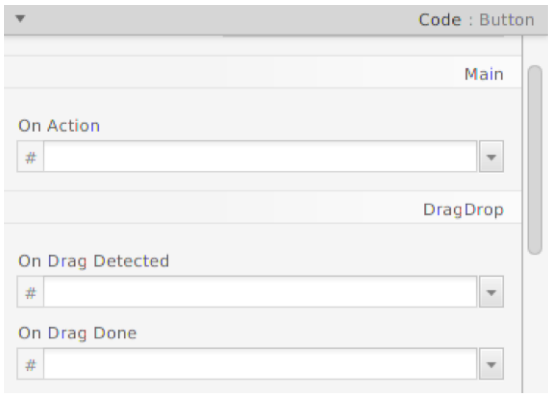
Every event produced in our application is a subclass of `Event` class. Some of the most common types (subclasses) of events are:

- `ActionEvent` : typically created when the user clicks on a button or a menu item (and also when he tabs to the button or menu item and presses the Enter key).
- `KeyEvent` : created when the user presses a key

- `MouseEvent` : created when the user does something with the mouse (click a button, move the mouse...)
- `WindowEvent` : created when the status of the window changes (for instance, it is maximized, minimized, or closed).

3.3.3.2. Defining handlers through Scene Builder

Most common events (but not all) can be connected to an event handler with Scene Builder (FXML). In order to do that, we select the element and in the right (*Code* tab) we will see the different event types we can bind to a method in the controller.



We just need to type the handler name in the desired event, save the changes manually add the corresponding method to the controller (as we did for the *fx:ids* of the components before, going to the "bulb" and choosing *Create method...* option). Then, a new event handler will appear in the code:

```
private void handleButton(ActionEvent event)
{
    // Code
}
```

Now, we just need to type the code associated to this handler.

3.3.3.3. Defining handlers by code

We can also define our event handlers in the code of the controller. This can be typically done in the `initialize` method, where every component is initialized as the application starts. Let's see some examples.

Action event over a button

This way we could define an action event over a button whose variable name is `button` :

```
@Override
public void initialize(URL url, ResourceBundle rb)
{
    ...
    button.setOnAction(new EventHandler<ActionEvent>()
    {
        @Override
        public void handle(ActionEvent event)
        {
            // Code
        }
    });
}
```

Event over a list view to detect selection changes

This way we could fire an event whenever we change the currently selected item of a list view (in this case, it is a list view of `String` elements):

```
listView.getSelectionModel().selectedItemProperty().addListener(
    new ChangeListener<String>()
    {
        @Override
        public void changed(ObservableValue<? extends String> obs,
                            String oldValue, String newValue)
        {
            // "newValue" contains the new selected item
            // and "oldValue" the previously selected one
        }
    }
);
```

Proposed exercises:

3.3.3.1. Complete *NotepadFX* project from previous exercises by adding these events:

- If we choose *File* > *Open* menu item, then the program will read a text file called "notes.txt" and write the text in the text area. Besides, the bottom label must show how many line have been read from the text file.
- If we choose *File* > *Save* menu item, the program will get the text of the text area and save it in the *notes.txt* file, erasing any previous content of this file. The bottom label must show a message indicating if the file has been successfully saved or not.
- If we choose *File* > *Exit* menu item, the application will close.

3.3.3.2. Complete *ContactsFX* project from previous exercises with these events:

> - At the beginning (in the `initialize` method), the program will load a list of contacts from a text file called "contacts.txt" (create a `Contact` class for this purpose) and show it in the left list view.
> - Whenever we select any contact from the list view, his information will be shown in the corresponding text fields of the right part of the window.
> - If we click on the *Add* button, then a new Contact with the information of the text fields will be created, and added to the list. Also, the new contacts list will be saved in "contacts.txt" file, erasing any previous contents of the file.
> - If we click on the *Remove* button, then the currently selected contact from the list (if any) will be removed, and the corresponding text file will be updated.

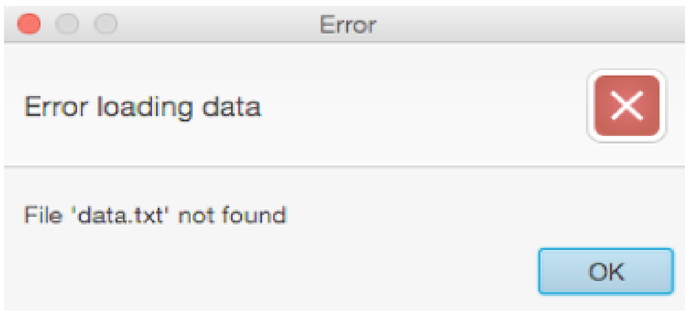
3.3.4. More JavaFX features

To finish with this unit, let's see some additional features that may be interesting for us to create helpful JavaFX applications.

3.3.4.1. Using dialogs

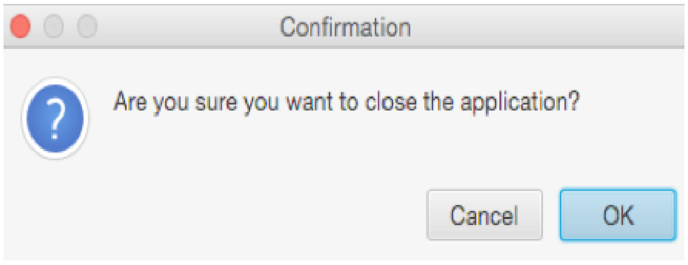
Since JavaFX 8 (or rather, since JavaFX version 8u40) there are some built-in dialogs available for JavaFX. Some of them show information messages or confirmation dialogs. Most of these dialogs can be built from the `Alert` class. It has methods to define the dialog title, header and content, although all these messages are optional. The basic usage of this class is to show basic messages, such as error messages, or information messages.

```
Alert dialog = new Alert(Alert.AlertType.ERROR);
dialog.setTitle("Error");
dialog.setHeaderText("Error loading data");
dialog.setContentText("File 'data.txt' not found");
dialog.showAndWait();
```



We can change the parameter of the constructor (`Alert.AlertType.ERROR`) for any other constant from AlertType class, such as CONFIRMATION, WARNING, INFORMATION... If we use a CONFIRMATION dialog, then two buttons will be shown:

```
Alert dialog = new Alert(Alert.AlertType.CONFIRMATION);
dialog.setTitle("Confirmation");
dialog.setHeaderText("");
dialog.setContentText("Are you sure you want to quit?");
Optional<ButtonType> result = dialog.showAndWait();

if (result.get() == ButtonType.OK)
    // Code for "OK"
else
    // Code for "Cancel"
```


Confirmation

Are you sure you want to close the application?

If we choose the "OK" button, then a value will be returned. Otherwise (if we click on the cancel button, or we close the dialog), no value will be returned.

Another built-in dialog since JavaFX 2.0 is the `FileChooser` dialog. We use it to display a dialog box to make the user choose a file to open/save data from/into. To use this dialog, we can just add these lines to our code:

```
FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("Open Resource File");
File selectedFile = fileChooser.showOpenDialog(stage);
```

There is also a `showSaveDialog` method to let the user save some data into a file, and some other useful methods (check the API for more details).

3.3.4.2. Loading images

It is very usual to add some images to our application to make some controls (or the general appearance) more appealing. To do this, we use the `Image` and `ImageView` classes (from `javafx.scene.image` package). The first one allows us to load an image from a file (normally in the application's root folder, although it can be anywhere). The `ImageView` class adapts the image loaded to something drawable in our application.

To put an image in an FXML application, just drag the *ImageView* control to the scene in Scene Builder. Then, write the image path relative to the FXML file (inside the *src* folder).

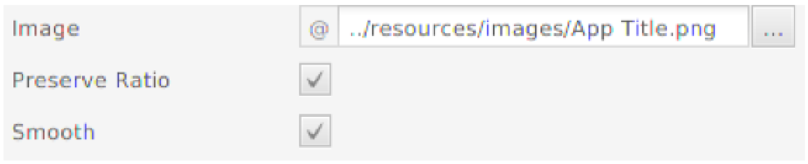
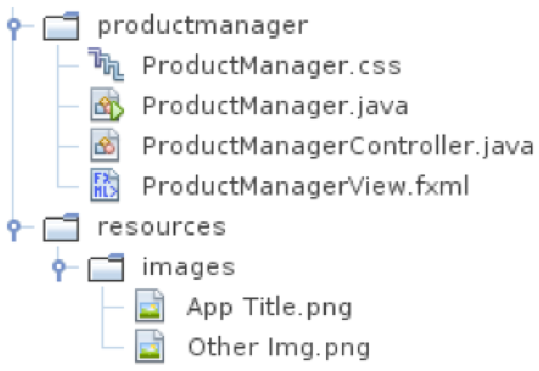


Image	../resources/images/App Title.png
Preserve Ratio	✓
Smooth	✓

We can change the image shown in the *ImageView* control in code, by setting a new `Image` object with another path (start the path with '/' to make it relative to the *src* folder).



```
productmanager
    ProductManager.css
    ProductManager.java
    ProductManagerController.java
    ProductManagerView.fxml
resources
    images
        App Title.png
        Other Img.png
```

```
imgTitle.setImage(new Image("/resources/images/Other Img.png"));
```

Proposed exercises:

3.3.4.1. Improve previous *NotepadFX* exercise by adding a *FileChooser* dialog to let us choose the file to be read or saved from the application, when we choose *File > Open* and *File > Save* menu items, respectively.

3.3.4.2. Improve previous *ContactsFX* exercise by adding some alerts:

- Show an error alert if we try to add a new contact with some empty field.
- Show an information alert if the contact has been successfully added to the list.