JavaFX - A Short Intro

Table of Contents

Before you start..

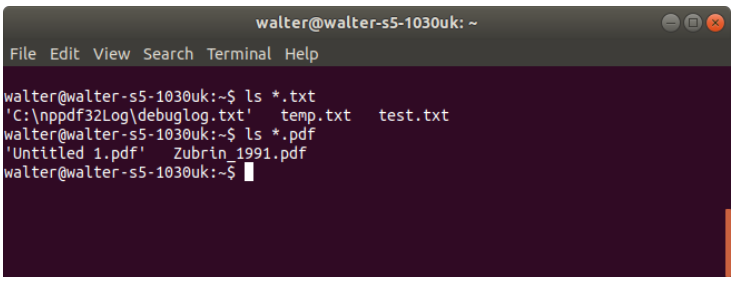
You need to know Java and OOP. Do not try this at home unless you know and understand core Java.

What is a GUI?

A GUI is a Graphical User Interface.

A User Interface is the way a user uses a system - inputting and outputting messages.

Early UI were command line interfaces - the user typed in commands, pressed Enter, and the command was executed. This is sometimes called a treminal or console or DOS box. It means the user needs to know correct commands and syntax.



A second type of UI is menu-driven - the user is shown a small set of choices:

A GUI uses windows, a mouse or other pointing device, icons, 'buttons', check boxes, scroll bars and so on.

Java GUI history

In 1995 with its first release there was AWT - the Abstract Windowing Toolkit

In 1997 Swing was introduced to replace most of AWT.

JavaFX 1 was released in 2008, and JavaFX 8 (this version) in 2014

The JavaFX 8 API is at

https://docs.oracle.com/javase/8/javafx/api/toc.htm

Why not?

JavaFX (and the other GUIs) are not part of the Java *language*. A developer does not need to know JavaFX. Many colleges teach Swing as if it were required. It is not. It is an option. It is also outdated, replaced by JavaFX.

A disadvantage with Java is that it requires a JRE installed. This is needed for Java to run cross-platform. Installing a JRE is no problem for technical users, but non-technical users are reluctant to do so.

The alternative is to have a web interface, in a browser, with Java code kept server-side. This needs no install of anything, and almost all devices have a web browser.

So desktop Java GUI apps are of limited use.

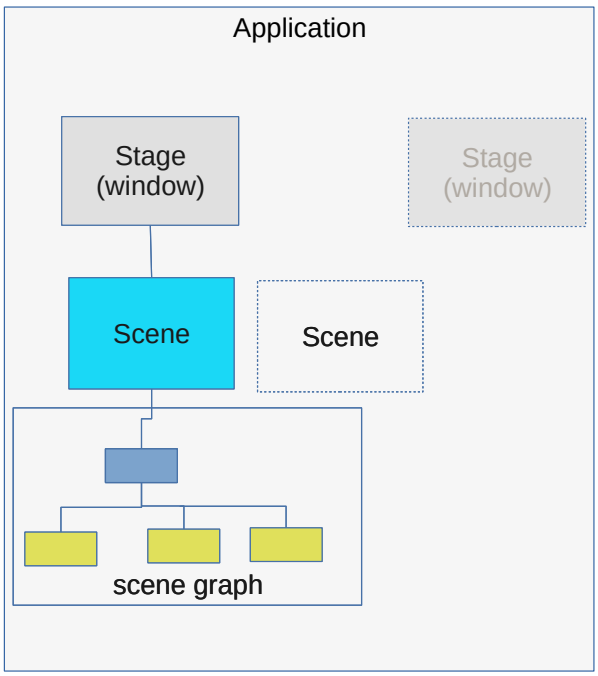
These notes are simply to get people started on a replacement for Swing, if they really want it.

Basic ideas

A Stage is a *window* - what was a JFrame in Swing. Stage is a class. A JavaFX application has one, or maybe more than one, Stage.

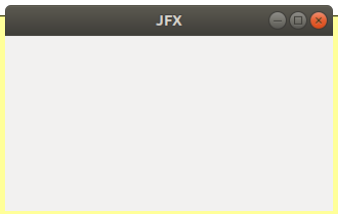
A Scene is a holder for a set of components, like text fields and buttons. Scene is also a class. A Stage has a Scene on it. The idea is that this is like a theatre, where we have a stage, and a scene appears on the stage. We can display different scenes on the same stage.

A scene graph is the set of components displayed on a Scene. Each of these is an instance of a sub-class of Node. Some Nodes are controls (like buttons). Others are containers of other components, called layouts. A GridPane is an example.



Basic Stage

A JavaFX application needs to be a sub-class of javafx.application.Application, which is abstract, and to implement the start method, which is called when the application is launched. We already have a Stage, passed as an argument to the start method. So a very simple program is:

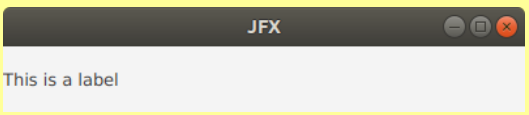



```
package javafxapplication2;

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


public class JFX extends Application {

    @Override
        public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("JFX");
        primaryStage.show();
    }

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

}
```

This is done in Netbeans, using Java 1.8, in a 'JavaFX project'. It could have been in Eclipse or any other IDE.

Adding a Scene

We can add a scene onto the stage:




```
package javafxapplication2;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class JFX extends Application {

  @Override
  public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("JFX");
    Label label = new Label("This is a label"); // make a label
    Scene scene = new Scene(label, 400, 50); // make a Scene
    primaryStage.setScene(scene); // put scene on stage
    primaryStage.show();
  }

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

}
```

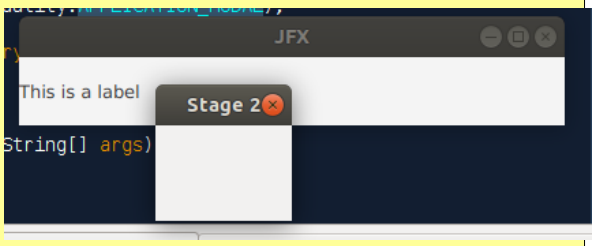
In the Scene constructor, the first parameter is the root node of the scene graph. In fact this scene graph only has one node - the label. The other two parameters are the initial width and height of the scene.

Two Stages

An application can have 2 Stages. Just listing the start method..

```
  public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("JFX");
    Label label = new Label("This is a label");
    Scene scene = new Scene(label, 400, 50);
```

```
    primaryStage.setScene(scene);
    primaryStage.setX(500); // set position
    primaryStage.setY(500);
    primaryStage.show();

    Stage stage2 = new Stage(); // make another stage
    stage2.setTitle("Stage 2");
    stage2.setWidth(100);
    stage2.setHeight(100);
    stage2.setX(600);
    stage2.setY(550);

    stage2.initModality(Modality.APPLICATION_MODAL);

    stage2.initOwner(primaryStage);
    stage2.show();
  }
```



Most of the methods do what they say - for example, setWidth. Except for the primary stage, a Stage has an 'owner' which we set here:

```
stage2.initOwner(primaryStage);
```

This:

```
stage2.initModality(Modality.APPLICATION_MODAL);
```

means the window is modal - so we cannot use - close or whatever - the primary window, until we first close the second stage. This is what you usually want for a dialog box, for example where you are asking the user to select a file to open. They cannot close the dialog until they have chosen the file. Read the API for the other modalities.

Controls - Buttons

As a simple example of a control node, we will add a button to the scene:

```
  public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("JFX");

    Button button = new Button("Click here");

    Scene scene = new Scene(button, 400, 50);

    primaryStage.setScene(scene);
    primaryStage.setX(500); // set position
    primaryStage.setY(500);
    primaryStage.show();
  }
```

The button looks strange because it fills the entire scene. This is because we have not set up a layout manager. Also the button does nothing when clicked. We need an event handler.

Event handlers

An event handler is some code which will run when some 'event' happens. Events are mostly user input actions, like clicking a button, typing text, dragging something, checking a checkbox and so on. Event handlers are call-back functions - you set them up, but do not call them. The system calls them when needed to handle the appropriate event.

We first make the control. Then we set the event handler to do what we want when the user uses the control. For a button, that means the user clicks on it. So we add this code..

```
   public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("JFX");

    Button button = new Button("Click here");
    button.setOnAction(new EventHandler() {
      @Override
```

```
      public void handle(Event actionEvent) {
       // we put here what to do when its clicked..
       System.out.println("Button clicked");
      }
    });
    ..
  }
```

This uses an *anonymous class*. The **new** EventHandler() {.. means this is an instance of a new class. The class is a sub-class of EventHandler - which is abstract - with an implementation of the handle method - to make it concrete. This new class does not have a name - it is anonymous.

In this example it is very simple and just prints on the console. But code here could do whatever we want.

GridPane - a layout manager

A layout manager is a scene graph node used to control the size and location of nodes below it in the graph.

A GridPane is a rectangular grid of cells. We can place other nodes in cells, and they can span more than 1 row or column. We

1. Make the components

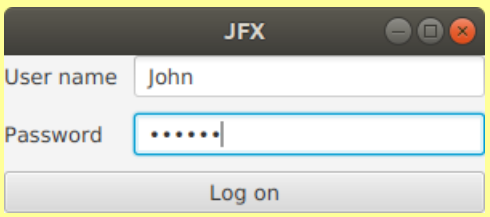
2. Make the grid

3. Add components to the grid

4. Make a scene, with the grid as root node

5. Add the scene to the stage:

```
public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("JFX");
    // make components
    Label nameLabel = new Label("User name");
    Label passwordLabel = new Label("Password");
    TextField name = new TextField();
    PasswordField pass = new PasswordField();
    Button button = new Button("Log on");
    button.setMinWidth(300);
    // make grid
    GridPane gridPane = new GridPane();
    gridPane.setHgap(10);
    gridPane.setVgap(10);
    // add components to grid
    gridPane.add(nameLabel, 0, 0, 1, 1);
    gridPane.add(passwordLabel, 0, 1, 1, 1);
    gridPane.add(name, 1, 0, 1, 1);
    gridPane.add(pass, 1, 1, 1, 1);
    gridPane.add(button, 0, 2, 2, 1);
    // make scene
    Scene scene = new Scene(gridPane);
    // set scene on stage
    primaryStage.setScene(scene);
    primaryStage.setX(500); // set position
    primaryStage.setY(500);
    primaryStage.show();
  }
```



A statement like

```
gridPane.add(nameLabel, 0, 0, 1, 1);
```

means to add nameLabel to the grid, row 0, column 0, spanning 1 row and column.

We would need to set an event handler on the button for how to log on.

FXML

Setting up a UI with program code like this is not a good idea. A program is a set of instructions, to *do something*. But a UI is static - it just sits there *doing nothing* - except in response to user events.

So a better approach is to set out the UI components in a separate file. This corresponds to how things are in a web page - html sets the content, CSS controls the layout and appearance, and Javascript does any coding needed.

An FXML file is an XML file which shows what nodes are used in a scene graph. We can add a file to our project - here named FXML.fxml:

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

<VBox>
    <children>
        <Label text="Hello world FXML"/>
        <Button text="Click me"/>
        <TextField text="Type here"/>

    </children>
</VBox>
```

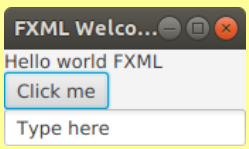
This uses a VBox - a layout manager which simply positions components in a vertical box. It will contain a label, a button and a textfield.

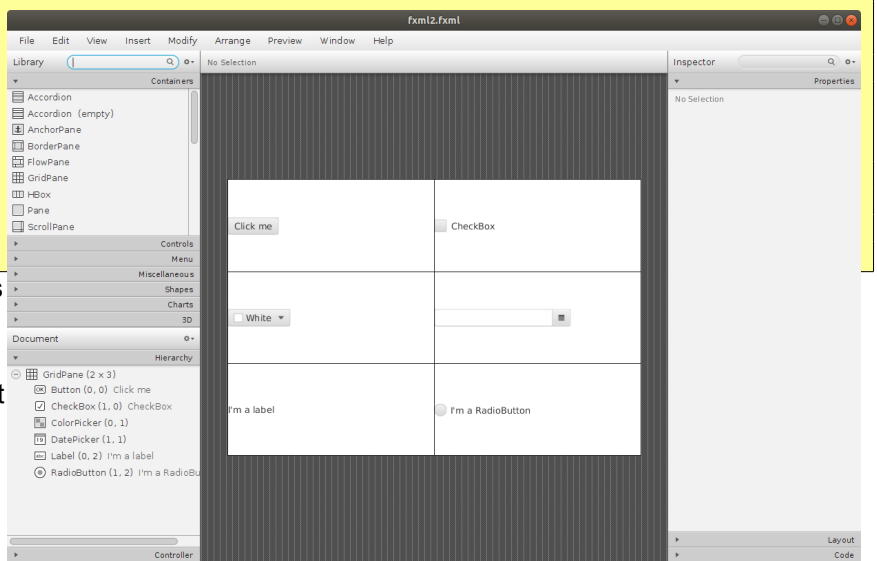
We need to import the Java classes used, like

```
<?import javafx.scene.layout.VBox?>
```

We load and use the FXML like this:

```
package javafxapplication2;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class JFX extends Application {

  @Override
  public void start(Stage stage) throws Exception {
    Parent root = FXMLLoader.load(getClass().getResource("FXML.fxml"));

    Scene scene = new Scene(root);

    stage.setTitle("FXML Welcome");
    stage.setScene(scene);
    stage.show();

  }

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

}
```





Parent is an abstract class which is the base class for classes which are nodes with children. FXMLLoader.load returns an object

which is a concrete sub-class of this. We make a new Scene with this as the root node, and set it on the stage.

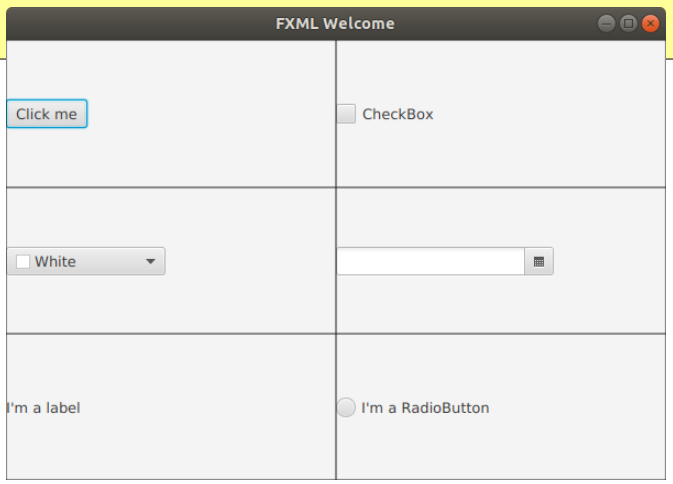
Scene Builder

In place of writing the FXML file manually, Scene Builder from Oracle is an application which can be used to make the interface visually - and then use the file produced before. For example:

This writes out the following FXML file:

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

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

<GridPane alignment="CENTER" gridLinesVisible="true" maxHeight="-Infinity" maxWidth="-Infinity"
minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0"
xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/8">
  <columnConstraints>
    <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
    <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
  </columnConstraints>
  <rowConstraints>
    <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
    <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
    <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
  </rowConstraints>
   <children>
      <Button mnemonicParsing="false" text="Click me" />
      <CheckBox mnemonicParsing="false" text="CheckBox" GridPane.columnIndex="1" />
      <ColorPicker GridPane.rowIndex="1" />
      <DatePicker GridPane.columnIndex="1" GridPane.rowIndex="1" />
      <Label alignment="CENTER" contentDisplay="CENTER" text="I'm a label"
textAlignment="CENTER" GridPane.rowIndex="2" />
      <RadioButton mnemonicParsing="false" text="I'm a RadioButton" GridPane.columnIndex="1"
GridPane.rowIndex="2" />
   </children>
</GridPane>
```

which runs as:



Using a controller class

If we use FXML to set up the GUI, we still need event handlers to make things happen when the controls are used.

It is possible to put that in the FXML, or in the sub-classed Application which started everything. But we get more separation of code by writing a separate *controller class*.

We need to tell the FXML what its controller class is, and we need a way of linking nodes in the scene graph with variables in the controller code.

We look at a simple example again, using a button click. The Application class is the same:

```
package javafxapplication2;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class JFX extends Application {

  @Override
  public void start(Stage stage) throws Exception {
    Parent root = FXMLLoader.load(getClass().getResource("FXML.fxml"));
    Scene scene = new Scene(root);
    stage.setTitle("FXML Welcome");
    stage.setScene(scene);
    stage.show();
  }

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

}
```

The FXML is:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TextField?>
<VBox xmlns:fx="http://javafx.com/fxml" fx:controller="javafxapplication2.Controller" >
    <children>
        <Label   fx:id="target" text="Hello world FXML"/>
        <Button text="Click me" onAction="#clickAction"/>
    </children>
</VBox>
```

and the controller is

```
package javafxapplication2;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Label;

public class Controller {

 @FXML private Label target;

    @FXML protected void clickAction(ActionEvent event) {
        target.setText("Button clicked");
    }


}
```

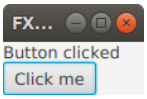
In the FXML,

```
<VBox xmlns:fx="http://javafx.com/fxml" fx:controller="javafxapplication2.Controller" >
```

identifies the controller class. In that, the @FXML annotation results in the linking of the method as an event handler, and the other node.

```
onAction="#clickAction"/>
```

in the FXML links with




```
 @FXML protected void clickAction(A....
```

in the controller. And

```
 <Label   fx:id="target"
```

links with

```
 @FXML private Label target;
```

in the controller.

Summary

We have different options for writing JavaFX:

1. All programmatically, in the sub-classed Application class

2. Using an FXML file to say what layouts and controls are in the scene graph

3. We can write the FXML by hand, or use Scene Builder to make it

4. We can use a controller class to place the event handlers.

Using CSS

Suppose we want to change the colours, use bigger text, different fonts and so on. In other words, change the appearance.

We can do that in code. But we will continue to separate out the different concerns in our project, by using a separate *style sheet*.

This is standard for web pages, An html file fixes content (text, images and links)while a CSS file controls the appearance. CSS is cascading stye sheet. It is the same idea, but slightly different styles, for JavaFX.

The application is:

```
package javafxapplication2;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class JFX extends Application {

  @Override
  public void start(Stage stage) throws Exception {
    Parent root = FXMLLoader.load(getClass().getResource("FXML.fxml"));
    Scene scene = new Scene(root);
    scene.getStylesheets().add("javafxapplication2/styles.css");
    stage.setTitle("FXML Welcome");
    stage.setScene(scene);
    stage.show();
  }

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

```
    JFX.launch(args);
  }

}
```

The FXML is

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TextField?>
<VBox >
    <children>
        <Label  text="Hello world FXML"/>
        <Button text="Click me" />
        <Label  text="Another label"/>
        <Button text="Another button" />
    </children>
</VBox>
```

And the stylesheet file, styles.css, is

```
.button {
   -fx-background-color: red;
   -fx-text-fill: white;
}

.label
{
    -fx-font: Garamond;
    -fx-font-size: 14pt;
    -fx-background-color: blueviolet;
    -fx-padding: 5px;
}
```

This is similar to a web CSS. We have a selector, like .button, then 1 or more style rules. These have the form attribute: value; like -fx-background-color: red; This is similar to a web version, but there you would say just background: red;