Beautiful User Interfaces with JavaFX

Systémes d'acquisition — 3EVE

S. Reynal

September 20, 2021

The current document is dedicated to giving you a small and quick insight into the JavaFX API, an extra Java library that will allow you to build beautifully styled user interfaces (UI). JavaFX has become a neat, modern and efficient replacement to the Swing API (the JFrame, JButton, etc. components), an API that surfaced more than ten years ago as an attempt to create the first Java cross-platform widget[1] kit and a replacement to the native AWT widget kit.

If you have been used to Swing components, you might find the philosophy behind JavaFX quite familiar at first, but this is merely the tip of the iceberg! JavaFX can do much more than Swing does, from fluid animations to CSS style sheet based styling to 3D shapes and transforms. Above all, JavaFX is hardware accelerated on most platform, either through DirectX on Windows, or through OpenGL on OS X or Linux.

Building a User Interface in JavaFX revolves mostly around creating a component tree made of nodes (branches and leaves), where leaves can be widgets (buttons and co), shapes (lines, rectangle, etc), text (with tons of styling options), images, sound or videos, charts and tables, and branches are actually groups thereof (e.g., menu, toolbars, complex drawings), see Figure 1. There are also special nodes that support dynamic processing, like transforms (rotations, translations) and animation effects (e.g., you want to make a 3D box rotate for 3 seconds), and every visible node can have an event listener attached to it, just as was already the case with Swing. Finally, every component in the tree supports a CSS based styling scheme, with the usual "classes" and "id" parameters of CSS styling. This makes it possible to split appearance issues from functionality issues, and have someone specialized in graphic design take care of the UI appearance while developers are busy designing the interface itself.

1 Installing things

There are three softwares to be installed:

[1] A "widget" is a component of a user interface like a button, a menu or a toolbar.

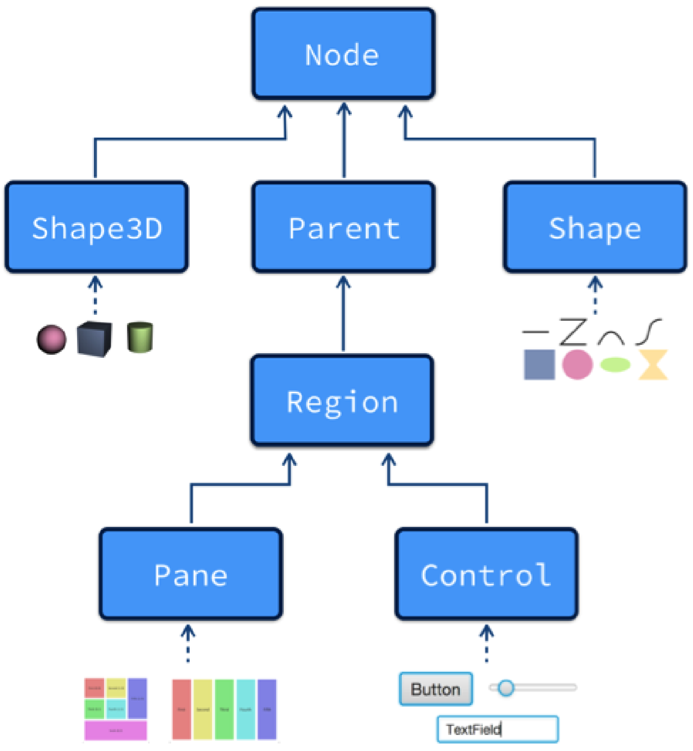

Figure 1: JavaFX component graph: the upper node is the root of the tree. It has three children, a Shape3D, a Shape2D, and a parent of a ”Region”, a group of nodes which itself aggregates together a Pane and a Control (e.g., a Button or a TextField). Applying a transform to a Region applies the transform to each of its children in turn, so that this is very efficient in terms of performance.

- A recent version of the Java Development Kit (aka JDK): you'll find it at https://www.oracle.com/java/technologies/downloads/ ; take the most recent version that matches your OS and install.
- Eclipse IDE: go to eclipse.org, go to "Download" and download the Eclipse installer ; run the installer and choose "Eclipse for Java". Launch Eclipse and pickup the proposed (automatically created) workspace folder or create a new one if the proposed one doesn't fit your needs (this folder will contain all your Java projects).
- JavaFX libraries: go to https://openjfx.io/ and click "Download" on the main page (scroll down a bit), this will take you to a dedicated website (aka Gluon) where here again you'll have to download a version that matches your platform. Uncompress to Eclipse workspace.

Then from inside Eclipse create a new Java project, and "link" the JavaFX libraries to your project :

- right click on your Project name
- select "Properties", then "Java Build Path", then "Libraries", then "Add External JARs..."
- add the following libraries (they should have been uncompressed to Eclipse workspace folder) : javafx.base.jar, javafx.control.jar, javafx.graphics.jar

(you'll find alternative tips at https://www.javatpoint.com/javafx-with-eclipse)

2 A basic application

Writing a JavaFX application begins with overriding the `start()` method of the `javafx.application.Application` class. The argument of the `start()` method is a `Stage` object which represents the primary stage of the UI. You can have several stages: these simply correspond to distinct windows. Every `Stage` has a title (which is displayed in the window upper bar) and a `Scene` object attached to it, where the `Scene` is somehow the "top" of the component UI tree. A `Scene` has a dimension, a location on screen, and contains a root component (a `BorderPane` in the example given in Fig. 2) which can itself have children if it is a branch node (containers are always branch nodes so this is ok). Finally, a `Scene` can have one or several CSS style sheets attached to it.

3 Running a JavaFX app from the command line

Open a shell (or a "Command" or a "Terminal", depending on your OS), and go to Eclipse workspace, then to your Project folder. Change to the

```
import javafx.animation.Application;

public class Main extends Application {

 public void start(Stage primaryStage) {
  try {
   BorderPane root = new BorderPane();
   root.setTop(createToolbar());
   root.setBottom(createStatusbar());
   root.setCenter(createMainContent());
   Scene scene = new Scene(root,800,400);
   scene.getStylesheets().add(getClass().
      getResource("application.css").toExternalForm());
   primaryStage.setScene(scene);
   primaryStage.setTitle("Demo␣JavaFX");
   primaryStage.show();
  } catch(Exception e) {e.printStackTrace();}
 }

 private Node createToolbar(){
  return new ToolBar(new Button("New"), new Button("Open"),
    new Separator(), new Button("Clean"), etc);
  }

 private Node createStatusbar(){
  HBox statusbar = new HBox();
  statusbar.getChildren().addAll(new Label("Name:"),
    new TextField("␣␣␣␣") etc.);
  return statusbar;
  }

 private Node createMainContent(){
  Group g = new Group();
  // here you fill g with whatever Nodes you want
  // using g.getChildren().add(...)
  return g;
 }

etc.
}
```

Figure 2: Building the component UI tree is simple: create a root node (here a `BorderPane`), fill its various inner areas (top, bottom, center), then attach it to a `Scene` and attach the `Scene` to the `Stage`.

”bin” folder where all compiled classes are located, and run the following command:

```
java --module-path path-to-javafx-libraries-folder
                 --add-modules javafx.controls application.Main
```

4 Displaying text, images, 2D and 3D shapes

You can easily add text and 2D or 3D geometrical shapes to your interface by creating the appropriate object (e.g., a `Line`) and adding it to a parent container. This container may either be a :

- `Group` (in which case it just acts as a way to maintain a link between every node belonging to it, for example, if you add a `Transform` to a `Group`, then every node in the group undergoes this transform)
- Or a sophisticated container, like e.g. `BorderPane`, that can also layout its children neatly on the screen, in particular because it takes care of how its dimension influences the layout.

Figure 3 shows a basic code block for adding text. Literally every parameter in the `Text` object can be tweaked, from font size to stroke color and width. Changes are reflected immediately on the screen through a hidden event signaling mechanism. Styling can also be specified using the associated CSS style sheet, using the ”.text” class.

The package `javafx.scene.shape` contains a lot of 2D shapes, from lines to circles to Bezier splines. It also includes classes dedicated to 3D graphics (boxes, spheres, cylinders) which, when associated with transforms (see package javax`.scene.transform`), makes it possible to build complicated 3D worlds. More information is available in the API document of javafx, `http://docs.oracle.com/javase/8/javafx/api/toc.htm`.

Figure 4 shows how to add a single line to your interface. Here again you may either want to add it to a parent group, or to a more sophisticated container.

Finally, it is also possible to add images, videos or sound directly to your scene by creating the appropriate objects and wrapping them into a parent group. Adding an image is quite straightforward, and is exemplified in Figure 5. Just create an `Image` with the appropriate image URL (many format are supported, but PNG is the preferred one), wrap it into an `ImageView` (which is the node responsible for rendering the image to the screen), set the `ImageView` parameters so that the image has the correct location and size on screen, possible add an effect to the image by attaching an effect object (see `javafx.scene.effect` package), and finally attache the `ImageView` to a parent group, as always.

```
Text text = new Text("Text can\nstraddle two lines");
text.setTextAlignment(TextAlignment.JUSTIFY);
text.setFont(Font.font("Times New Roman",
    FontWeight.BOLD, FontPosture.REGULAR, 40));
text.setFill(Color.CYAN);
text.setStrokeWidth(2);
text.setStroke(Color.BLACK);
text.setUnderline(true);
text.setX(50);
text.setY(150);
... (then add to parent group)
```

Figure 3: Adding text to your interface reduces to instanciating a `Text` object, setting its font, location, stroking and alignement properties, and adding it to a `Group` or any other type of parent node (e.g., a `Borderpane`).

```
Line line = new Line();
line.setStartX(300.0);
line.setStartY(150.0);
line.setEndX(500.0);
line.setEndY(150.0);
line.setStroke(Color.GREEN);
```

Figure 4: Creating geometrical shapes is not much complicated: create a shape using one of the many classes available in the `javafx.geometry` package, and add it to a parent container.

```
Image image = new Image(url of PNG image);
ImageView imageView = new ImageView(image);
imageView.setX(500);
imageView.setY(70);
imageView.setFitHeight(200);
imageView.setFitWidth(400);
Glow glow = new Glow();
glow.setLevel(0.9);
imageView.setEffect(glow);
```

Figure 5: To add an image to the UI, create an `Image` object, wrap it into an `ImageView`, possibly add visual effets like `Glow` (or any other in the `javafx.scene.effect` package), and finally add the `ImageView` to a parent container.

```
import javafx.scene.input.MouseEvent;
...
node.setOnMouseClicked(
  new EventHandler<MouseEvent>() {
    public void handle(MouseEvent e) {
      System.out.println("mouse␣click!");
    }
  });
}
```

Figure 6: Handling events in JavaFX: it is enough to just attach an event handler to a (visible) node to capture any event occuring on this node (a button, a shape, a piece of text, etc).

5 Adding widgets and handling events

Figure 6 and 7 show how to implement a simple UI with ”widgets” (buttons, menus, etc) and how to handle events triggered on them by a mouse click. The approach is very similar to Swing, since in Swing components were already handled by a ”component graph” made up of leaves and nodes. Those used to Swing will surely recognize the familiar `Button` and `GridPane` objects: as it turns out they work pretty much like in Swing and you can seamlessly translate a piece of Swing code into a piece of JavaFX code with minimum effort. Mouse and keyboard events are handled using `EventHandler` objects with appropriate filters (here a `MouseEvent` filter), and bear strong resemblance with Swing's `ActionListener`'s.

```
Text txt = new Text("Password");
PasswordField passField = new PasswordField();
Button button = new Button("Submit");

GridPane gridPane = new GridPane();
gridPane.setMinSize(400, 200);
gridPane.setPadding(new Insets(10, 10, 10, 10));
gridPane.setVgap(5);
gridPane.setHgap(5);
gridPane.setAlignment(Pos.CENTER);

gridPane.add(txt, 0, 1);
gridPane.add(passField, 1, 1);
gridPane.add(button, 0, 2);

button.setStyle("-fx-background-color: darkslateblue");
```

Figure 7: Basic steps for creating a UI with buttons and textfields.