Lesson 6, Part I: Building GUIs with JavaFX

Wholeness of the Lesson

JavaFX is a UI library in Java that allows developers to create user interfaces that are rich in content and functionality.

The ultimate provider of tools for the creation of beautiful and functional content in manifest existence is pure intelligence itself; all creativity arises from this field's self-interacting dynamics.

Overview

▸ Using JavaFX components (learn more from api docs at http://docs.oracle.com/javafx/2/api/)

▸ Layout basics

▸ Handling GUI events

▸ Using CSS to style your app

▸ Declarative UI building using FXML

▸ Deployment

First JavaFX Application

```java
public class FirstApp extends Application {
    @Override
    public void start(Stage stage) {
        Label label = new Label("Welcome");
        StackPane root = new StackPane(); // Layout
        root.getChildren().add(label);  // adding components in the layout

        Scene scene = new Scene(root, 400, 100);
        stage.setTitle("First FX");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```
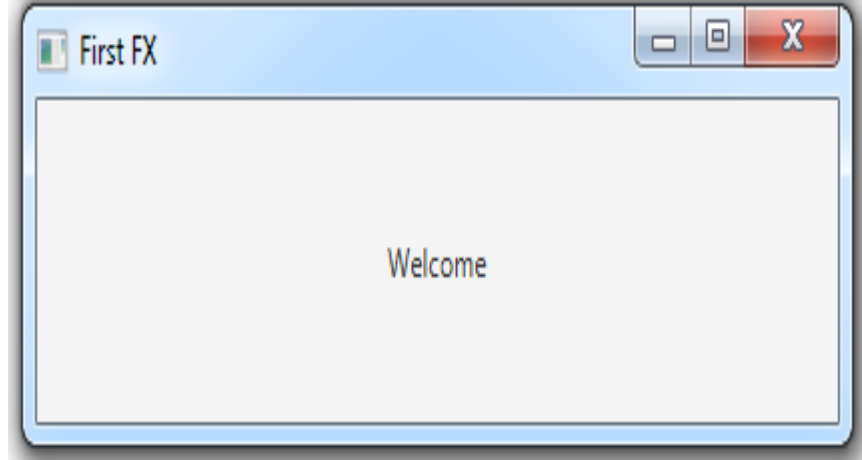
About FirstApp.java

- **Application class**
  - The entry point for a JavaFX application is always a user-defined subclass of the abstract class javafx.application.Application class.
  - The start() method starts up the application – it is the only abstract method of Application (and so must be implemented).

- **Stage Class**
  - A JavaFX application defines the user interface container by means of a *stage* and a *scene*.
  - The JavaFX Stage class is the top-level JavaFX container. (Entire Window)
  - The JavaFX Scene class is the container for all content. (Contents on a stage)
  - The demo above creates the stage and scene and makes the scene visible in a given pixel size.

▸ **StackPane Class (Kind of Layout)**

▸ In JavaFX, the content of the scene is represented as a hierarchical scene graph of *nodes*.

▸ In this example, the root node is a StackPane object, which is a resizable *layout* node.

Scene Graph
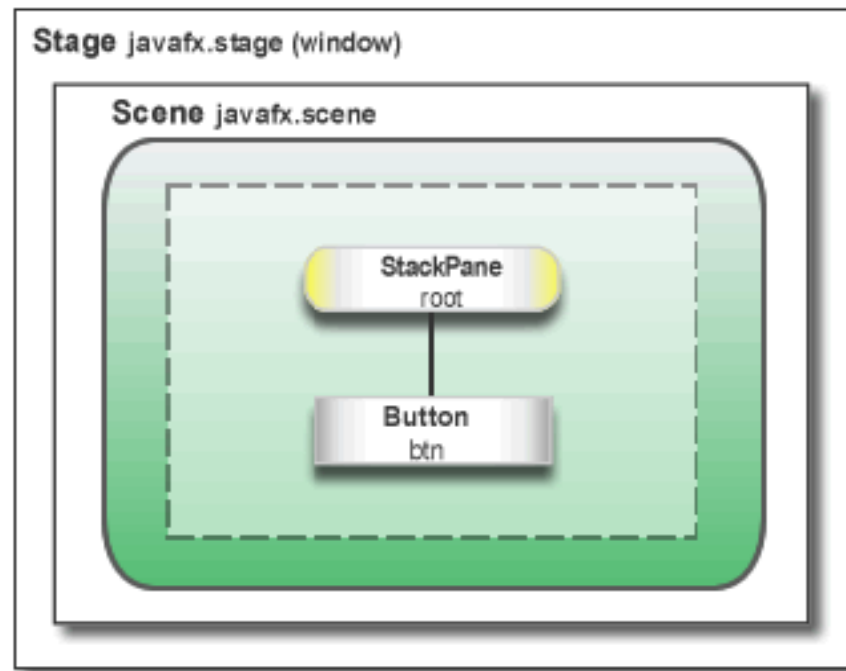
Hello World in JavaFX

Hello World

Stage

Scene

Group (root)

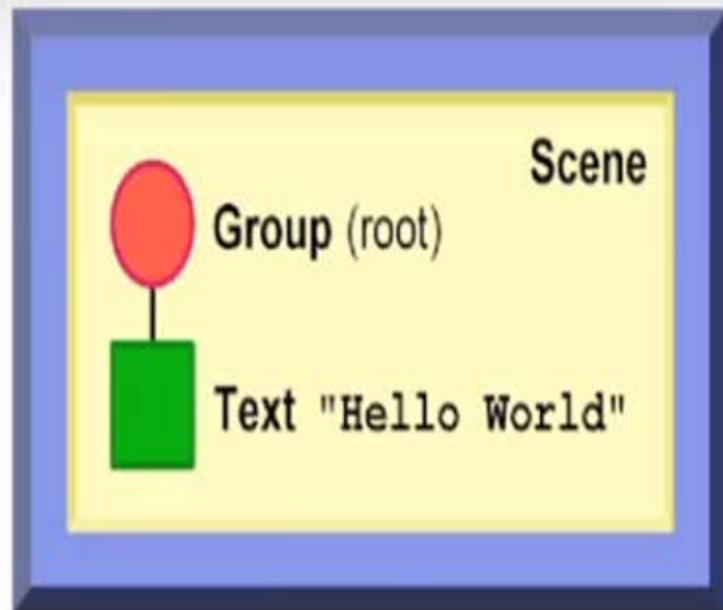**Text** "Hello World"

```
@Override public void start(Stage stage) {
    Group root = new Group();
    Scene scene = new Scene(root, 200, 100);
    root.getChildren().add(new Text(50, 50, "Hello World"));
    stage.setScene(scene);
    stage.show();
}
```

Main Point 1

For creating the look of a JavaFX application, two types of classes are primary: *components* and *containers*.

A screen is created by setting the stage with the Stage and Scene container classes. And then the screen is populated with components, like buttons, textboxes, labels, and so on, starting at a root node and extending.

Components and containers are analogous to the *manifest* and *unmanifest* fields of life; manifest existence, in the form of individual expressions, lives and moves within the unbounded container of pure existence.

Some General Points about JavaFX Apps

▸ The main() method is not required for JavaFX applications when the JAR file for the application is created with the JavaFX Packager tool.

  ▸ The JavaFX Packager tool is a commandline tool used to compile, package and deploy a Java FX application (can be found here: <path to Java> \Java\jdk1.8.0_45\bin\javafxpackager.exe)

  ▸ Using the tool embeds the JavaFX Launcher in the output JAR file, so there is no need to call the Launcher from a main method.

▸ It is useful to include the main() method because

  ▸ You can run JAR files that were created without the JavaFX Launcher (such as when using an IDE in which the JavaFX tools are not fully integrated, which is the case with Eclipse).

  ▸ Swing applications that embed JavaFX code require the main() method.

▸ Read about the Packager tool here: http://docs.oracle.com/javafx/2/deployment/packager.htm

▸

The Life-cycle of a JavaFX Application

The entry point for JavaFX applications is the Application class. The JavaFX runtime does the following, in the following order, whenever an application is launched:

1. Constructs an instance of the specified Application class

2. Calls the init() method (for initializing; typical use: read commandline args See HelloWorld demo)

3. Calls the start(javafx.stage.Stage) method

4. Waits for the application to finish, which happens when either of the following occurs:

   ▸ the application calls Platform.exit() (this can be done explicitly in code)

   ▸ the last window has been closed (this is the default behavior, but it can be changed) - See HelloSecondWindow demo

5. Calls the stop() method (typical use: clean up connections to resources)

   ▸ The start method is abstract and must be overridden.

   ▸ The init and stop methods have concrete implementations that do nothing by default

Two Threads

1. *Launcher thread.* Application constructor and init method are called on this thread

2. *Application thread.* JavaFX creates an application thread for running the application start method, processing input events, and running animation timelines.

  ▸ Creation of JavaFX [Scene] and [Stage] objects as well as modification of scene graph operations to live objects (those objects already attached to a scene) must be done on the JavaFX application thread.

Second Example - Creating a Form in JavaFX

▸ In this example we build a login form – illustrates:

  ▸ basics of screen layout

  ▸ how to add controls to a layout pane

  ▸ how to create input events and handle them.

Create a GridPane Layout

▸ GridPane is a layout node that works like an HTML table. You can place controls in any cell in the grid, and you can make controls span cells as needed.

```
GridPane grid = new GridPane();
grid.setAlignment(Pos.CENTER);
grid.setHgap(10);
grid.setVgap(10);
grid.setPadding(new Insets(25, 25, 25, 25));
Scene scene = new Scene(grid, 300, 275);
```

▸ *alignment* property changes the default position of the grid from the top left of the scene to the center.

▸ *gap* properties control the spacing between the rows and columns in the grid

▸ *padding* property controls the space around the edges of the grid pane

▸ *insets* occur in this order: *top, right, bottom*, and *left*. (In this example, there are 25 pixels of padding on each side).

▸

Add Text, Labels, and Text Fields

▸ Sample Code:

```
Label scenetitle = new Label("Welcome");
scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));  //better to set these
values in a style sheet
 // Parameter are : Node child, int columnIndex, int rowIndex, int colspan, int rowspan
    grid.add(scenetitle, 0, 0, 2, 1);

    Label userName = new Label("User Name:");
    grid.add(userName, 0, 1);

    TextField userTextField = new TextField();
    grid.add(userTextField, 1, 1);

    Label pw = new Label("Password:");
    grid.add(pw, 0, 2);

    PasswordField pwBox = new PasswordField();
    grid.add(pwBox, 1, 2);
```

▷

▸ TextFields are constructed to have width of 12 pixels by default. The preferred width can be modified using setPrefColumnCount(numCols), or setPrefWidth(pixels), but layout may override preferences.

▸ apply the setPrefColumnCount method of the TextInput class to set the size of the text field

▸ How positions in the grid are specified: (*column num, row num*):

(0, 0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)

▸ Text scenetitle = new Text("Welcome");

▸ grid.add(scenetitle, 0, 0, 2, 1);

▸ The last two arguments of the grid.add() method set the column span to 2 and the row span to 1.

Debugging components on a GridPane

▸ For debugging, GridPane allows you to display the grid lines. Do this here by adding this line of code grid.setGridLinesVisible(true)

▸ When code is run, you see this:
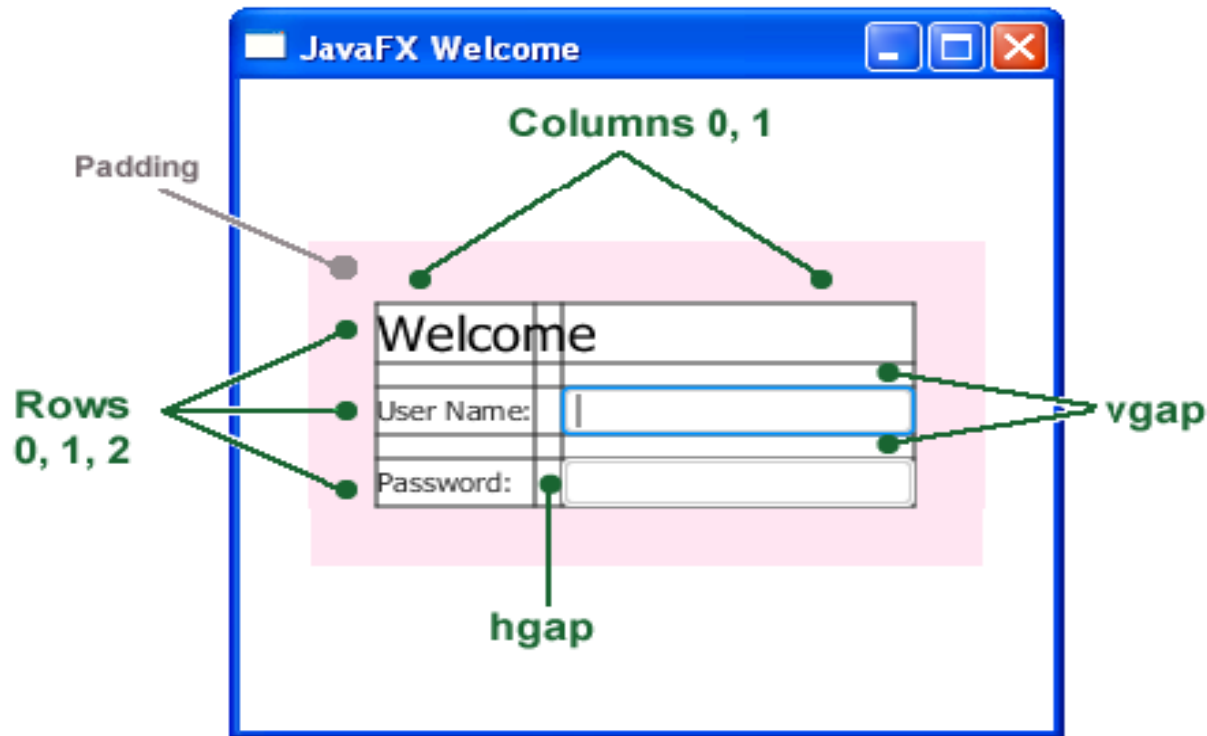
Positioning a Component in GridPane with HBox

▸ Sample code:

```
Button btn = new Button("Sign in");
HBox hbBtn = new HBox(10);
hbBtn.setAlignment(Pos.BOTTOM_RIGHT);
hbBtn.getChildren().add(btn);
grid.add(hbBtn, 1, 4);
```

▸ The HBox layout pane is created to allow you to place the button in a special place.

▸ HBox constructor accepts a spacing parameter (spacing between components). In this case, alignment is set to Pos.BOTTOM_RIGHT

▸ If other components are added to an HBox, they are laid out from left to right.

▸

Other Layouts

Pane Class	Description
HBox, VBox	Lines up children horizontally or vertically.
GridPane	Lays out children in a tabular grid, similar to the Swing GridBagLayout.
TilePane	Lays out children in a grid, giving them all the same size, similar to the Swing GridLayout.
BorderPane	Provides the areas North, East, South, West, and Center, similar to the Swing BorderLayout.
FlowPane	Flows children in rows, making new rows when there isn't sufficient space, similar to the Swing FlowLayout.
AnchorPane	Children can be positioned in absolute positions, or relative to pane's boundaries. This is the default in the SceneBuilder layout tool.
StackPane	Stacks children above each other. Can be useful for decorating components, such as stacking a button over a colored rectangle.

Main Point 2

In JavaFX, components are arranged in a container through the use of *layouts* that organize components in different ways. The most convenient layout is GridPane, which allows you to organize components in a table format, and suffices for most layout needs. For special layout requirements, JavaFX provides half a dozen other layout types.

Likewise, all of manifest life is conducted by a vast network of natural laws.

Event Handling

▸ Sample code:

```
final Text actiontarget = new Text();
grid.add(actiontarget, 1, 6);
// Anonymous implementation
btn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent e) {
        actiontarget.setFill(Color.FIREBRICK);
        actiontarget.setText("Sign-in button pressed");
    }
});
```

▸ Add a Text control for displaying the message, as shown below.

▸ The setOnAction() method is used to register an event handler that sets the value of actiontarget to "Sign-in button pressed" when the user presses the button. The color of the actiontarget object is set to firebrick red.
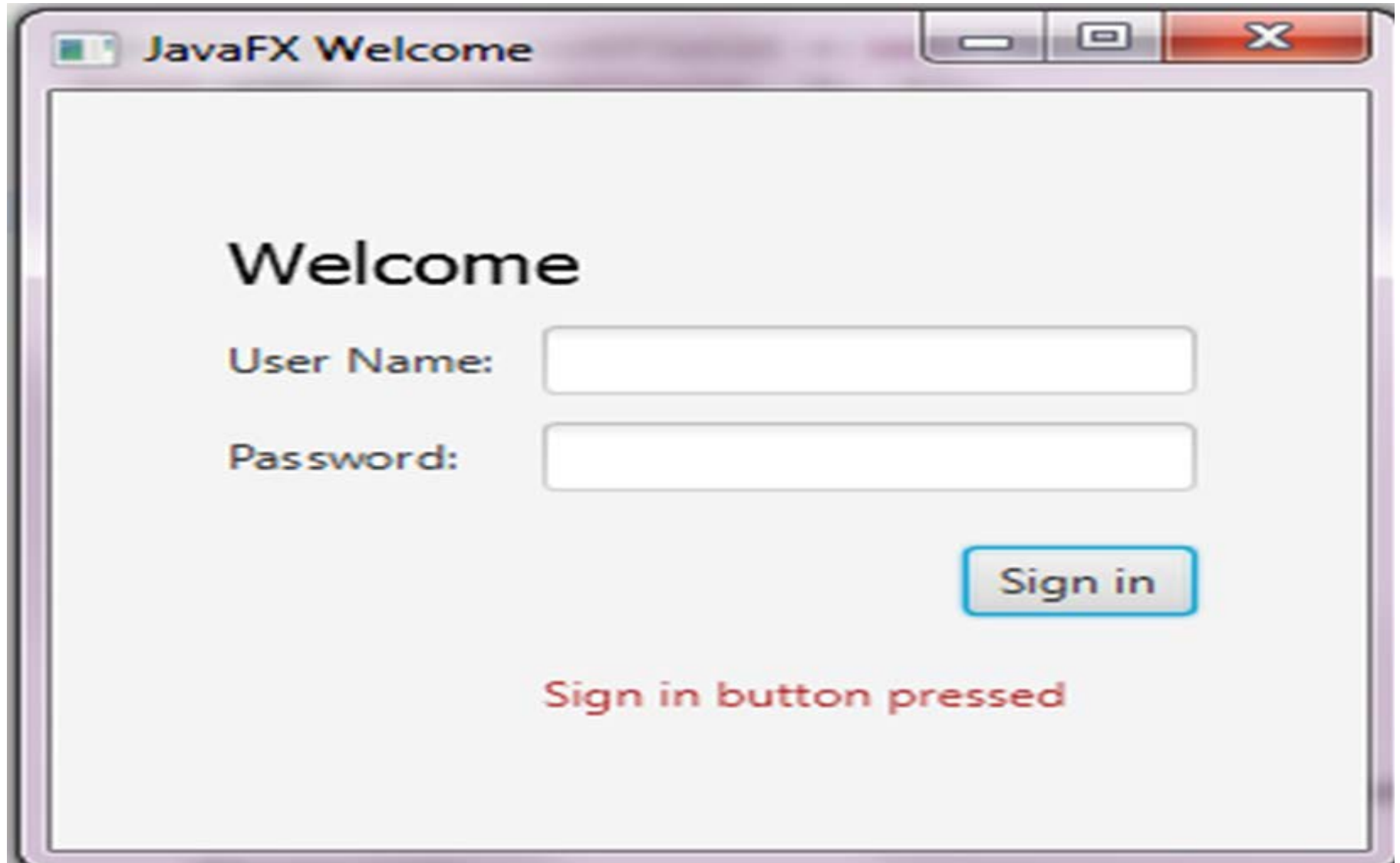
More Examples on Demo

Demo lesson6.lecture.javafx.secondwindow illustrates:

▸ Setting background color of the root
▸ Creating and using a status bar
▸ Setting up a ComboBox and responding to user selections with a ChangeListener
▸ ToggleButton for toggling between states in response to button clicks
▸ Creating multiple Stages and how communication between them is accomplished.

Demo lesson6.lecture.javafx.tables illustrates the use of menus and tables in JavaFX.

▸ The Start class creates a MenuBar, adds two Menus, and adds MenuItems to the first Menu.
▸ The ShoppingCartWindow creates a TableView. TableViews are created one column at a time. In this example, all cells have been made read-only, but with more work, cells can be editable. A TableView requires some Java class to provide the data that it will read and present. In this example, ShoppingCart provides the necessary data. During TableView construction, the field names from ShoppingCart are specified and each will represent one of the TableView columns. After the ShoppingCartWindow has been created, data for the table can be set using the setData method that has been provided. This method accepts a List of ShoppingCarts; each ShoppingCart in the list will be displayed as a row in the table.

Main Point 3

A GUI becomes responsive to user interaction (for example, button clicks and mouse clicks) through the event-handling model of JavaFX, in which event sources are associated with EventHandler classes, whose handle method is called (and is passed an Event object) whenever a relevant action occurs. To make use of this event-handling model, the developer defines a handler class, implements the handle method, and, when defining an event source (like a button), registers the handler class with this event source component.

The "observer" pattern that is used in JavaFX mirrors the fact that in creation, the influence of every action is felt everywhere; existence is a field of infinite correlation; every behavior is "listented to" throughout creation.

Fancy Forms with JavaFX CSS

▸ For this example, we are going to add a Cascading Style Sheet (CSS) to the JavaFX application as shown below. See

Add CSS Styling to the Login Class

▸ Create a new CSS file and save it in the same directory as Login.css.

▸ Specify location of the CSS file using the following code:

Scene scene = new Scene(grid, 300, 275);
primaryStage.setScene(scene);
**scene.getStylesheets().add(
getClass().getResource("Login.css").toExternalForm());**
primaryStage.show();

Reference:
▸ http://docs.oracle.com/javafx/2/css_tutorial/jfxpub-css_tutorial.htm

Detailed reference:
▸ http://docs.oracle.com/cd/E17802_01/javafx/javafx/1.3/docs/api/javafx.scene/doc-files/cssref.html

Other ways to access a CSS file are discussed here:
▸ https://blog.idrsolutions.com/2014/04/use-external-css-files-javafx/

▸

Add a Background Image

```
.root {
-fx-background-image: url("background.jpg");
}
```

The background image is applied to the .root style, which means it is applied to the root node of the Scene instance.

The style definition consists of

the *name* of the property: -fx-background-image, and

the *value* for the property: url("background.jpg").

Style the Labels

When you specify *.label* in your stylesheet, the values that are set affect all Labels in the form.

.label {

-fx-font-size: 12px;

-fx-font-weight: bold;

-fx-text-fill: #333333;

-fx-effect: dropshadow(

gaussian , rgba(255,255,255,0.5) , 0,0,0,1);

}

This example

- ▸ sets the font size and weight
- ▸ sets text-fill to gray
- ▸ applies a drop shadow (the purpose of the drop shadow is to add contrast between the dark gray text and the light gray background).
- ▸ rgba is RGB + alpha. Alpha (values in range 0..1) specifies opacity
- ▸ dropshadow parameters can be looked up at http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

▸

Style Text

Apply css styling to the two Text objects: *scenetitle* (includes the text Welcome) and *actiontarget* ("signed in" message at bottom).

Steps:

1. Remove the Java coding of styles (we will replace them with CSS) Remove the following lines of code that define the inline styles currently set for the text objects:

   scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
   actiontarget.setFill(Color.FIREBRICK);

2. Create an ID for each text node by using the setID() method of the Node class:

   scenetitle.setId("welcome-text");
   actiontarget.setId("actiontarget");

Style Text

3. In the Login.css file, define the style properties for the welcome-text and actiontarget IDs. For the style name, use the ID preceded by a number sign (#), as shown below –

```
#welcome-text {
-fx-font-size: 32px;
-fx-font-family: "Arial Black";
-fx-fill: #818181;
-fx-effect: innershadow( three-pass-box , rgba(0,0,0,0.7) , 6, 0.0 , 0 , 2 );
}

#actiontarget {
-fx-fill: FIREBRICK;
-fx-font-weight: bold;
-fx-effect: dropshadow( gaussian , rgba(255,255,255,0.5) , 0,0,0,1 );
}
```

NOTES: The text fill color for welcome-text is set to a dark gray color (#818181) and an inner shadow effect is applied, creating an embossing effect.

Style the Button

We style the button so that it changes style when the user hovers the mouse over it.

Steps:

Create the style for the initial state of the button by adding the code below. This code uses the .button style class selector, such that if you add a button to the form at a later date, then the new button will also use this style.

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

Create a slightly different look for when the user hovers the mouse over the button. You do this with the hover *pseudo-class*. A pseudo-class includes the selector for the class and the name for the state separated by a colon (:), as shown below -

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

Using FXML to Create a User Interface

1. *Mark-up Language.* FXML is a mark-up language based on XML that is used to design and lay out JavaFX components, and attach event handlers; FXML markup is rendered by the JVM into a fully functioning UI.

2. *Declarative programming.* FXML makes it possible to develop UI code in a *declarative* style, using XML commands to declare *what* is needed rather than writing the Java code that accomplishes the goal. This flexibility makes it possible to develop FXML code using a designer tool, which supports drag-and-drop layout of components and event-handling. The tool that accomplishes this is called SceneBuilder.

3) Topics for FXML: (Refer : Lesson-Review of XML.doc)

a) We give a quick review of XML, including an example of how it is read and used in a Java program

b) We re-build the small Login app using FXML (without the use of SceneBuilider) showing FXML markup syntax and how it is used by JavaFX code.

c) In Part II, we show how to work with FXML documents using Scene Builder.

FXML Basics

▸ When building a UI using FXML, there are two types of documents to create: the FXML file(s) and the controlling Java file(s). Here are shells of these for the Login app:

```java
public class FXMLExample1 extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("fxml_example1.fxml"));

        stage.setTitle("FXML Welcome");
        stage.setScene(new Scene(root, 300, 275));
        stage.show();
    }

    public static void main(String[] args) {
        Application.launch(FXMLExample1.class, args);
    }
}
```

Java Code

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

<?import java.net.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.text.*?>

<GridPane alignment="center" hgap="10" vgap="10">
  <padding><Insets top="25" right="25" bottom="10" left="25" /></padding>

  <!-- children of the GridPane root go here -->


</GridPane>
```

FXML Code

1. XML processing instructions are used to specify Java imports

2. The root of the FXML document is the root of the Scene that is being built.

3. Nesting in the FXML document parallels the nesting that is done in building JavaFX components (as we did in building the Login app).

4. The Java code shows how to access the values specified in the FXML document. After obtaining the root, further JavaFX changes can be made directly in Java.

FXML Example 1: Component Layout

The sample code in lesson6.lecture.javafx.fxmlexample.FXMLExample1 shows how components are laid out using an FXML document

FXML Example 2: Connect Java and FXML Code

The demo

lesson6.lecture.javafx.fxmlexample.FXMLExample2

shows how to link between Java code and the FXML code, using id fields (similar to the approach in JavaScript).

Note:

1. The id fields are specified as attributes of component elements in the FXML document.
   `<Text id="actiontarget" GridPane.columnIndex="1" GridPane.rowIndex="6" />`

2. You reference components having id tags in the FXML document using Java syntax like the following:
   `Text target = (Text)root.lookup("#actiontarget");`

3. In this approach, event-handling can be done inside the Java code by retrieving (by id) an event-generating component (like a Button) and attaching an event-handler (see the demo).

FXML Example 3: Event Handling

The demo lesson6.lecture.javafx.fxmlexample.FXMLExample3

shows how to handle events in a better way, by injecting event-generating components into a Controller class, responsible for event-handling code.

 Notes:

 1. You inject components into a controller class using the @FXML annotation.
2. Components that need to be referenced in the controller must have "fx:id" tags instead of simply "id" tags.
3. You reference an event handler within the FXML document with code like this:

  `<Button text="Sign In" onAction="#handleSubmitButtonAction" />`
When the button is clicked, the JVM will look for a method handleSubmitButtonAction in the controller class, and will execute this method.
4. To tell the JVM about the class that you will use as controller, you include an fx:controller attribute in the root, like this:

  `fx:controller= "lesson6.lecture.javafx.fxmlexample.FXMLExampleController"`
5. To tell the JVM about the "fx" namespace, you also include in the root the attribute xmlns:fx, like the following:

xmlns:fx=http://javafx.com/fxml
6. Using a Controller class to be responsible for event-handlers is an application of the *Mediator design pattern* (discussed in Software Engineering) and supports the principle "separation of concerns" – separating the static UI layout code from the dynamic event-handling code.

FXML Example 4:

- **Referencing CSS Styles in the FXML Document**

The demo

lesson6.lecture.javafx.fxmlexample.FXMLExample4

adds lines to the FXML document to handle CSS styling.

Deploying Your First JavaFX Application

‣ JavaFX applications can be run in several ways:
  ▸ Launch as a desktop application from a JAR file or self-contained application launcher
  ▸ Launch from the command line using the Java launcher
  ▸ Launch by clicking a link in the browser to download an application
  ▸ View in a web page when opened

‣ For this course, we are going to do it in the first way.

‣ Here are instructions for doing that:

‣ From your Eclipse workspace:
  ▸ right click your JavaFX application
  ▸ create a special Run Configuration – example: HelloWorldToJar (Duplicate existing Run Configuration and rename it) , then Close
  ▸ right click on the application again and select export…select Java folder
  ▸ Runnable JAR file (Next)
  ▸ Select launch configuration you just created
  ▸ choose Export Destination

‣ Now, run the JAR file you just created by double-clicking the file.

Lesson 6, Part II: Building GUIs with SceneBuilder

▸ Make sure you have a version of Eclipse that supports SceneBuilder. We have used efxclipse, which is Luna with several SceneBuilder plugins preloaded
http://efxclipse.bestsolution.at/install.html#all-in-one

▸ Download and install SceneBuilder. Instructions are in the setup folder for the course

▸ Go through a demo to build the Login app using SceneBuilder

Steps for creating JavaFX Project using Scene Builder

▸ File → New → Other → JavaFX → JavaFXProject

▸ Add new FXML Document by right click on the project

    ▸ File → New → Other → JavaFX → New FXML Document

▸ Create a Controller class in the same package to do the necessary functionalities.

▸ Right click on the FXML Document → OpenwithSceneBuilder

▸ Design your screen according to your need

▸ In the screenBuilder, Left side of the screen select controller to choose your controller class.

▸ In Controller class to access the controls from the screen declare the private variables as the type of controls and put @FXML annotation before the variables and for action event methods too.

▸ To bind the declared variables in to Scene Builder, on the ride side of control properties, select code and set the variable names in fx:id.

▸ To perform the action, select the particular control and on the right side select code, then select onAction and choose the specific function.

▸ In your main class remove or comment line the code line //BorderPane root = new BorderPane();

▸ Add the following code

▸ Parent root = FXMLLoader.*load(getClass().getResource("/application/Main.fxml"));*

▸ Run your Main class