# Lesson 6:
# 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.

***About This Lesson.*** This is a quick introduction to

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

b. Layout basics

c. Handling GUI events

d. Style sheets

e. Declarative UI building using FXML

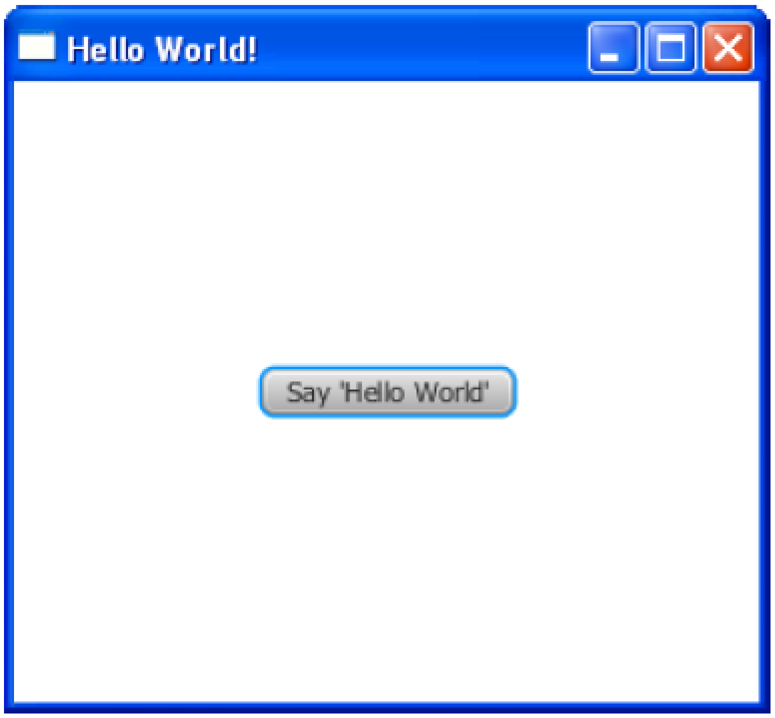
f. Deployment

## First Example - HelloWorld.java

```
public class HelloWorld extends Application {
   public static void main(String[] args) {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage) {
      primaryStage.setTitle("Hello World!");
      Button btn = new Button();
      btn.setText("Say 'Hello World'");
      btn.setOnAction(new EventHandler<ActionEvent>() {
         @Override
         public void handle(ActionEvent event) {
            System.out.println("Hello World!");
         }
      });
      StackPane root = new StackPane();
      root.getChildren().add(btn);
      primaryStage.setScene(new Scene(root, 300, 250));
      primaryStage.show();
   }
}
```

**Output:**



## Points About the Hello World Example

### 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. The JavaFX `Scene` class is the container for all content. The demo above creates the stage and scene and makes the scene visible in a given pixel size.

### StackPane Class

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. This means that the root node's size tracks the scene's size and changes when the stage is resized by a user. (In a StackPane nodes are added along the z axis. Makes it easy to overlay text on a shape or image or to overlap common shapes to create a complex shape)

### Components

The root node contains one child node, a button control with text, plus an event handler to print a message when the button is pressed.

## Main Point 1

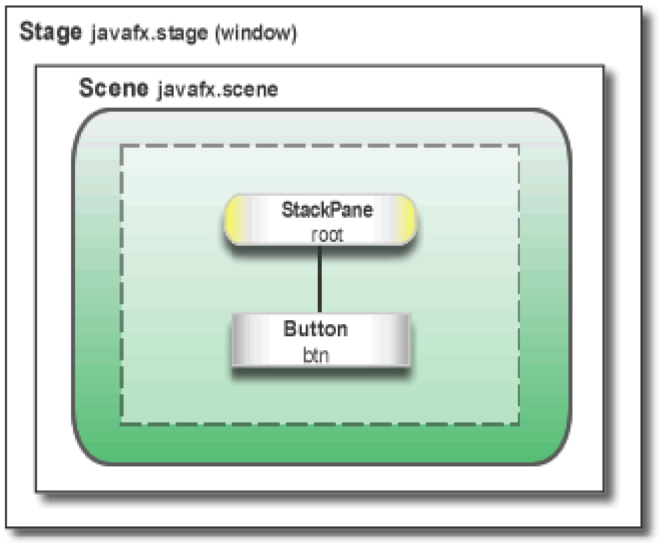
For creating the look of a JavaFX application, two types of classes are primary: *components* and *containers.* A screen is created by setting the stages with the Stage and Scene container classes. And then the screen is populated with components, like buttons, textboxes, labels, and so on. Components and containers are analgous 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.

## Main Method Is not Required

1. 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_05\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.
2. 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.
3. Read about the Packager tool here: http://docs.oracle.com/javafx/2/deployment/packager.htm

## Scene Graph

The figure below shows the scene graph for the Hello World application.



## Life-cycle

The entry point for JavaFX applications is the Application class. The JavaFX runtime does the following, in 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)

NOTE:

- 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 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.

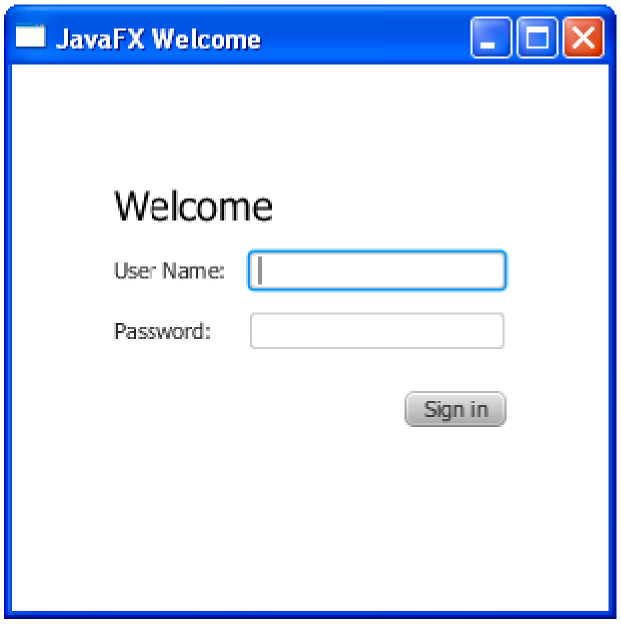
   - Creation of JavaFX Scene and Stage objects as well as modification of scene graph operations to live objects (those objects already attached to a scene) must be done on the JavaFX application thread.

NOTE: This means that an application must not construct a Scene or a Stage in either the constructor or in the init method.

## 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.



## Sample Code

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("JavaFX Welcome");
    GridPane grid = new GridPane();
    grid.setAlignment(Pos.CENTER);
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets(25, 25, 25, 25));

    Text scenetitle = new Text("Welcome");
    scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL,
          20));
    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);

    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);

    final Text actiontarget = new Text();
    grid.add(actiontarget, 1, 6);

    btn.setOnAction(new EventHandler<ActionEvent>() {

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

    Scene scene = new Scene(grid, 300, 275);
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

## 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);
primaryStage.setScene(scene);
```

- *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).

NOTES:
- When the scene is created, the grid pane is the root node – this is usual approach when using layout containers
- As the window is resized, the nodes within the grid pane are resized according to their layout constraints.
- The padding properties ensure there is a padding around the grid pane when you make the window smaller.
- Code sets the scene width and height to 300 by 275. If you do not set the scene dimensions, the scene defaults to the minimum size needed to display its contents.

# Add Text, Labels, and Text Fields

Sample Code:

```
Text scenetitle = new Text("Welcome");
scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL,
     20));  //better to set these values in a stylesheet
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.

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

| | | |
|---|---|---|
| (0, 0) | (1, 0) | (2, 0) |
| (0, 1) | (1, 1) | (2, 1) |

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

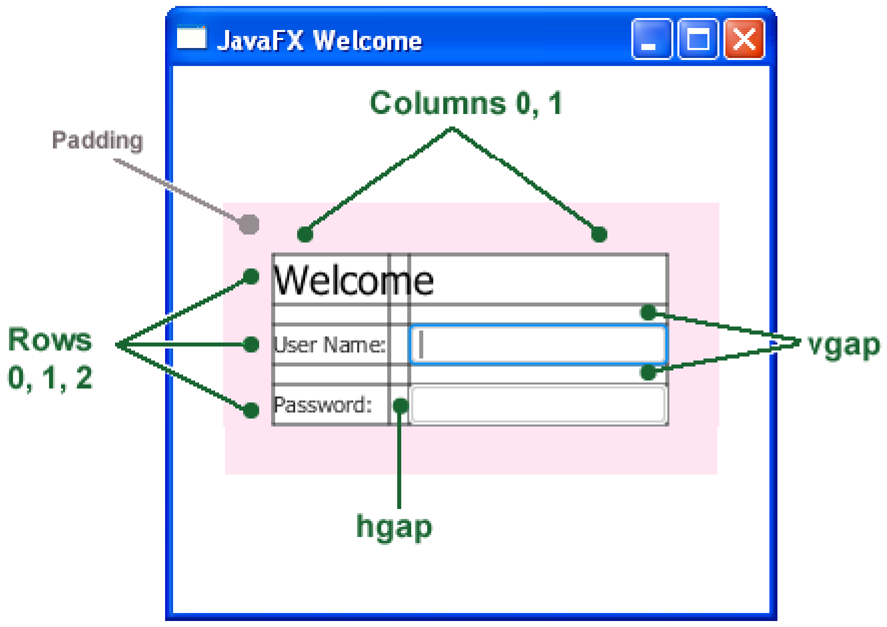
## Debugging

For debugging, `GridPane` allows you to display the grid lines. Do this here by adding this line of code

```
grid.setGridLinesVisible(true)
```

right after adding the password field.

When code is run, you see this:



## Positioning a Component in GridPane with HBox

Sample code:

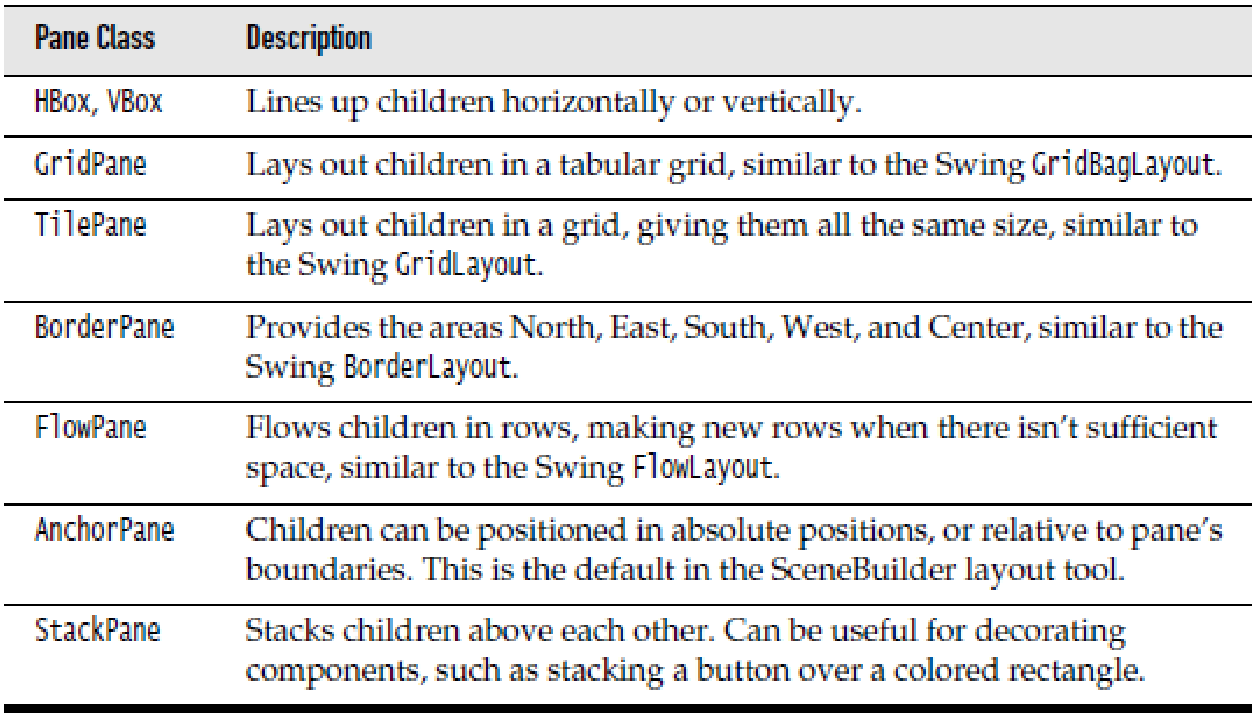
```
Button btn = new Button("Sign in");
HBox hbBtn = new HBox(10);
hbBtn.setAlignment(Pos.BOTTOM_RIGHT);
hbBtn.getChildren().add(btn);
grid.add(hbBtn, 1, 4);
```

- 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:

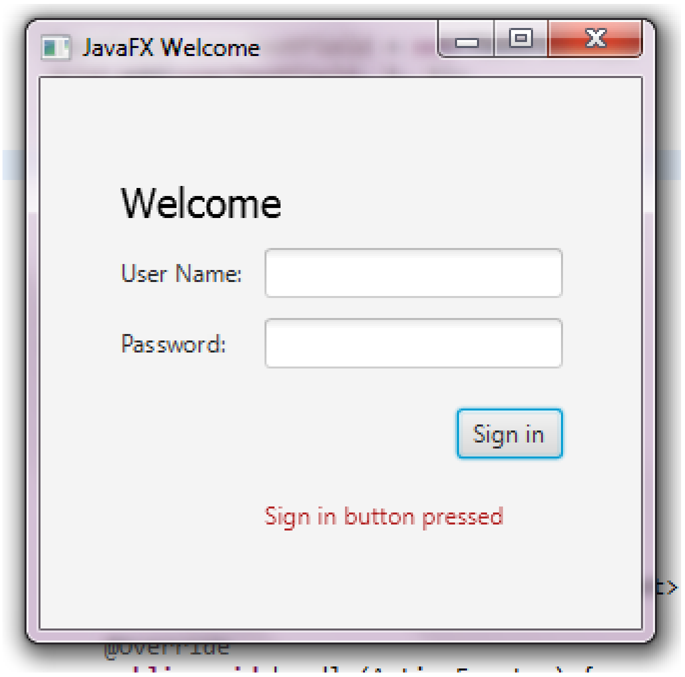
```
final Text actiontarget = new Text();
grid.add(actiontarget, 1, 6);

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 actiontarget object to Sign in button pressed when the user presses the button. The color of the actiontarget object is set to firebrick red.



Sign in button pressed

## Third Example – Other Techniques

Demo `HelloSecondWindow` illustrates:

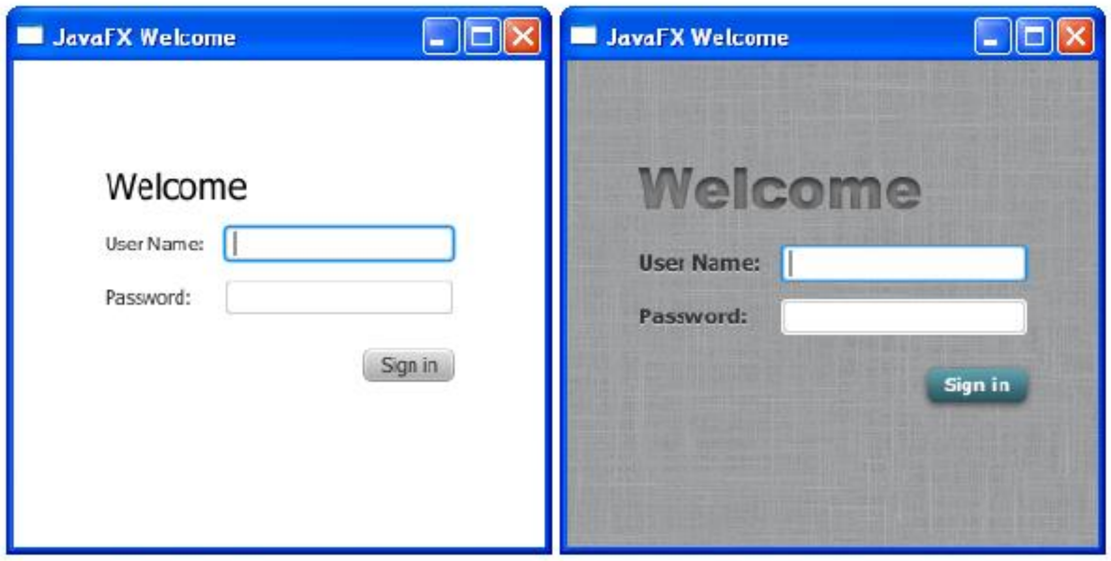
1. Setting background color of the root
2. Creating and using  a status bar
3. Setting up a ComboBox and responding to user selections with a ChangeListener
4. ToggleButton for toggling between states in response to button clicks
5. Creating multiple Stages and how communication between them is accomplished.

## 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.

## Fourth Example - 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.



# Add CSS Styling to the Login Class

Steps:

1. Create a new CSS file and save it in the same directory as Login.java.

2. 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();
```

3. 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](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");
```

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:

1. 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 );
   }
   ```

2. 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. 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. 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:

    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.

# Review of XML

1. What is XML?
   - XML stands for EXtensible Markup Language
   - XML is a markup language much like HTML
   - XML was designed to describe data, not to display data
   - XML tags are not predefined. You must define your own tags

2. Sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

   First line is a *declaration*. The tag `<note>` is called the *root*.

3. General Tree Structure:

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

4. Points about XML

   a. All XML elements must have a closing tag

| | |
|---|---|
| `<p>This is a paragraph.`<br>`<br>` | `<p>This is a paragraph.</p>`<br>`<br />` |
| illegal | correct |

   b. XML tags are case-sensitive : `<Note>` is different from `<note>`

c. Tags must be properly nested (often overlooked in older HTML documents)

| `<b><i>This text is bold and italic</b></i>` | `<b><i>This text is bold and italic</i></b>` |
|---|---|
| illegal | correct |

d. XML documents must have a root element

e. XML attributes must be quoted (often overlooked in HTML)

| <pre>&lt;note date=12/11/2007&gt;<br>  &lt;to&gt;Tove&lt;/to&gt;<br>  &lt;from&gt;Jani&lt;/from&gt;<br>&lt;/note&gt;</pre> | <pre>&lt;note date="12/11/2007"&gt;<br>  &lt;to&gt;Tove&lt;/to&gt;<br>  &lt;from&gt;Jani&lt;/from&gt;<br>&lt;/note&gt;</pre> |
|---|---|
| illegal | correct |

f. *Processing instructions* are tags beginning and ending with ? that are typically used to give instructions to an application that processes the XML. Same syntax as the XML declaration, but for a different purpose.

An example found in HTML docs to declare a CSS stylesheet is the following:

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

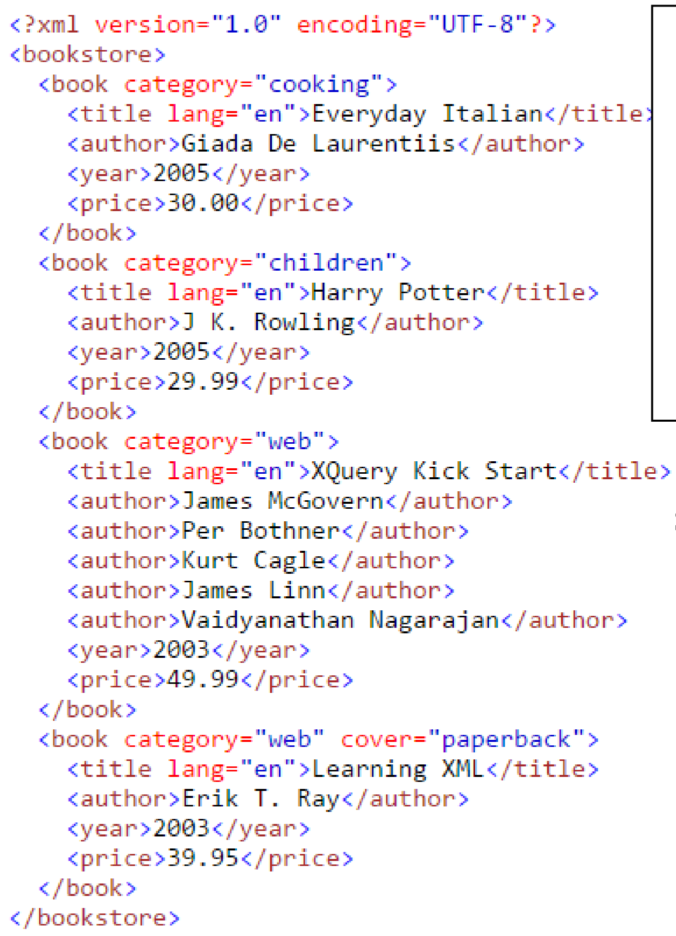
g. *Parsing XML*. There have been two main approaches for parsing XML files, both of which are supported in Java

i. Scanning each character starting at the beginning of the file (SAX parsing)

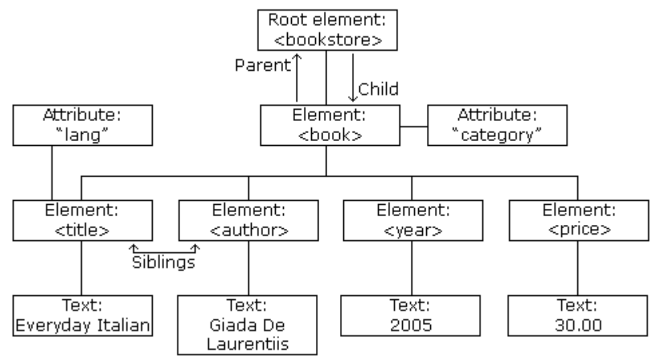
ii. Treat elements and attributes of the XML document as nodes in a tree and traverse the tree to get information about the XML content (DOM parsing)

iii. DOM = Document Object Model. JavaScript uses DOM to navigate the elements of an HTML page. We illustrate how DOM is used to read in XML files (next slide), and give a simple XML example of DOM processing in `lesson6.lecture.javafx.domprocessing`

# DOM Example



```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">XQuery Kick Start</title>
    <author>James McGovern</author>
    <author>Per Bothner</author>
    <author>Kurt Cagle</author>
    <author>James Linn</author>
    <author>Vaidyanathan Nagarajan</author>
    <year>2003</year>
    <price>49.99</price>
  </book>
  <book category="web" cover="paperback">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

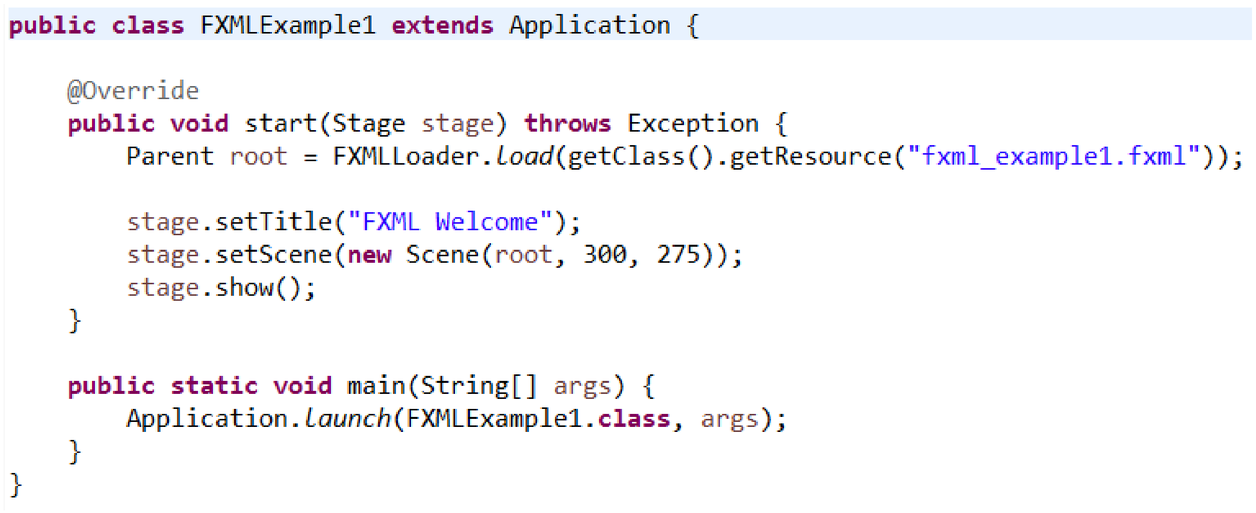


DOM Tree

See XML demo code in
`lesson6.lecture.javafx.domprocessing`

# 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:



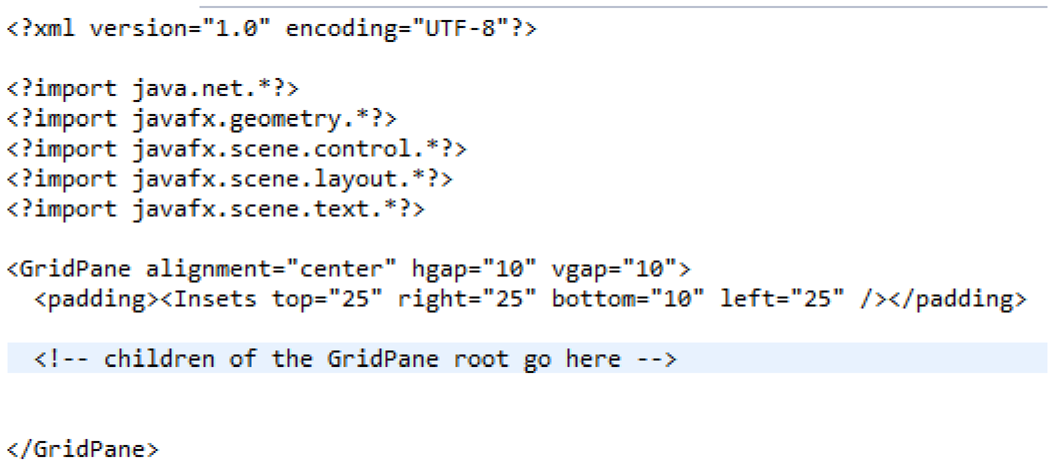
```java
public class FXMLExample1 extends Application {

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

2. 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. 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.

## CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

*The self-referral dynamics*
*arising from the reflexive association of container classes*

1. In JavaFX, components are placed and arranged in container classes for attractive display.

2. In JavaFX, certain container classes (like HBox) are also considered to be components; this makes it possible to place and arrange container classes inside other container classes. These self-referral dyanmics support a much broader range of possibilities in the design of GUIs.

3. **Transcendental Consciousness**:  TC is the self-referral field of all possibilities.

4. **Wholeness moving within Itself**: In Unity Consciousness, all activity is appreciated as the self-referral dynamics of one's own Self.