

2. Introduction to JavaFX

Service and Process Programming

Arturo Bernal
Nacho Iborra

IES San Vicente



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Table of Contents

2. Introduction to JavaFX

1. First steps with JavaFX.....	3
1.1. <i>What is JavaFX?</i>	3
1.2. <i>Basic structure of a JavaFX application</i>	3
1.3. <i>Our first JavaFX application: Hello world!</i>	4
2. Basic controls and layouts.....	6
2.1. <i>The Stage and Scene classes</i>	6
2.2. <i>Some of the most useful controls in JavaFX</i>	7
2.3. <i>Organizing controls. The "layout" package</i>	10
3. Events.....	14
3.1. <i>Main event types</i>	14
3.2. <i>Defining handlers and connecting them with events</i>	14
3.3. <i>Examples</i>	17
4. More about stages and layouts.....	22
4.1. <i>Adding images to our application</i>	22
4.2. <i>Common control properties: borders and colors</i>	22
4.3. <i>Dialog boxes</i>	23
4.4. <i>Defining controls and layout through XML files</i>	27
4.5. <i>Changing the default appearance with CSS</i>	29
5. Some advanced features.....	32
5.1. <i>Special effects</i>	32
5.2. <i>Animations</i>	34
5.3. <i>Multi-device deployment</i>	36
6 Basic 2D Game Programming.....	37
6.1 <i>Rendering on a Canvas</i>	37
6.2 <i>The game loop</i>	38
6.3 <i>Events and interaction</i>	39
6.4 <i>Sprites</i>	40

1. First steps with JavaFX

1.1. 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. It was first an additional library that we needed to add to our projects, but from Java version 8 it is included in Java core (if using OpenJDK in Linux make sure to install the **openjfx** package). This allows us to:

- Create JavaFX applications directly from our preferred IDE (Eclipse, Netbeans...), without installing anything else.
- Run our JavaFX programs on any device that runs Java 8 applications (desktops, laptops, tablets, mobile phones...)

1.1.1. What do I need to install before going on?

The only thing we need to have installed in our computer to start coding our JavaFX applications is JDK (version 8 preferably) and an appropriate IDE, such as Eclipse or Netbeans. In this tutorial, we are going to use NetBeans IDE, since it is one of the most popular ones, and we assume that you have both installed (JDK + IDE) by now. If not, please go to the Oracle Java page to download and install and install JDK 8 (or later), and to the Netbeans page (netbeans.org) to download the Netbeans IDE. You can also download and install both at the same time by typing "netbeans java" on Google and going to the Oracle web page that offers them as a bundle.

1.2. Basic structure of a JavaFX application

When we start creating a JavaFX program, we need a main class that extends the ***Application*** class (from *javafx.application* package). When we do this, we will also need to override the ***start*** method from that class. This is the method that will be run when our JavaFX application will be launched. Finally our *main* method will call ***launch*** method from *Application* class, that is in charge of preparing the environment and calling the *start* method that we have defined. Putting all of this together, the basic schema for our main class would be like this:

```
import javafx.application.Application;
import javafx.stage.Stage;

public class MyFirstJavaFXApplication extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.show();
    }
}
```

```

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

```

The *start* method has a parameter of type *Stage*. This class is a reference to the main window, and it will be explained later. Our *start* method just shows the application window for now. If we try to run the program at this moment, we will only see a blank window.

1.3. Our first JavaFX application: Hello world!

When we are starting with a new programming language, the first program that we are asked to write is one that simply writes "Hello world" in the screen. In a console application, this program might take just two or three lines of code, but if we are creating a graphical application, it is slightly more difficult, since we need to import several packages, add some controls to the window and launch it.

In previous example we have already built the basic structure of our JavaFX application. From then on, we are going to add a control to the window to show a "Hello world!" message. As we have said before, JavaFX comes with a complete bunch of classical controls such as labels, buttons, and so on. We are going to add a Label to the Window to show our message. To do this, we need to define this *start* method:

```

@Override
public void start(Stage primaryStage)
{
    Label lblMessage = new Label("Hello world!");
    BorderPane pane = new BorderPane();
    pane.setCenter(lblMessage);
    Scene scene = new Scene(pane, 500, 400);
    primaryStage.setScene(scene);

    primaryStage.show();
}

```

We will need to add the packages for the new classes (*javafx.scene.control.Label*, *javafx.scene.layout.BorderPane*, *javafx.scene.Scene*). But apart from that, let's explain briefly what each line of code does:

```
Label lblMessage = new Label("Hello world!");
```

We create a control of type *Label* with the text *Hello World!* on it.

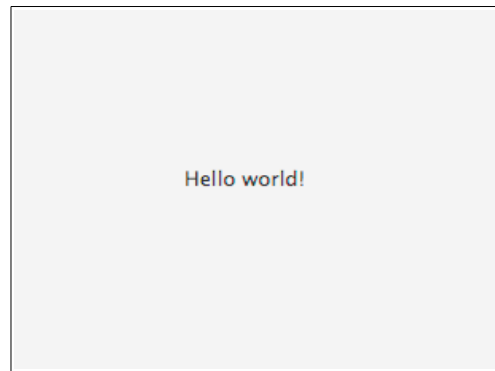
```
BorderPane pane = new BorderPane();
pane.setCenter(lblMessage);
```

We define a layout manager of type *BorderPane*. These layout managers are in charge of displaying the controls in the specified positions (our label, in this case, that is put in the center of the layout manager).

```
Scene scene = new Scene(pane, 500, 400);
primaryStage.setScene(scene);
```

We define a *Scene*, this is, the main container of everything that we want to add to our window. In the constructor, we define the layout manager that will be added to the scene, and the dimensions of the window (500 pixels width, 400 pixels height). Then, we call the *setScene* method from our *Stage* object to add or establish the scene of our window.

If we run our program now, we will see something like this:



And that is it! We have our *Hello world* program in JavaFX. Notice that we have needed six lines of code inside our *start* method (and also some imports, and a *main* and *start* methods) to create it.

1.3.1. Regarding the *imports*

In the example shown, we have added an *import* for every new class that we need. However, in our JavaFX applications we will need more than just four or five classes, and most of them belong to the packages that we have previously added. So it is a good idea to import everything from the packages that we have already included:

```
import javafx.application.*;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.*;
```

Anyway, throughout this unit you will see a lot of new classes. In most cases the package to which these classes belong will be omitted, but if you use an IDE like NetBeans or Eclipse, it is easy to add the corresponding import: if you get a compilation error when you add a new class name to your code, place the cursor in the class name and press *Alt + Enter*. A list of options to solve the error will be shown, and you only have to choose the one that imports the corresponding *javafx* package.

In next sections we will learn how to add controls to our application in different ways, and how to make them respond when the user interacts with them. We will finish the unit with some advanced features added in JavaFX that are not present in previous graphical libraries, such as animations, lighting and other effects.

2. Basic controls and layouts

In this section we will learn some of the most important basic concepts to develop JavaFX applications, such as some typical controls that we can use, and how to arrange them in the window. Later in this unit we will learn other ways of doing this without typing any Java code (with XML and CSS files).

2.1. The Stage and Scene classes

2.1.1. The Stage class

As we have already seen in previous (and simple) examples, when we create a JavaFX application, we need a class that extends the *Application* class, and then we override a *start* method that has a *Stage* object as a parameter. The *Stage* object is a reference to the main container of our application. This will be a window in operating systems like Linux, Windows or Mac OS X, but it can be the full screen if our application runs on a smartphone or a tablet.

The *Stage* class provides some useful methods to change some features (size, behaviour...). Some of the most useful methods are:

- ***setTitle(String)***: sets the application title (it is visible in the upper bar of the window).
- ***setScene(Scene)***: sets our application scene (where all the controls will be placed). We will learn later that there can be more than one scene in a stage.
- ***show***: makes the application (stage) visible, and keeps on running next instructions
- ***showAndWait***: makes the application (stage) visible, and waits until it is closed before going on.
- ***setMinWidth(double)*, *setMaxWidth(double)***: set the minimum and maximum width (respectively) of the window, so that we will not be able to resize it beyond these limits.
- ***setMinHeight(double)*, *setMaxHeight(double)***: set the minimum and maximum height (respectively) of the window, similar to the width methods seen before.
- ***getMinWidth*, *getMaxWidth*, *getMinHeight*, *getMaxHeight***: get the maximum or minimum width or height of the application.
- ***setFullScreen(boolean)***: sets if our application will run in full screen mode (so it will not be resizable, and there will not be any upper bar), or not.
- ***setMaximized(boolean)***: sets if our application is maximized or not.
- ***setIconified(boolean)***: sets if our application is iconified (minimized) or not.
- ***setResizable(boolean)***: sets if our application is resizable or not.

<https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html>

For instance, with these lines inside the *start* method we can define the window title, and the maximum and minimum size for our window (if we resize it):

```
@Override
```

```
public void start(Stage primaryStage)
{
    primaryStage.setTitle("My first JavaFX application");
    primaryStage.setMinimumWidth(200);
    primaryStage.setMaximumWidth(500);
    primaryStage.setMinimumHeight(100);
    primaryStage.setMaximumHeight(400);
    ...
    primaryStage.show();
}
```

2.1.2. The Scene class

Every JavaFX program has (at least) one *Scene* object to hold all the controls of the application. When we create it, we need to specify its main node (a layout manager, as we have done in previous examples) and (optionally) its initial width and height:

```
Scene scene = new Scene(pane, 500, 400);
```

Then we can find some useful methods inside *Scene* class, such as:

- ***getWidth, getHeight***: gets the scene's current width and height
- ***getX, getY***: gets the scene's current coordinates in the screen (referring its upper left corner)
- ***setRoot (Parent)***: sets a new parent (layout manager) for this scene.

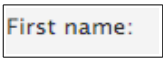
As we have said before, a *Stage* can have multiple *Scenes*, and it can switch between them by calling its *setScene* method.

<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Scene.html>

2.2. Some of the most useful controls in JavaFX

If you look at the JavaFX API, you will find a lot of controls that you can use in your applications. Some of them, such as *TreeViews* or *Accordions* are barely used, but some others such as *Buttons*, *Labels*, *TextFields*... are used in almost every application. We are going to focus on these controls, and explain briefly how to use them. Most of these controls belong to *javafx.scene.control* package.

2.2.1. Labels

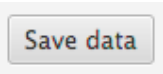


If we want to use a label, we only have to create it and put it in a layout manager (we will see layout managers in next subsection 2.3).

```
Label lbl = new Label("Label text");
```

There are other constructors that we can use to create a label (for instance, a constructor that allows us to add an icon to the label, as we will see later). Once we have created the label, there are some useful methods in the *Label* class, such as ***getText*** or ***setText***, to get/set the text of the label.

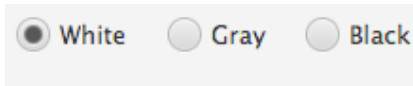
2.2.2. Buttons



If we want to use a button, we create it with its text (there are also other constructors, similar to the ones that we can find for the *Label* class):

```
Button btn = new Button("Button text");
```

2.2.3. Radio buttons



Radio buttons are 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.

```
ToggleGroup group = new ToggleGroup();

RadioButton rb1 = new RadioButton("White");
rb1.setToggleGroup(group);
rb1.setSelected(true);

RadioButton rb2 = new RadioButton("Gray");
rb2.setToggleGroup(group);

RadioButton rb3 = new RadioButton("Black");
rb3.setToggleGroup(group);
```

We can also define the default selected button with the **setSelected** method, as shown in the code above, and we can use the **isSelected** method to check if a given radio button is currently selected.

2.2.4. Checkboxes



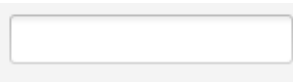
A checkbox is a control that we can check and uncheck, alternatively, every time we click on it.

```
CheckBox cb = new CheckBox("I accept the terms and conditions");
cb.setSelected(false);
...
if (cb.isSelected())
    ...
```

We can also define if the checkbox is initially selected or not with the **setSelected** method, as shown in the code above, and use the **isSelected** method to check if it is currently selected.

2.2.5. Text fields

Regarding text fields, the most common controls that we can use in our applications are *TextFields* (for short text inputs, with a single line), and *TextAreas* (for longer texts, with several rows and columns).



If we want to use a **TextField**, we create it (either empty or with a given initial text):

```
TextField txt = new TextField("Type something");
```

Then, there are some methods such as **getText** or **setText** to get and set the text of the control, respectively. There are also methods like **setPromptText** (to set a text that will be deleted as soon as the user starts typing something in the text field), or **setPrefColumnCount** (to set the number of columns that will be visible in the text field).

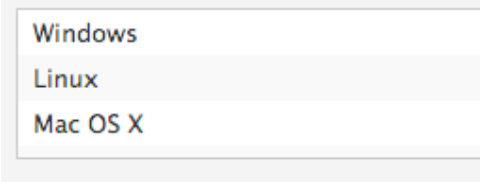
If we want to use a **TextArea**, we create it with an empty constructor (or with an initial text), and then we have two methods to set the initial number of rows and columns:

```
TextArea area = new TextArea("This is a textarea");
area.setPrefRowCount(5);
area.setPrefColumnCount(40);
```


There are some other methods that might be useful, such as **setWrapText** (sets if new lines must be added when text exceeds the length of the text area), or **getText/setText**, as in *TextField* class.

2.2.6. Lists

There are two main types of lists that we can use in any application:

-  Lists with a fixed size, where some elements are shown, and we can scroll the list to look for the element(s) that we want. To work with these lists in JavaFX, we have the **ListView** control. Here is an example of how to use it:

```
ObservableList<String> items = FXCollections.observableArrayList(
    "Windows", "Linux", "Mac OS X");
ListView<String> myList = new ListView<String>(items);
myList.setPrefHeight(80);
```

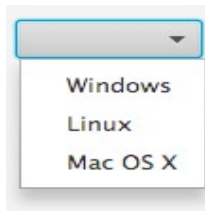
We need to indicate the type of elements that will be showed in the list (in this example, we work with *Strings*). Then, we create the list of objects by calling the *FXCollections.observableArrayList* method, and finally we create the *ListView* object with these items. The method **setPrefHeight** is useful to control the vertical size of the list (otherwise, it may be too long or too short).

We can define the selection model of the list, this is, if we want to enable the selection of multiple elements, or if we only want to select one element. We can change this feature with these methods:

```
myList.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
myList.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);
```

If we want to get the selected item(s) of the list, we have to get the selection model of the list, and then call to **getSelectedItem** (for single selection lists) or **getSelectedItems** (for multiple selection lists). We also have the **getSelectedIndex** and **getSelectedIndexes**, if we want to get the position(s) of the selected item(s), instead of their values.

```
String element = myList.getSelectionModel().getSelectedItem();
```

-  Dropdown lists, where only one element is shown, and we can choose any other element by spreading out the list. If we want to use this type of lists in our JavaFX applications, we can choose between **ChoiceBox** and **ComboBox** classes. The differences between them are quite subtle, although *ComboBox* is more appropriate when we work with large lists.

Either if we work with *ChoiceBox* or *ComboBox*, we create the lists in a very similar way to the one shown for *ListView*: we need an *ObservableList* to set the elements, and then we create the list with them.

```
ObservableList<String> items = FXCollections.observableArrayList(
    "Windows", "Linux", "Mac OS X");
ChoiceBox myList = new ChoiceBox(items);
```

If we want to get the selected item of the list, we can use the **getValue** method (and the **setValue** method to set the selected item, if we want to).

There are some other ways of adding items to the lists, such as calling a *getItems* method and then call the *addAll* method to add more items:

```
myList.getItems().addAll("Android", "iOS");
```

2.2.7. 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 create a menu bar, define the categories for the menus, and add menu items to the categories. Let's see an example:

```
MenuBar bar = new MenuBar();

Menu menuFile = new Menu("File");
MenuItem menuItemNew = new MenuItem("New");
MenuItem menuItemOpen = new MenuItem("Open");
menuFile.getItems().addAll(menuItemNew, menuItemOpen);

Menu menuEdit = new Menu("Edit");
MenuItem menuItemUndo = new MenuItem("Undo");
MenuItem menuItemCopy = new MenuItem("Copy");
menuEdit.getItems().addAll(menuItemUndo, menuItemCopy);

menuBar.getMenus().addAll(menuFile, menuEdit);
```

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

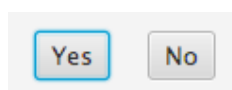
- **MenuBar** to define the bar where all menus and menu items will be placed
- **Menu** to define the categories for the menu items. A category is an item that can be displayed, but it has no action (if we click on it, nothing else happens but showing the items that this category contains).
- **MenuItem** 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 some typical menu items of almost every application, such as "New", "Open", "Copy"..
 - 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.

2.3. Organizing controls. The "layout" package

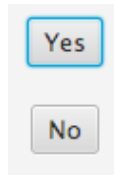
One of the most tedious tasks when designing a graphical application is to arrange the controls in the window. This task is carried out by the **layout manager** or **layout pane**. You are about to know that there are some of them, and depending on your choice(s), controls can be arranged in many different ways.

Some of the most common layout panes in JavaFX are:

- **HBox**: arranges controls horizontally, one next to the other.



- **VBox**: arranges controls vertically, one above the other.



- **FlowPane**: arranges controls next to each other until there is no more space. Then, it goes to next row (or column, depending on its configuration) to keep on arranging more controls.
- **BorderPane**: this layout divides the pane into five regions: top, bottom, left, right and center, and we can add a control (or a pane with some controls) in each region.
- 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.

http://docs.oracle.com/javase/8/javafx/layout-tutorial/builtin_layouts.htm

Let's see how to use these layout managers with some code examples.

2.3.1. Working with HBox and VBox panes

These layout panes arrange the controls horizontally/vertically. To use these panes, we call either a default constructor, or a constructor with a *double* argument, indicating the spacing between controls when we add more than one to the pane:

```
// We create an HBox with 20 pixels of spacing between controls
HBox hbox = new HBox(20);
// We define a default VBox
VBox vbox = new VBox();
```

Then, if we want to add controls to the pane, we call its *getChildren* method to access this pane's children place, and *add* or *addAll* methods to add a single control, or a list of them, respectively.

```
// We define some controls
Button btn1 = new Button("One");
Button btn2 = new Button("Two");
Label lbl1 = new Label("Label 1");
TextField txt1 = new TextField();

// Add the buttons to the vbox
vbox.getChildren().addAll(btn1, btn2);
// Add the label and text field to the hbox
hbox.getChildren().addAll(lbl1, txt1);
```

Controls are placed in the pane in the same order that they are placed in the *addAll* method.

2.3.2. Working with FlowPane

If we want to use a *FlowPane*, we can use several constructors, from a default (empty) constructor, to one that lets us specify the orientation (vertical or horizontal), and the horizontal and vertical spacing between controls.

```
// Horizontal FlowPane with 20px horizontal spacing and 30px vertical spacing
FlowPane fp = new FlowPane(20, 30);
```

```
// Vertical FlowPane with 20px horizontal spacing and 30 px vertical spacing
FlowPane fp2 = new FlowPane(Orientation.VERTICAL, 20, 30);
```

To add controls to the pane, we also call the *getChildren* method, and then the *addAll* method (just like *HBox* or *Vbox*):

```
fp.getChildren().add(btn1);
fp2.getChildren().addAll(btn2, txt1);
```

Controls are placed according to the orientation specified in the constructor (horizontal by default).

2.3.3. Working with *BorderPane*

When working with *BorderPanes*, we only have an empty constructor:

```
BorderPane bp = new BorderPane();
```

Then, to add controls to the pane, we call the corresponding method, depending on which of the five regions of the pane is the one where we want to place the control. For instance, if we want to add a button at the top and a label at the bottom (from the controls that we created in the example in section 2.3.1), we do it this way:

```
bp.setTop(btn1);
bp.setBottom(lbl1);
```

2.3.4. More options with panes

Once we have our layout pane ready, we can add it to the scene. We do it when we create the scene, passing the layout pane as a parameter:

```
Scene scene = new Scene (pane, 500, 400);
```

However, we can **combine multiple layout panes** to create a more complex one, before adding them to the scene. This example creates a *VBox* with a *FlowPane* in one of its rows, and then add the *VBox* to the scene:

```
VBox vbox = new VBox(20);
FlowPane fp = new FlowPane();

fp.getChildren().addAll(lbl1, txt1);
vbox.getChildren().addAll(btn1, fp, btn2);

Scene scene = new Scene(vbox);
```

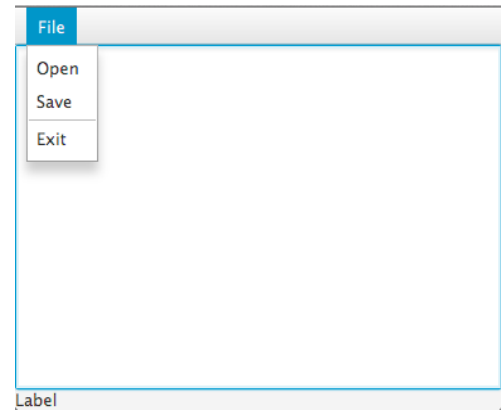
We can also set some **pane properties**, such as the control alignment, padding... For instance, these instructions align the controls to the center of a *VBox*, and set a padding of 10 pixels in every edge (top, left, bottom and right).

```
VBox vbox = new VBox();
...
vbox.setAlignment(Pos.CENTER);
vbox.setPadding(new Insets(10));
```

There are some other useful methods to control spacing, independent padding for each edge, and so on. Check the API for more details.

Exercise 1

Create a project called **NotepadJavaFX**. Use a *BorderPane*, and add a menu at the top, a *TextArea* at the center and a label at the bottom. The menu must have a *File* menu with four menu items: *Open*, *Save*, a separator item and *Exit*. Set a scene of 400 pixels width and 300 pixels height. It should look like the image on the right.



Exercise 2

Create a project called **Calculator** with this appearance (you can use the layout manager(s) that you consider to get it):

A screenshot of a calculator application window. The window has a light gray background. It contains two text input fields, one labeled '1st number:' and one labeled '2nd number:'. Between these two fields is a dropdown menu labeled 'Choose operation:'. Below the '2nd number:' field is a button labeled 'Go'. At the bottom of the window is a text input field labeled 'Result:'.

The dropdown list can be done either with a *ChoiceBox* or a *ComboBox*. It must have the options "+", "-", "*" and "/" (the four basic arithmetic operators).

3. Events

If we only add controls to our JavaFX application (buttons, labels, text boxes...) 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 an object (sometimes a simple 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 values of some text boxes, adds them all and shows the result.

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...)
- **TouchEvent**: created when the user touches something in the app (on devices that allow touch input)
- **WindowEvent**: created when the status of the window changes (for instance, it is maximized, minimized, or closed).

3.2. Defining handlers and connecting them with events

Now that we know what an event is, and its main subtypes, let's see how to define handlers to control them. As you will see, there are many ways of defining handlers, and you can choose any of them depending on your application features.

3.2.1. Defining handlers by implementing the *EventHandler* interface

The first way of defining handlers that we are going to see is by implementing the **EventHandler** interface. This is a parameterized interface, and we need to indicate which type of event we are going to deal with.

We can do this in our main class, if we want. For instance, here we define an event handler to respond to action events (button or menu item clicks):

```
public class MyJavaFXApplication
    extends Application implements EventHandler<ActionEvent>
```

We can also create an inner (or outer) class that implements the interface:

```
class MyHandler implements EventHandler<ActionEvent>
```

In both cases, we need to define a **handle** method to fulfill the *EventHandler* requirements.

```

@Override
public void handle(ActionEvent e)
{
    ... // Code to respond to the event
}

```

Once we have defined the handler, we need to connect it to the event source (i.e. the control that created the event, in this case, a button or menu item from our application). This can be typically done in the *start* method.

```

public class MyFirstJavaFXApplication
    extends Application implements EventHandler<ActionEvent>
{
    Button btnTest;

    @Override
    public void start(Stage primaryStage)
    {
        btnTest = new Button("Hello");
        btnTest.setOnAction(this);
        ...
        primaryStage.show();
    }

    @Override
    public void handle(ActionEvent e)
    {
        ... // Code to respond to the event
    }

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

```

As you will learn, every control (button, text box, label...) has its own methods to connect it to an event handler. In the case of buttons or menu items, we can use the ***setOnAction*** method to connect them to an *ActionEvent* handler.

If we had used an inner or outer class, we only have to replace the *this* parameter of *setOnAction* with the object that implements the *EventHandler* interface:

```

class MyHandler implements EventHandler<ActionEvent>
{
    @Override
    public void handle(ActionEvent e)
    {
        ... // Code to respond to the event
    }
}

public class MyFirstJavaFXApplication extends Application
{
    Button btnTest;

    @Override
    public void start(Stage primaryStage)
    {
        MyHandler handler = new MyHandler();
        btnTest = new Button("Hello");
        btnTest.setOnAction(handler);
    }
}

```

```

        ...
        primaryStage.show();
    }

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

```

One of the problems of using this method is that, if we need to handle different types of events, we need to implement a generic event handler from *Event* class:

```
public class MyApplication extends Application implements EventHandler<Event>
```

And then, in our *handle* method, we need to distinguish the type of event that has been launched:

```

@Override
public void handle(Event e)
{
    if (e instanceof ActionEvent)
    {
        ...
    } else if (e instanceof MouseEvent) {
        ...
    }
    ...
}

```

3.2.2. Defining event handlers through anonymous classes

A second option to define our event handlers is to create an anonymous class everytime we need to define a handler. You should have heard of anonymous classes in Unit 1; we create an object of a given class (even interfaces or abstract classes) just when we need it. Regarding events, we create the anonymous class when we call the event source's method to connect to an event handler. In our previous example, we would create the anonymous class when we call the *setOnAction* method from our button object:

```

public class MyFirstJavaFXApplication extends Application
{
    Button btnTest;

    @Override
    public void start(Stage primaryStage)
    {
        btnTest = new Button("Hello");
        btnTest.setOnAction(
            new EventHandler<ActionEvent>()
            {
                @Override
                public void handle(ActionEvent e)
                {
                    ... // Code to respond to the event
                }
            });
        ...
        primaryStage.show();
    }
    ...
}

```


This way, we do not need to implement any interface, we are creating some kind of *EventHandler* object (although *EventHandler* is not a class, but an interface), and overriding its *handle* method.

Do not worry if this way of defining event handlers is not clear for now, we will use it in some examples later.

3.2.3. Defining event handlers through lambda expressions

A third way of defining event handlers is by using lambda expressions (you also heard of them in Unit 1). In this case, we also add the code when we call the connection method from our event source. In our example, we add it when calling the *setOnAction* method:

```
public class MyFirstJavaFXApplication extends Application
{
    Button btnTest;

    @Override
    public void start(Stage primaryStage)
    {
        MyHandler handler = new MyHandler();
        btnTest = new Button("Hello");
        btnTest.setOnAction( e ->
            {
                ... // Code to respond to the event
            });
        ...
        primaryStage.show();
    }
    ...
}
```

The mechanism is very similar to using an anonymous class, but in this case we do not need to create any object. As soon as we click on the button, the lambda expression is fired, and its code is executed.

If we only need to call a single method inside the lambda expression, we just write it next to the arrow:

```
btnTest.setOnAction( e -> myMethod() );
```

As with previous mechanism, do not hesitate if you do not understand how this can work. We will use all these ways of defining handlers in some examples along this unit.

3.3. Examples

3.3.1. ActionEvent to change a button's text

This application has a button in the middle of it, and the button text changes every time we click on it:

```
import javafx.application.Application;
import javafx.event.*;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class ExampleActionEvent
extends Application implements EventHandler<ActionEvent>
{
    Button btnHelloGoodbye;
```

```

// Application main function
@Override
public void start(Stage primaryStage)
{
    // Create the button and define its event handler
    btnHelloGoodbye = new Button("Hello");
    btnHelloGoodbye.setOnAction(this);

    // Add the button to a layout manager and scene
    BorderPane pane = new BorderPane();
    pane.setCenter(btnHelloGoodbye);
    Scene scene = new Scene(pane, 500, 400);
    primaryStage.setScene(scene);

    // Show window
    primaryStage.show();
}

// Event handler
@Override
public void handle(ActionEvent e)
{
    if (e.getSource() == btnHelloGoodbye)
    {
        // Swap "Hello" and "Goodbye" texts
        if (btnHelloGoodbye.getText() == "Hello")
            btnHelloGoodbye.setText("Goodbye");
        else
            btnHelloGoodbye.setText("Hello");
    }
}

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

```

We have defined an event handler by implementing the *EventHandler* interface in main class. In the *start* method, we just create the button, the layout manager, the scene, and put it all together. We also define an event handler for the button, through its *setOnAction* method. Regarding this handler, we only check if the event source is our expected button (in this example no one else could fire the event handler, but in other applications with more than one button, we will need to distinguish which button has been clicked). Then, we swap the button's text for its contrary ("Hello" / "Goodbye").

3.3.2. MouseEvent to change a label's background color

This example adds a label in the middle of the window, and if we enter the mouse on it, it changes its background color to red, whereas if we move the mouse out of the label, it recovers its original background color.

```

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

```

```

public class ExampleMouseEvent extends Application
{
    Label lblColor;

    // Application main function
    @Override
    public void start(Stage primaryStage)
    {
        // Create the label and define its event handlers
        lblColor = new Label("Change my background color!");

        // Handler for entering the label (change color to green)
        lblColor.setOnMouseEntered(
            new EventHandler<MouseEvent>()
            {
                public void handle(MouseEvent e)
                {
                    lblColor.setStyle("-fx-background-color:green;");
                }
            }
        );

        // Handler for exiting the label (set color to none)
        lblColor.setOnMouseExited(
            new EventHandler<MouseEvent>()
            {
                public void handle(MouseEvent e)
                {
                    lblColor.setStyle("-fx-background-color:none;");
                }
            }
        );

        // Add the label to a layout manager and scene
        BorderPane pane = new BorderPane();
        pane.setCenter(lblColor);
        Scene scene = new Scene(pane, 500, 400);
        primaryStage.setScene(scene);

        // Show window
        primaryStage.show();
    }

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

```

Notice that, in this case, we have defined the event handlers by using anonymous classes. We have needed two event handlers, one for mouse entering the label (it changes the background color to green), and another one for mouse exiting the label (it sets no background color). See how we use some kind of CSS style to change the background color. We will talk about this later in this unit.

3.3.3. KeyEvent to clone a text field into a label

This example uses a *KeyEvent* to capture every key pressed inside a text field, and copy its contents to a label.

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class ExampleKeyEvent extends Application
{
    Label lblCopy;
    TextField txtOriginal;

    // Application main function
    @Override
    public void start(Stage primaryStage)
    {
        // Create the label and the text field
        lblCopy = new Label("");
        txtOriginal = new TextField();

        // Handler for copying the text from text field to the label
        txtOriginal.setOnKeyTyped( e ->
        {
            lblCopy.setText(txtOriginal.getText());
        }
        );

        // Add the text field and the label to a layout manager and scene
        BorderPane pane = new BorderPane();
        pane.setCenter(txtOriginal);
        pane.setBottom(lblCopy);
        Scene scene = new Scene(pane, 500, 400);
        primaryStage.setScene(scene);

        // Show window
        primaryStage.show();
    }

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

```

In this example, we have used lambda expressions to define the handler. When we type anything in the text field, the *setOnKeyTyped* method will be fired, and thus the lambda function will be executed. It just copies the text field's current text to the label.

Exercise 3

Create a project called **CalculatorEvent** that will be a copy of previous project *Calculator* from *Exercise 2*. Add an *ActionEvent* to the button so that, when it is clicked, it gets the numbers typed in the first two text fields, and the type of operation from the choice box, and then prints the result in the last text field. For instance, if the *1st number* is 2, the *2nd number* is 3 and the operation is "*", the result should show 6.

If no numbers are typed in 1st or 2nd text fields, assume that there is a 0 on them.

Exercise 4

Create a project called **CurrencyConverter** that allows us to convert between three different types of currencies: euro (EUR), dollar (USD) and british pound (GBP). There will be a menu to choose one of the six possible combinations, by using *RadioMenuItem*s: EUR>USD (default option), EUR>GBP, USD>EUR, USD>GBP, GBP>EUR and GBP>USD. Below that, there will be a text field, and a label. Everytime we type anything in the text field, the program must convert the amount automatically to the given currency, and show the result in the label. For instance, if we have chosen EUR>GBP, and we know that 1 EUR = 0.8 GBP, then when we type "12" in the text field, the program should look like this:

To complete the application, add an *ActionEvent* to each *RadioButtonItem* that clears the text in the text field and label, to start a new conversion with new currencies.

In order to help you finish the program, assume that the currency exchanges are as follows:

- 1 EUR = 1.10 USD
- 1 EUR = 0.8 GBP
- 1 USD = 0.7 GBP

4. More about stages and layouts

We have covered some of the basic concepts of creating a JavaFX application. But there are some more options that are commonly used, and we are going to talk about them in this section.

4.1. Adding images to our application

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 need to 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.

We can add an image directly as any other control:

```
Image img = new Image("photo.jpg");
ImageView view = new ImageView(img);
```

```
BorderPane pane = new BorderPane();
pane.setCenter(view);
```

Or we can add the images in some controls that have specific constructors to add images to them, such as Labels or Buttons:

```
Image img = new Image("photo.jpg");
ImageView view = new ImageView(img);
Label lbl = new Label("Label text", view);
```

In both cases, we load the image with the **Image** class, and we add it to the application with the **ImageViewer** class.

If you want to keep the image in your project ready to be loaded, it is very important that you put it in a source folder instead of a "normal" folder. So you can add your images and Java classes to the same source, or you can create an additional source folder (for instance, a folder called *resources*), and put on it all the additional resources that your program needs (images, text files...).

4.2. Common control properties: borders and colors

Although every control that we add to our application comes with its default colors and lines, we can specify both the text and background color of the control, and also its border type. You will see later that we can define all these styles from a CSS file as well.

4.2.1. Borders

Every border of a control is composed of one or more strokes. We control the border style of the controls with the **setBorder** method and the **Border** class. To define the style of each stroke, we have the **BorderStroke** class. It has some constructors, and we can define the border color, style (solid, dotted...), corner radius, width... of the stroke(s).

The following example creates a red, solid border on a label, with 10px of corner radius and 5px width.

```
Label lbl = new Label("Hello");
lbl.setBorder(new Border(new BorderStroke(Color.RED,
```



```
BorderStrokeStyle.SOLID, new CornerRadii(10), new BorderWidths(5))));
```

4.2.2. Colors

We can define either the background color (or image) of a control, or its text color. If we want to change the background, every control has a **setBackground** method, where we can define the type of background that we want (a color, an image...). For instance, if we want to apply a background color, we use the **BackgroundFill** class, and in its constructor we specify the background color, the corner radius and the insets (internal padding).

The following example creates a yellow background color for the label of previous example, with a border radius of 10px and insets of 10px as well.

```
lbl.setBackground(new Background(new BackgroundFill(Color.YELLOW,  
    new CornerRadii(10), new Insets(10))));
```



Regarding the text color, we have the **setTextFill** method, to specify the desired color:

```
lbl.setTextFill(Color.GREEN);
```

4.3. Dialog boxes

You have probably used some desktop applications on which, when you try to click a button, or select a menu option, an error message appears, or if you want to save your work, a small window that lets you choose the folder and file name is shown. These "small windows" that are shown from the main application are known as **dialog boxes**.

In some other libraries, such as Swing, there were some built-in classes that made it easy to create these type of dialogs. For instance, we could use the *JOptionPane* class to display an error or confirmation message.



This type of built-in classes are not present in JavaFX (apart from some specific dialogs that we will see later), and we need to define our own stage, scene and controls to represent these dialogs.

4.3.1. Basic dialog boxes

If we want to create a basic dialog box to show an information or error message, we create a class like this one:

```
public class MyDialog  
{  
    public static void show(String message)  
    {  
        Stage stage = new Stage();  
        stage.initModality(Modality.APPLICATION_MODAL);  
  
        Label lblMessage = new Label(message);  
        Button btnOk = new Button("OK");  
        btnOk.setOnAction( e ->
```

```

        {
            stage.close();
        });

        VBox pane = new VBox(50);
        pane.getChildren().addAll(lblMessage, btnOk);
        pane.setAlignment(Pos.CENTER);

        Scene scene = new Scene(pane);
        stage.setScene(scene);
        stage.showAndWait();
    }
}

```

We have created a class with a static method called *show*. Inside this method, we define a stage, a label, a button, a layout manager and a scene, and we add the corresponding instructions to build a window with all these items. Notice that we have used a method from the *Stage* class called *initModality*. When we launch a dialog box, we can't usually go back to our main application unless we close the dialog. This is what we call a **modal** window (a window that opens from another window and stays on top of the application until it is closed). Anyway, we can also create non-modal dialog boxes by changing the parameter of the *initModality* method.

We also define an event handler for the "OK" button, so that when we click on it, the dialog closes and we can go back to our main application.

If we want to show this dialog when we click a button from our main application, we just show it from the corresponding *setOnAction* command:

```

public class MyApplication extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        Button btnDialog = new Button ("Click me to show the dialog");
        btnDialog.setOnAction( e ->
        {
            MyDialog.show("You have clicked the dialog button!!");
        });
    }
}

```

Exercise 5

Create a project called **OKErrorDialog**, and define your own dialog class called *OkErrorDialog* to show a customizable message (passed as a parameter). The dialog will also show an "OK" or "Error" icon, depending on the type of message that we want to show (an error message, or just an information message).

Define your main application called *EmptyFieldTest*, with a text field and a button. If we click on the button leaving the text field empty, we should show an error message saying "*The text field can't be empty*", and if we click the button with anything written in the text field, then it will show an information message saying "*Everything is OK*".

4.3.2. Confirmation dialog boxes

What if we do not want to just show a message to be read by the user? What if we want to make him choose between "Yes" or "No"? In this case, we need to create a confirmation dialog box, with a label and two or more buttons to make the user choose one of them.

The *show* method that we did for our basic dialog should now return a boolean value, depending on the option chosen by the user ("Yes" = *true*, "No" = *false*).

```
public class MyConfirmationDialog
{
    boolean result;

    public static boolean show(String message)
    {
        result = false;

        Stage stage = new Stage();
        stage.initModality(Modality.APPLICATION_MODAL);

        Label lblMessage = new Label(message);

        Button btnYes = new Button("Yes");
        btnYes.setOnAction(e ->
        {
            result = true;
            stage.close();
        });

        Button btnNo = new Button("No");
        btnNo.setOnAction(e ->
        {
            result = false;
            stage.close();
        });

        HBox paneButtons = new HBox(20);
        paneButtons.getChildren().addAll(btnYes, btnNo);

        VBox pane = new VBox(50);
        pane.getChildren().addAll(lblMessage, paneButtons);
        pane.setAlignment(Pos.CENTER);

        Scene scene = new Scene(pane);
        stage.setScene(scene);
        stage.showAndWait();

        return result;
    }
}
```

We have defined a boolean attribute to store the user choice. If it clicks the "Yes" button, it is set to *true*, otherwise it is set to *false* (its default value is also *false*). This way, if we close the dialog by clicking its close button in the upper bar, the dialog will return *false*, as its default value.

4.3.3. Closing dialogs and windows

We have just dealt with a very common problem. If the user tries to close the application window with the close button of the upper bar (or typing *Alt+F4* or any other shortcut), our efforts to try to save the data before closing, or get a confirmation from the user before exiting the program will be skipped.

But do not worry. There is a way of catching this event and redirect it to our control methods. In previous dialog, for instance, if we want to catch the event when the user tries

to close the window, we call the `setOnCloseRequest` method from the `Stage` object, and then we can simply do the same as if the user clicked the "No" button:

```
stage.setOnCloseRequest(e ->
{
    result = false;
});
```

In this case, it is not necessary to close the stage, since, when the user clicks the close button, the closing process goes on unless we avoid it. And... how to avoid the inexorable closing? We can use the `consume` method from the event (`e`). This is a very useful method, that consumes the upcoming event instead of letting it go on. If we changed previous event handling for this one:

```
stage.setOnCloseRequest(e ->
{
    e.consume();
    result = false;
});
```

then our dialog would not close when we click on the close button. It would only close by clicking the "Yes" or "No" buttons.

Exercise 6

Create a project called **TotalConfirmationDialog**. Define a main application with a text field and a button with the text "Close". If we have typed anything in the text field, and we try to close the application (either by clicking on the "Close" button, or trying to close the application in any other way), a confirmation dialog must be shown asking us to confirm our action ("Are you sure you want to exit?"). If we click the "Yes" button, the application will close, otherwise it will remain visible.

4.3.4. Built-in dialog boxes: FileChooser

There are, however, some built-in dialog boxes in JavaFX focused on some specific tasks (that are very difficult to be developed manually). In particular, we talk about 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).

Exercise 7

Create a project called **NotepadJavaFXEvent**, that will be a copy of project *NotepadJavaFX* done in *Exercise 1*. We are going to add some events to that project so that we will have a completely functional notepad:

- Add an action event to the *File > Open* menu that shows a *FileChooser* dialog to open a file, and load its contents into the text area.
- Add an action event to the *File > Save* menu that shows a *FileChooser* dialog to save the text in the text area into a chosen file.
- Add events and methods to control the window closing, either from the button of its upper bar, or from the *File > Exit* menu. If anything has been typed in the text area

after the last open or save operation, a confirmation dialog must be shown before closing (as any text processor does).

Use the label at the bottom of the window to show some information messages. For instance:

- When the user selects a file to open, we could set the label text to "File XXXXX opened".
- When the user selects a file to save, we could set the label text to "File XXXXX saved".
- When the user types something in the text area, we could set the text to "Text changed".

Exercise 8

Create a project called **MySimpleWizard** that simulates an installation or configuration wizard. It will guide the user throughout 3 steps before finalizing.

The **first step** will show a text field and will ask the user to enter his e-mail. If he clicks on the *Next* button without writing anything, an error message will be shown. Otherwise, we will go to step 2.

The **second step** will show two radio buttons, to let the user choose between typical installation (default option) or custom installation. From this point, the user can go back to first step by clicking the *Back* button, or he can go to the last step by clicking the *Next* button.

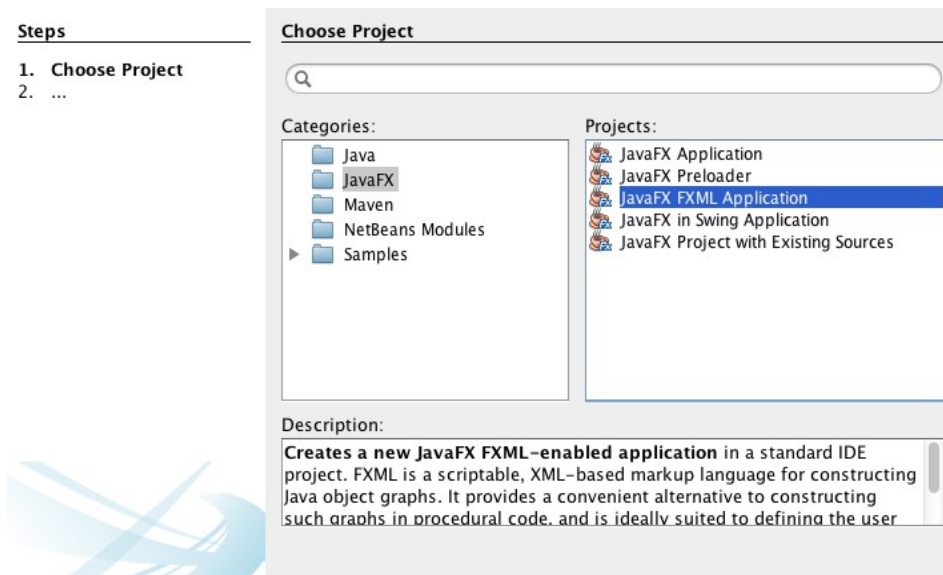
The **third and last step** will show a *ListView* with some hobbies or interests (for instance, *Computers, Videogames, Travels, Books, Photography*), and will ask the user to choose as many as he wants. From this point, the user can also go back with the *Back* button (to step number 2), or finish the wizard with the *Next* button.

The **final scene** will show a summary of all the things that the user has typed and chosen along the wizard. There will be a *Close* button to finish the application.

HELP: To do this exercise, you should create four different Scenes, each one with the corresponding controls. The stage will change the scene each time the user clicks on a Next or Back button from the wizard, by calling its `setScene` method.

4.4. Defining controls and layout through XML files

Previously in this unit, you were told that there was an alternative way of creating user interfaces, without typing any Java code. This alternative way uses XML files with a concrete syntax to specify the layout and controls of our application. From NetBeans, it is very easy to create such type of applications. Just go to *File > New Project*, and then select the *JavaFX* folder and the *JavaFX FXML Application* option.



Notice that there is also a *JavaFX Application* option in this dialog. If we choose it, the basic structure of a JavaFX application (main class, imports...) is created, but we have been creating all this code manually so far, so that you become familiar with it.

Once we have our project created, you will see three files in it:

- A n **FXML** file that will have the graphical structure of our application (layouts, controls, texts...).
- A Java file with suffix **Controller**, that will handle all the events produced in the controls specified in the FXML file.
- A main **JavaFX application**. If you take a look at its code, it just loads the controls from the FXML file, and shows the stage with the loaded scene.

Let's have a look now at the **FXML** file. Its initial structure is something like this:

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

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320" xmlns:... >
  <children>
    <Button layoutX="126" layoutY="90" text="Click Me!"
      onAction="#handleButtonAction" fx:id="button" />
    <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69"
      fx:id="label" />
  </children>
</AnchorPane>
```

1. At the beginning of the FXML there is an *?xml* header, indicating the XML version and the file encoding.
2. Then, we add some imports to the file, so that we can use the classes inside these imports.

3. After these lines, there is a layout pane (of type *AnchorPane*; we have not seen this class, but we can use any other layout pane instead of this one).
4. Inside this main layout pane, there is a *children* tag, with all the content managed by this layout pane. Inside this *children* tag we can add simple controls (buttons, labels...), or more layout panes with their own children.

This FXML file is loaded from **main application**, with this instruction:

```
Parent root = FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));
```

Then, we add this *root* node to a scene, and create the main stage with this scene.

Finally, we have the **controller**, where we will put the event handlers associated to the controls of our FXML file. In the default example, there is an *ActionEvent* associated to the button. In the FXML file, we define the event handler this way, inside the *Button* tag:

```
<Button layoutX="126" layoutY="90" text="Click Me!"  
  onAction="#handleButtonAction" fx:id="button" />
```

Then, in the controller, we define the code for this event this way:

```
@FXML  
private Label label;  
  
@FXML  
private void handleButtonAction(ActionEvent event)  
{  
    System.out.println("You clicked me!");  
    label.setText("Hello World!");  
}
```

We use the *@FXML* annotation to refer to things that have been declared in the FXML file. So in the controller, we have added a private attribute that is a reference to the label in the FXML file, so that we can use this label inside the event handler, to change its text. Notice that the event handler name (*handleButtonAction*) and the label *id* (*label*) are the same in the FXML file and in the controller.

http://docs.oracle.com/javase/8/javafx/api/javafx/fxml/doc-files/introduction_to_fxml.html

<https://docs.oracle.com/javase/8/javafx/fxml-tutorial/>

Exercise 9

Create a project called **CalculatorFXML** and try to get the same result than the one in exercise 3 (project *CalculatorEvent*), by using an FXML file and its corresponding controller.

HELP: populate the ChoiceBox of operations in the "initialize" method of the controller, instead of trying to populate it in the FXML file. Normally you will need to read a file or a database to populate such type of lists, and this can't be done in the FXML file.

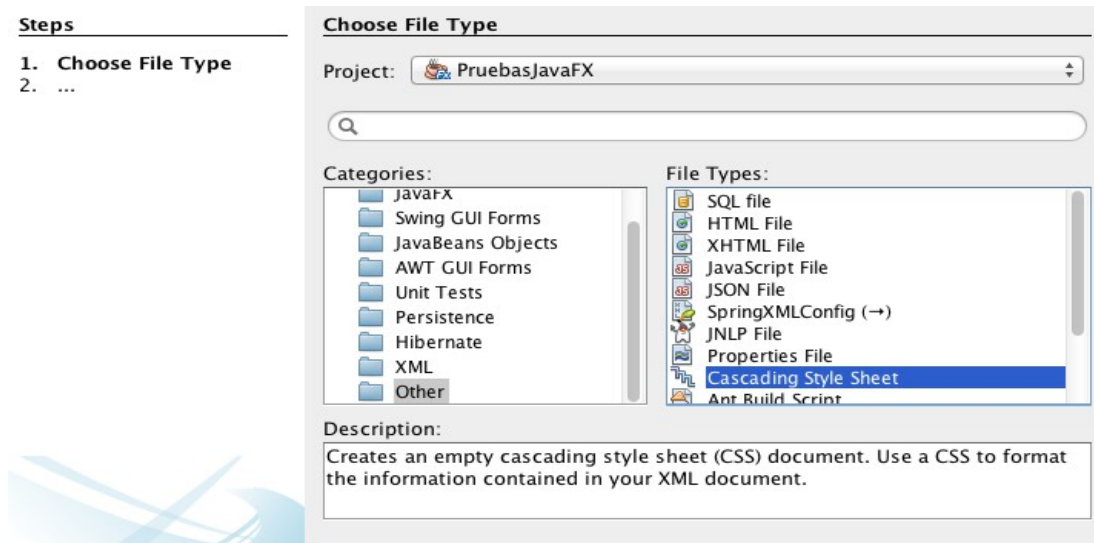
4.5. Changing the default appearance with CSS

If you look at any of the applications that we have created up to now, all of them have an old fashioned style, similar to the applications that we used in former operating systems, such as Windows 95.

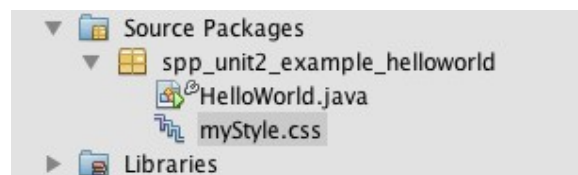
But there is a way to change this default appearance for a modern one, by using **CSS** files, a file type very used in web design. We are not going to go into CSS syntax much in

depth, since we assume that you have an idea of what it is and how it works. We are going to see how to add these files to our JavaFX applications.

First of all, we need to add a CSS file to our project. To do this, go to *File > New File*, and select the *Other* folder, and then the *Cascading Style Sheet* option.



Give it an appropriate name, and then this file will be created inside your source folder.



Then, we can add the styles to our CSS file. There is a special naming that we need to follow in order to set the styles to the appropriate controls. For instance, if we want to apply general styles to all the elements of the application (such as background color, font styles...) we use the *root* class selector:

```
.root
{
    -fx-background-color: lightgray;
    -fx-font-family: "Arial";
}
```

Notice that every CSS style begins with the *-fx-* prefix, followed by the typical CSS style name (*background-color, font-family...*). There are also some other class selectors that we can use to apply styles to some specific controls (buttons, labels...). For instance, this style is applied to every button of our application:

```
.button
{
    -fx-background-color: #A22E15;
    -fx-color: white;
}
```

We can also define our own class or id selectors, such as

```
.myButton { ... }
#myID { ... }
```

Then, if we want to apply these styles to some selected controls, we need to assign them the corresponding class or id, from the Java code:

```
Button btn = new Button("Click me");
Label lbl = new Label("Text label");
btn.getStyleClass().add("myButton");
lbl.setId("myID");
```

Once we have our CSS file created, we can add it to a scene with this instruction (as long as the CSS is placed in a source folder to be copied to the build destination directory):

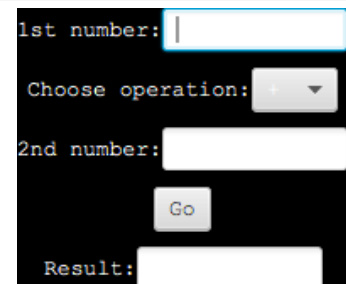
```
scene.getStylesheets().add("myStyle.css");
```

Anyway, we can also apply the styles to some elements directly in the Java code, with the **setStyle** method:

```
lbl.setStyle("-fx-background-color:green;");
```

Exercise 10

Create a project called **CalculatorEventCSS**, that will be a copy of project *CalculatorEvent* from exercise 3. Change its default style by using a CSS file that sets the background color of the whole application to black, the text color (for the labels) to white, and the font family to *Courier New*. At the end, your application should look like this:



HELP: Place the CSS file in a source folder (it can be the default source folder), so that it will be copied together with the classes. To change the text color, use the "-fx-text-fill" property in the .label selector.

4.5.1. Combining FXML and CSS

If we want to apply a stylesheet to a FXML file, we need to add a *stylesheets* tag inside the element whose content is affected by this CSS. For instance:

```
<BorderPane...>
    ...
    <stylesheets>
        <String fx:value="myStyle.css" />
    </stylesheets>
</BorderPane>
```

Then, if we want to apply some class or *id* styles to some controls, we use the *styleClass* or *id* attributes in their tags. For instance, if our CSS has a style called *.special* for all members of class *special*, and we want to apply it to a button, we do it like this:

```
<Button ..... styleClass="special" ..... />
```

If we want to apply an *id* style to an element with ID *#main*, we do it like this:

```
<Label ..... id="main" ..... />
```

Exercise 11

Create a project called **CalculatorFXMLCSS**, that will be a copy of project *CalculatorFXML* from exercise 9. Apply the same CSS style of exercise 10 to this project, and check if you get the same result.

5. Some advanced features

In this final section we are going to cover some non-standard features that JavaFX provides, such as some special effects to apply to our programs (shadows, lighting...) and some animation effects.

5.1. Special effects

In this subsection we are going to have an overview of some of the most interesting effects that we can apply to our JavaFX programs. These effects are the reason of the *FX* suffix in *JavaFX*, and most of them are subclasses of the *Effect* class. The general pattern to work with them is:

1. Create an instance of the effect's class that we want to use
2. Set the properties of the effect (colors, lengths... depending on the chosen effect)
3. Apply the effect to a node or control. When we apply an effect to a control, it is automatically applied to all of its children (the controls contained in this control or node). To apply the effect, every node/control has a **setEffect** method, as we will see in the following examples.

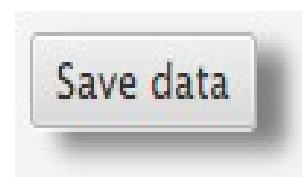
5.1.1. Shadows

To apply shadows to objects in our application, we use the **DropShadow** class (if we want a shadow outside the chosen node or control) or **InnerShadow** (for shadows inside the chosen node or control). Once we create the object (with a default constructor), there are some useful methods to set some properties, such as:

- **setColor**: sets the color of the shadow
- **setWidth, setHeight**: set the width and height of the shadow, respectively
- **setOffsetX, setOffsetY**: set the horizontal and vertical offset for the shadow (if we want it to stand out of the limits of the control/node (default is 0))
- **setRadius**: radius of the shadow's blur effect (default is 10)

Here is a code that creates a shadow, sets its color and offset, and applies it to a button.

```
Button btn = new Button("Save data");
DropShadow shadow = new DropShadow();
shadow.setColor(Color.GRAY);
shadow.setOffsetX(10);
shadow.setOffsetY(5);
btn.setEffect(shadow);
```



5.1.2. Lighting

There are different types of light that we can add to our application:

- **Light.Point**: is a light source that starts from a fixed point in space, and radiates light equally in all directions
- **Light.Distant**: is a light source with no given start point, it seems to come from all directions.

For instance, if we want to add a point light, we create it (we can specify a given light color), we move it, and we add it to a **Lighting** object. Then we set this object as the effect to our control/node.

The following example adds a yellow point light, moves it and applies it to a button:

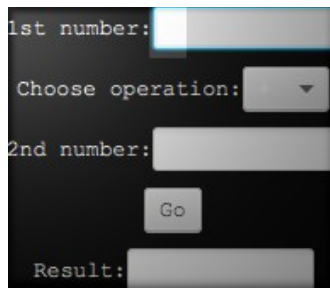
```
Light.Point pl = new Light.Point();
pl.setColor(Color.YELLOW);
pl.setX(10);
pl.setY(20);
pl.setZ(50);
Lighting light = new Lighting();
light.setLight(pl);
Button btn = new Button("Save data");
btn.setEffect(light);
```



Notice that the coordinates for the point light are relative to the control where it is applied (X = 10, Y = 20 for a button of W=200, H = 100, in previous example).

Exercise 12

Create a project called **CalculatorEventCSSLight** that will be a copy of previous project **CalculatorEventCSS** from *Exercise 10*. Add a white point light at coordinates X = 0, Y = 0, Z = 150, and see the final result. It should look like this:



5.1.3. Blur, glow and other effects

There are some more effects that you can use in your applications, but we will not cover them all in this unit, because we just want you to have an overview of what you can do with JavaFX. Some of the most useful additional effects that you can apply are:

- Make objects blurry with the **Blurry** effect. Just create it from one of its subtypes (*GaussianBlur*, *BoxBlur*, *MotionBlur*), set its properties (radius, angle... depending on the chosen subtype), and apply it to a node or control. This example applies a gaussian blur to a Label:

```
GaussianBlur blur = new GaussianBlur();
blur.setRadius(5);
Label lbl = new Label("First name");
lbl.setEffect(blur);
```



- Make controls glow with the **Glow** effect. Create it, set the glow level (from 0 to 1) and apply it to a node or control. This example applies a glow of 0.8 to a *RadioButton* (the one on the left):

```
Glow gl = new Glow();
gl.setLevel(0.8);
rbl.setEffect(gl);
```



- Other effects are available, such as adding perspective to our 2D controls to make them look like three-dimensional.

5.2. Animations

JavaFX also provides some ways to animate our controls or nodes. Basically, what we do is to change the properties of the nodes (size, position, color...) at regular intervals. In this process, there are two important factors:

- How much do we change the node properties. For instance, if we are moving a button, it will move faster if we increase its X position from 3 to 3 pixels than if we increase it from 1 to 1 pixel.
- The frequency of these changes. If we change the position of the button every 20 milliseconds, it will move slower than if we change it every 10 milliseconds.

There are two ways of making an animation: one of them is totally manual, and the other is (almost) totally automatic.

5.2.1. Automatic animations

The easiest way to create an animation is by using transition classes. There are some types of transition classes, such as ***TranslateTransition*** (to move elements), ***RotateTransition*** (to rotate elements), ***FadeTransition*** (to change the opacity)... All these transitions have some useful methods, such as:

- ***play / playFromStart***: plays the animation from its current position or from the start, respectively
- ***stop / pause***: stops or pauses the animation, respectively.
- ***setCycleCount***: sets how many times should the animation be repeated. If we want to repeat it indefinitely, we can use the *Timeline.INDEFINITE* constant.
- ***setAutoReverse***: sets if the animation should reverse every time it finishes.
- ***setInterpolator***: sets how to calculate the intermediate positions of the animation. Some of its possible values are *Interpolator.LINEAR* (always at the same speed), *Interpolator.EASE_IN* (faster at the beginning), *Interpolator.EASE_OUT* (faster at the end), *Interpolator.EASE_BOTH* (default value, it means faster at the beginning and at the end)...

Apart from these common methods, there are also some particular methods that belong to each concrete type of transition. For instance, if we use a *FadeTransition*, we will have to set the initial value of the opacity, and the final value. If we use a *RotateTransition*, we will have to specify the initial and the final angles.

The following example moves a label from the left of the window to the right (assuming a window of 300px width), in 3 seconds.

```
Label lbl = new Label("Hello");
FlowPane pane = new FlowPane();
pane.getChildren().add(lbl);

TranslateTransition t = new TranslateTransition(Duration.millis(3000), lbl);
t.setFromX(0);
t.setFromY(0);
t.setToX(300);
t.setToY(0);
```

```
t.play();
```

In the transition constructor, we specify the duration of the transition (3 seconds), and the node or control to which it will be applied (the label). Then, we set the initial and final values (position) of the node (label), and the *TranslateTransition* object takes the control. In this example, we could have omitted the lines referring to Y coordinate (*t.setFromY* and *t.setToY* methods), since we are moving the label horizontally.

We can also combine some transitions to run one after the other by using the **SequentialTransition** class. We add as many transitions as we need in its constructor, and then call its *play* method.

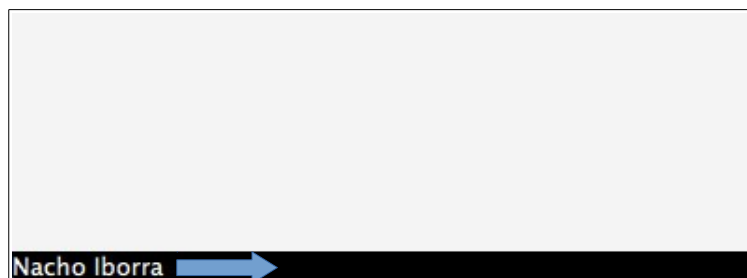
```
SequentialTransition s = new SequentialTransition(t1, t2, t3, t4);  
s.play();
```

If we need to run some transitions in parallel (simultaneously), we use the **ParallelTransition** class, in the same way that *SequentialTransition*:

```
ParallelTransition p = new ParallelTransition(t1, t2);  
p.play();
```

Exercise 13

Create a project named **AnimationCredits**. The Window will have a black *FlowPane* at its bottom (use a *BorderPane*), and inside this *FlowPane* there will be a label with white text with your name. This text must be continuously coming from the left side of the window and hiding at the right side, like a ticker.



5.2.2. Manual animations

If we prefer to create an animation in a more manual way, we need to work with these classes:

- **KeyFrame**, a timing interval that launches an *ActionEvent* every time the interval expires. We need to specify the duration of the interval and the event handler that will control the *ActionEvent*.
- **Timeline**, a sequence of *KeyFrames*. When we call it, each *KeyFrame* is played in sequence

If we wanted to create the same animation as in previous example with these classes, we would have the following code:

```
Label lbl = new Label("Hello");  
FlowPane pane = new FlowPane();  
pane.getChildren().add(lbl);  
  
KeyFrame kf = new KeyFrame(Duration.millis(10), e ->  
{  
    lbl.setTranslateX(lbl.getTranslateX() + 1);  
});
```

```
Timeline t = new Timeline(kf);
t.setCycleCount(300);
t.play();
```

First of all, we define the label and add it to the layout pane, as in previous example. Then, we define the *KeyFrame*. It will be activated every 10 milliseconds, and then it will increase the label X coordinate in 1 pixel. Finally, we add the *KeyFrame* to a *Timeline*, and set the cycle count. This value must be calculated manually. In our example, if we want to move the label from X = 0 to X = 300 in 3 seconds, moving it 1 pixel every 10 milliseconds, then we need a total of 300 iterations to get to X = 300.

Exercise 14

Create a project called **GrowingButton** with a button in the middle (use a *BorderPane*), with a preferred width of 100 pixels and a preferred height of 50 pixels (use the methods *setPrefWidth* and *setPrefHeight* to set these values at the beginning). Every time we move the mouse inside the button, its width and height must grow up to 300 x 250 pixels, in 2 seconds. As soon as we move the mouse outside of the button, it must recover its initial size. Do this animation with the manual technique.

5.3. Multi-device deployment

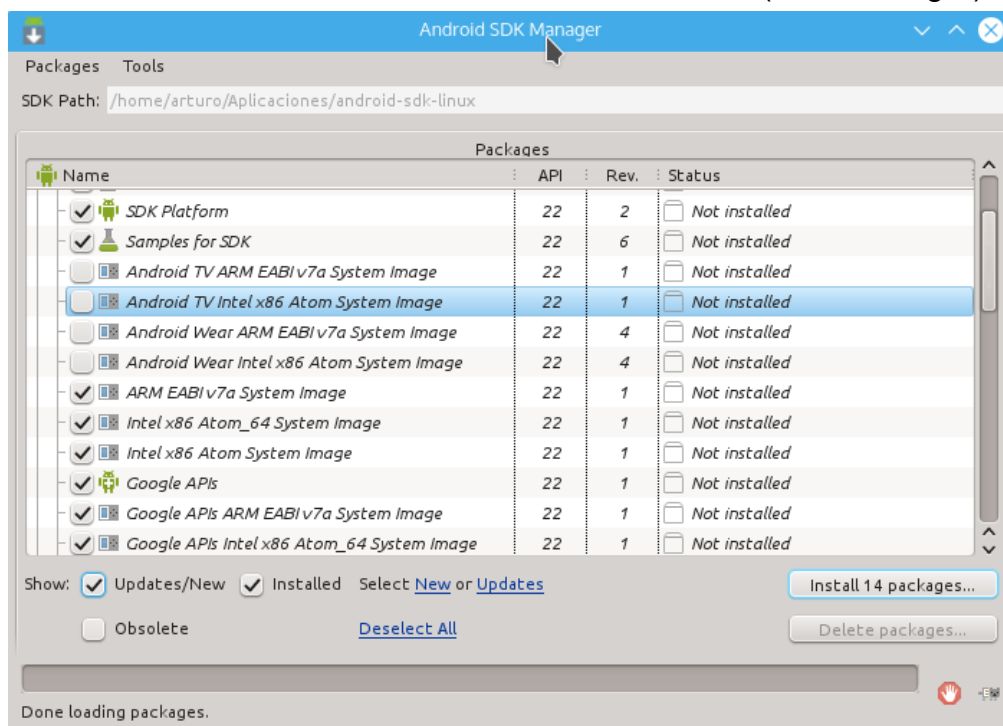
In this section we are going to see how to compile or package our JavaFX application so that it can be used in several platforms (desktops, Android devices, iOS devices...).

We are going to show an example of how to generate an Android apk, from our javafx source code. First, we'll download the JavaFX Mobile SDK:

<http://gluonhq.com/open-source/javafxports/downloads/>

Install android sdk: <https://developer.android.com/sdk/installing/index.html>

Then, we'll install the android platform we want to build for (in this case 5.1.1 or version 22). Go to Android SDK folder → tools → execute android file (SDK Manager)



To build a JavaFX application, go to the example HelloWorld/javafx application located inside the JavaFX SDK. This application is a perfect template for our JavaFX apps. The **gradlew** executable is a script we'll need to manage the compilation. To see what you can do execute **gradlew tasks**. To build the apk, use **gradlew build**.

Before doing anything, edit **local.properties** file to configure the paths to android SDK and JavaFX SDK:

```
# change this to the location of the android sdk on your local system
sdk.dir=/home/arturo/Aplicaciones/android-sdk-linux
# change this to the location of the JavaFX Dalvik SDK
# this location should have a subdirectory called "rt"
javafx.dir=/home/arturo/Descargas/dalvik-sdk
```

Inside **settings.gradle**, we can specify the name of our application. Inside build.gradle you can specify many things, but you must choose an installed Android platform and tools version:

```
android {
    compileSdkVersion 22
    buildToolsVersion "23"
    ...
}
```

The first time, maybe it will fail executing command called **lint**. Add this inside **build.gradle** (android section) to continue with the compilation.

```
android {
    ...
    lintOptions {
        abortOnError false
    }
}
```

Then we go to the **platform-tools** directory inside Android SDK, and start the adb server:

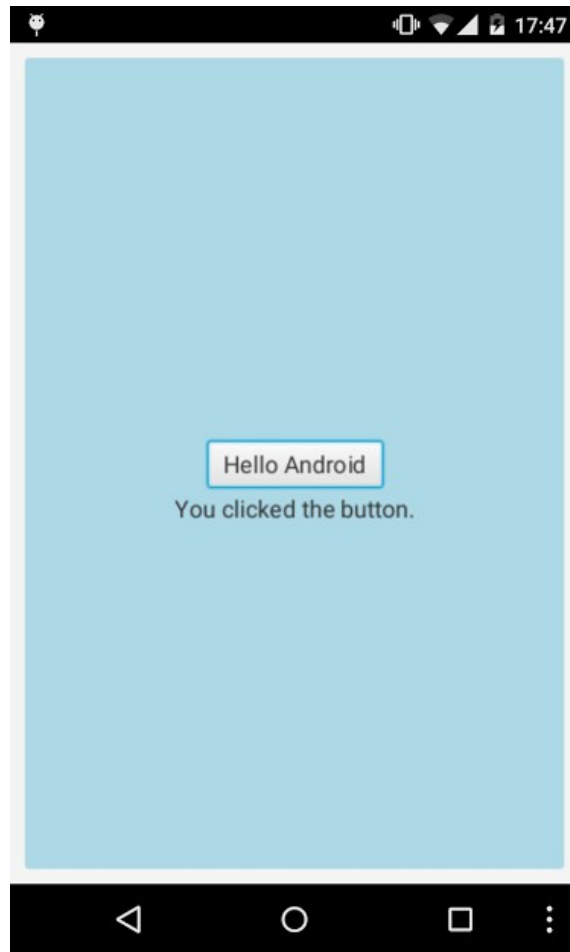
./adb start-server

Using **./adb devices**, we'll know the active Android devices (emulators included) present in our computer (we'll have to activate debug mode in the phone first).

Finally, when we see our device or emulator in the list, we can install our newly generated apk with this command:

./adb -s device_id install -r /route/to/apk/HelloWorld-debug.apk

And we're done, execute it and try. Just change the src files to change the app and execute **gradlew build** again.



Useful links:

<http://www.infoq.com/articles/Building-JavaFX-Android-Apps>

<http://docs.gluonhq.com/javafxports/>

6 Basic 2D Game Programming

While Java FX can be used for 3D rendering, there are a lot of good 3D engines out there that are far more advanced and easy to use than JavaFX. However, drawing and creating simple games in 2D is not very complicated with JavaFX. Here, in the last part of the unit we'll make a brief introduction about making games with JavaFX.

6.1 Rendering on a Canvas

In Java FX, the [Canvas](#) object is a rectangular area onto which we can draw shapes, texts or images (sprites). For drawing onto a Canvas, we use the [GraphicsContext](#) class. You can draw a shape or a text defining its fill and stroke color (using a [Paint](#) derived subclass object, like [Color](#)). You can also apply an [Effect](#) derived object to the result.

With the [Image](#) class, you can load an image from a file and draw it on the canvas.

```
public void start(Stage primaryStage) {
    Group root = new Group();
    Scene scene = new Scene(root, 400, 200);

    Canvas canvas = new Canvas(400, 200);
    root.getChildren().add( canvas );

    GraphicsContext gc = canvas.getGraphicsContext2D();

    //Effect (shadow) for drawing a rectangle
    DropShadow drop = new DropShadow(10, Color.GREY);
    drop.setOffsetX(4);
    drop.setOffsetY(6);

    // Draw a rectangle
    gc.setFill(Color.LIGHTSKYBLUE);
    gc.setEffect(drop);
    gc.fillRoundRect(10, 10, 380, 60, 10, 10);

    // Draw a text with a stroke
    gc.setEffect(null); // Unset the effect
    gc.setFill( Color.BLUE );
    gc.setStroke( Color.BLACK );
    gc.setLineWidth(2);
    Font myFont = Font.font( "Times New Roman", FontWeight.BOLD, 36 );
    gc.setFont( myFont );
    gc.fillText( "Hello, World!", 60, 50 );
    gc.strokeText( "Hello, World!", 60, 50 );

    // Draw an image
    Image earth;
    try {
        earth = new Image(Files.newInputStream(Paths.get("earth.png")));
        gc.drawImage( earth, 140, 85 );
    } catch (IOException e) {
        e.printStackTrace();
    }

    primaryStage.setTitle("Hello world!");
    primaryStage.setScene(scene);
}
```

```

    primaryStage.show();
}

```



6.2 The game loop

If we want to animate things, we'll have to periodically update the screen (scene). For doing that we need to create a loop that goes on until the animation finishes (which can be when the game ends). Usually, when we start making our first games (without using a game engine), we do that manually. JavaFX is not a game engine, but it has some helper classes like [AnimationTimer](#) for doing animation or game loops. It has an abstract method, **handle()**, that we'll have to implement extending a class or using an anonymous class:

```

public void start(Stage stage) {
    Group root = new Group();
    Scene scene = new Scene(root, 512, 512);
    stage.setScene(scene);

    Canvas canvas = new Canvas(512, 512);
    root.getChildren().add(canvas);

    GraphicsContext gc = canvas.getGraphicsContext2D();

    Image earth;
    try {
        earth = new Image(Files.newInputStream(Paths.get("earth.png")));
    } catch (IOException e) {
        e.printStackTrace();
        return;
    }

    final long startNanoTime = System.nanoTime();

    new AnimationTimer()
    {
        private float accumulatedTime = 0;
        private int lastFps = 0, countFrames = 0;
        private double lastNanoTime = startNanoTime;

        public void handle(long currentNanoTime)
        {
            // seconds total

```



```

        double t = (currentNanoTime - startNanoTime) / 1000000000.0;
        double x = Math.floor(t*100) % 450; // Every 10ms move 1px

        gc.setFill(Color.WHITE);
        gc.fillRect(0, 0, 512, 512);
        gc.drawImage( earth, x, 200 );

        // This is for keeping the frame count (default 60fps)
        countFrames++;
        accumulatedTime += (currentNanoTime - lastNanoTime) /
1000000000.0; // since last frame
        if(accumulatedTime >= 1.0) { // Every second restart counters
            lastFps = countFrames;
            accumulatedTime = 0;
            countFrames = 0;
        }

        gc.setFill(Color.RED);
        gc.fillText("FPS: " + lastFps, 400, 50);
        lastNanoTime = currentNanoTime;
    }
}.start();

stage.setTitle("Game Loop");
stage.show();
}

```

The **handle()** method of every **AnimationTimer** created is called about 60 times per second (60fps) by default. In this example an image is constantly moving to the right and when it reaches the end of the canvas (more or less), the animation begins again.

6.3 Events and interaction

Firstly, we are going to manage keyboard events. In general, we'll have to make our scene to listen to events (keyboard, mouse,...) by registering them using the appropriate methods. When registering keys that are pressed, we usually want to be able to register all that are at the same time. For keeping that track, we'll use a Set collection (we don't want to repeat keys) to store key codes.

```

private Set<KeyCode> activeKeys;

@Override
public void start(Stage stage) {
    ...

    activeKeys = new HashSet<>();
    scene.setOnKeyPressed(e -> activeKeys.add(e.getCode()));
    scene.setOnKeyReleased(e -> activeKeys.remove(e.getCode()));

    new AnimationTimer()
    {

        private int x = 0, y = 200;

        public void handle(long currentNanoTime) {
            if(activeKeys.contains(KeyCode.RIGHT))
                x++;
        }
    }
}

```

```

        if(activeKeys.contains(KeyCode.LEFT))
            x--;
        if(activeKeys.contains(KeyCode.UP))
            y--;
        if(activeKeys.contains(KeyCode.DOWN))
            y++;

        gc.setFill(Color.WHITE);
        gc.fillRect(0, 0, 512, 512);
        gc.drawImage( earth, x, y );
    }
}.start();

stage.setTitle("Game events");
stage.show();
}

```

The way to handle mouse events is the same:

```

int[] pos = {200, 200};

scene.setOnMouseClicked(e -> {
    pos[0] = (int)e.getX() - 25; // Image is 50x50
    pos[1] = (int)e.getY() - 25; });

new AnimationTimer()
{
    public void handle(long currentTime) {
        if(activeKeys.contains(KeyCode.RIGHT))
            pos[0]++;
        if(activeKeys.contains(KeyCode.LEFT))
            pos[0]--;
        if(activeKeys.contains(KeyCode.UP))
            pos[1]--;
        if(activeKeys.contains(KeyCode.DOWN))
            pos[1]++;

        gc.setFill(Color.WHITE);
        gc.fillRect(0, 0, 512, 512);
        gc.drawImage( earth, pos[0], pos[1] );
    }
}.start();

```

6.4 Sprites

In games, a Sprite represents a visual entity on the screen. It usually has methods to update the position, draw the image it contains, get the rectangle that represents the area it occupies on the screen for calculating collisions, check if it collides with another sprite, and many more...

We'll have a class called Sprite that has all we need for generic sprites. Then, we will make the derived classes necessary for the elements in the game (or sometimes we'll want to use composition and make the class, for example Enemy, store its Sprite object rather than inherit from it).

```

public class Sprite {
    private double x,y,width,height,velX,velY, maxVel;

```

```

private Image img;

public Sprite(double x, double y, double height, double width,
              double maxVel, Image img) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.img = img;
    this.maxVel = maxVel;
    this.velX = 0;
    this.velY = 0;
}

public void update(double elapsedTime) { // Time since last frame
    this.x += velX * elapsedTime;
    this.y += velY * elapsedTime;
}

public void accel(double accelX, double accelY) {
    velX += accelX;
    velY += accelY;

    if(velX > maxVel) velX = maxVel;
    if(velX < -maxVel) velX = -maxVel;
    if(velY > maxVel) velY = maxVel;
    if(velY < -maxVel) velY = -maxVel;
}

public void draw(GraphicsContext gc) {
    gc.drawImage(img, x, y);
}

public Rectangle2D getBoundary() {
    return new Rectangle2D(x, y, width, height);
}

public boolean intersects(Sprite s) {
    return s.getBoundary().intersects( this.getBoundary() );
}
}

```

Next, we're going to instantiate some Sprites and check for collisions, incrementing the score and removing them from the screen when our main sprite collides with them. Even when we want to make a simple game, the result should be far more complex and better organized in classes than this. This is only an example.

```

Sprite mainSprite = new Sprite(225, 225, 50, 50, 80, horse);

Random rand = new Random(System.nanoTime());
List<Sprite> fruits = new ArrayList<>();
for(int i = 0; i < 10; i++) {
    fruits.add(new Sprite(rand.nextInt(450), rand.nextInt(450),
                          50, 50, 0, apple));
}

new AnimationTimer()

```

```

{
    private double lastNanoTime = System.nanoTime();;

    public void handle(long currentNanoTime) {
        double elapsedTime = (currentNanoTime - lastNanoTime) /
10000000000.0;
        lastNanoTime = currentNanoTime;

        // Movement
        if(activeKeys.contains(KeyCode.RIGHT))
            mainSprite.accel(5, 0);
        if(activeKeys.contains(KeyCode.LEFT))
            mainSprite.accel(-5, 0);
        if(activeKeys.contains(KeyCode.UP))
            mainSprite.accel(0, -5);
        if(activeKeys.contains(KeyCode.DOWN))
            mainSprite.accel(0, 5);

        mainSprite.update(elapsedTime);

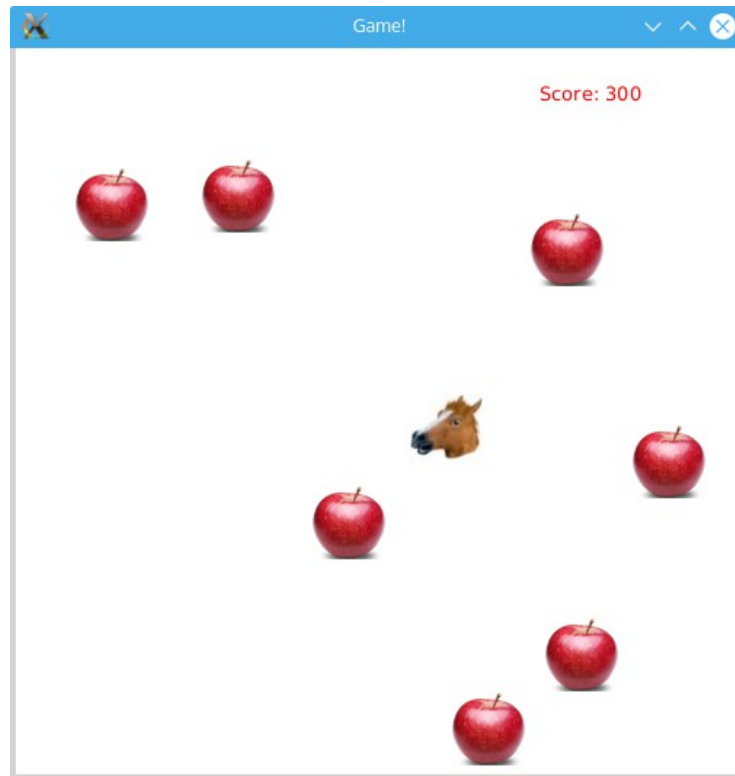
        // collision detection
        Iterator<Sprite> fruitsIter = fruits.iterator();
        while ( fruitsIter.hasNext() )
        {
            Sprite apple = fruitsIter.next();
            if ( apple.intersects(mainSprite) )
            {
                fruitsIter.remove();
                score += 100;
            }
        }

        // Render
        gc.setFill(Color.WHITE);
        gc.fillRect(0, 0, 512, 512);

        for(Sprite apple: fruits)
            apple.draw(gc);
        mainSprite.draw(gc);

        gc.setFill(Color.RED);
        String pointsText = "Score: " + score;
        gc.fillText( pointsText, 360, 36 );
    }
}.start();

```



Exercise 14

Modify the example above and make these changes and additions:

- The main character can't get out the boundaries of the window. When it reaches a border, it will rebound (change velocity's sign).
- There will be three red apples and 2 poisonous apples (for example green ones). Instead of disappearing, when a horse touches a red apple, it will get 100 points and that apple will move to another random place. When the horse touches a green apple three times the game will end (call **this.stop()** inside the **handle** method).
- Draw a counter starting in 3, showing the horse's lives. When it touches a poisonous apple it will decrement.