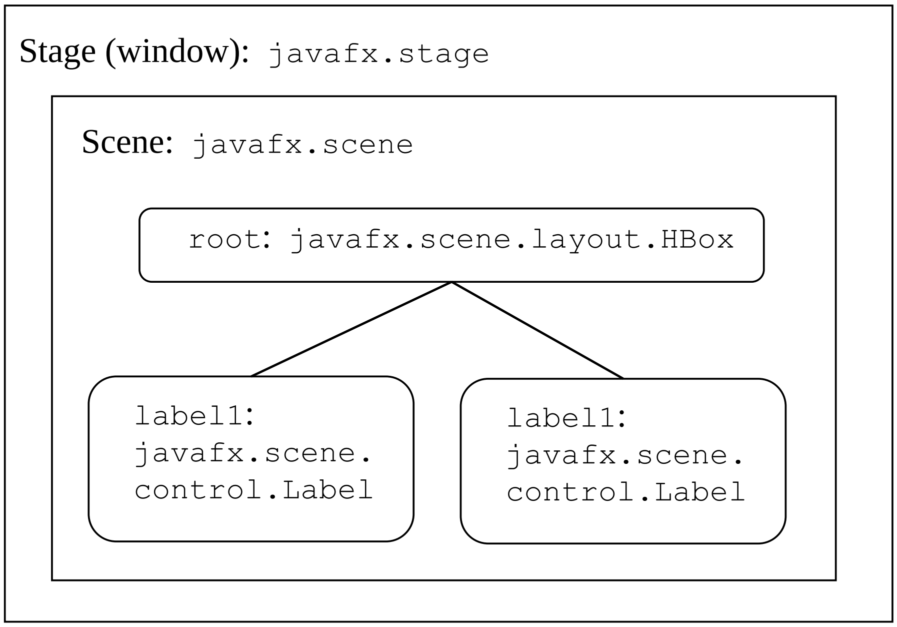
Introduction to JavaFX

JavaFX is a set of packages that allow Java programmers to create rich graphics and media applications. Potential applications include GUI interfaces, 2D and 3D games, animations, visual effects, touch-enabled applications, and multimedia applications. JavaFX 8 is the latest version. JavaFX has several advantages over other graphical libraries, including hardware-accelerated graphics and a high-performance media engine. The platform includes built-in UI controls and supports XML-based markup for the design and layout of UI components and Cascading Style Sheets for presentation. This separates the controlling code from the UI while simplifying the UI design. Most IDEs assist with many of these details in the construction of a JavaFX application. At some point JavaFX will replace Swing as the standard library for creating graphical interfaces. However, both JavaFX and Swing are expected to be included in Java distributions for the foreseeable future.

A JavaFX application uses the metaphor of a stage and scenes, just like the stage and scene of a theater. The `Stage` class is a top-level JavaFX container and in our examples will correspond to an OS window. Every `Stage` has an associated `Scene` object. The `Scene` object contains a set of nodes called a *scene graph*. These nodes describe a scene of the application, just like the script, actors, and props describe a scene in a play or movie. In JavaFX the scene graph is a hierarchical set of nodes where the *root* node is at the top of the tree. Underneath the root node we can create sub-trees consisting of layout components (e.g. panels), UI controls (e.g. buttons or textfields), shapes, or charts. Nodes have properties that includes items such as the text, size, position, or color, can be styled with CSS, generate events, and be associated with visual effects.

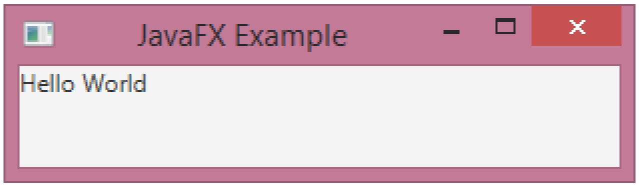
The class structure and scene graph for a simple “Hello world” JavaFX application is shown below. In this example the `Stage` contains a `Scene` that is composed of an `HBox` layout pane which simply arranges nodes in a horizontal row. Inside the `HBox` are two labels. One label displays “Hello” and the second label displays “World.”


Stage (window): javafx.stage
Scene: javafx.scene
root: javafx.scene.layout.HBox
label1: javafx.scene. control.Label
label1: javafx.scene. control.Label

Class Structure and Scene Graph for a Simple JavaFX Application

Code that implements the JavaFX structure is below. The JavaFX program must extend `Application`. The entry point is the `start` method which is invoked through the `launch` method. The JavaFX panes are similar in principle to the Swing layout classes. Other JavaFX layout panes include the `BorderPane` which is like Swing’s `BorderLayout`, `HBox` for horizontal layout of nodes, `VBox` for vertical layout of nodes, `StackPane` to place nodes within a single stack on top of previous nodes, `FlowPane` which is like Swing’s `FlowLayout`, `TilePane` which is like a tiled `FlowPane`, `GridPane` which is like Swing’s `GridLayout`, and `AnchorPane` which allows nodes to be anchored to edges of the pane.

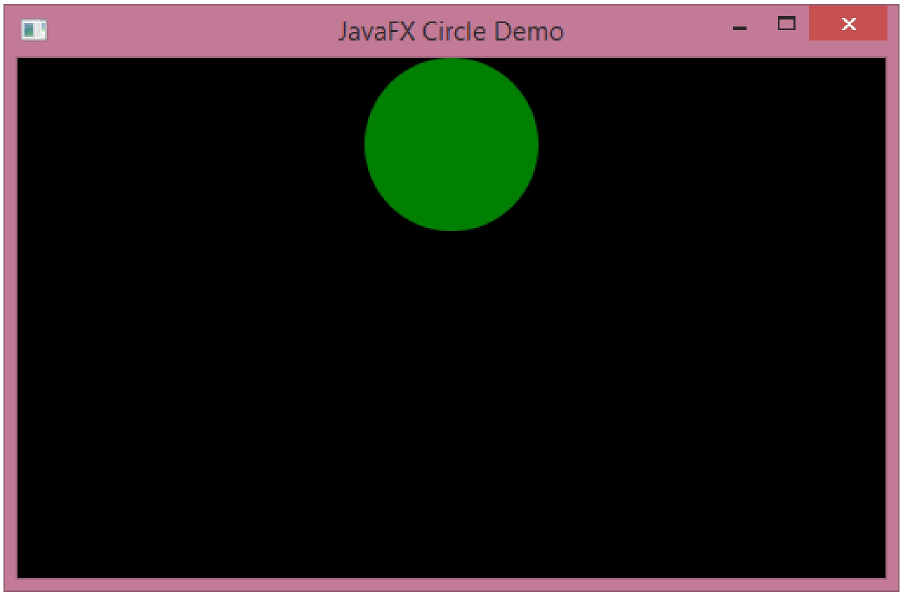
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class JavaFXHelloWorld extends Application
{
    public void start(Stage primaryStage)
    {
            Label label1 = new Label();
            Label label2 = new Label();
            label1.setText("Hello");
            label2.setText(" World");

        HBox root = new HBox();
        root.getChildren().add(label1);
        root.getChildren().add(label2);

        Scene scene = new Scene(root, 300, 50);

        primaryStage.setTitle("JavaFX Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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



Hello World

The next two programs demonstrate how JavaFX allows us to achieve impressive visual effects with a minimal amount of code through a declarative rather than procedural programming model. In a declarative program we specify what the program should do rather than the individual steps needed to achieve the end result. The program below draws a green circle on a black background. The program uses the `AnchorPane` layout with no anchors; we demonstrate anchoring a bit later.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.layout.AnchorPane;

public class JavaFXCircle extends Application
{
    public void start(Stage stage)
    {
       Circle c = new Circle(250,50,50);
       c.setFill(Color.GREEN);

       AnchorPane root = new AnchorPane();
       root.getChildren().add(c);

       Scene scene = new Scene(root, 500, 300,
                               Color.BLACK);

       stage.setTitle("JavaFX Circle Demo");
       stage.setScene(scene);
       stage.show();
    }

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



What if you wanted to animate the circle up and down while changing colors? As seen previously this is not too hard, but it's also not trivial. Ideally we need to set up a thread and redraw the circle inside the timer. JavaFX lets us do this easily by attaching transitions to the circle node. In the modified program below we have attached a fill transition from green to blue and a translation transition in the vertical dimension to y coordinate 200 starting from coordinate (250,50). JavaFX includes many other transitions, such as changing the scale, fade effect, or rotation. The parallel transition tells JavaFX to apply all of the transitions in parallel rather than sequentially. This is an example of declarative programming; we are telling JavaFX the desired end result and the library handles the sequential details to implement the instructions.

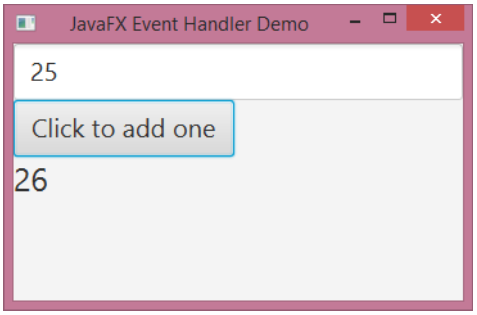
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.layout.AnchorPane;
import javafx.animation.FillTransition;
import javafx.animation.Timeline;
import javafx.animation.ParallelTransition;
import javafx.animation.TranslateTransition;
import javafx.util.Duration;

public class JavaFXCircleAnimate extends Application
{
    public void start(Stage stage)
    {
       Circle c = new Circle(250,50,50);
       c.setFill(Color.GREEN);

       AnchorPane root = new AnchorPane();
       root.getChildren().add(c);

         FillTransition fill = new
             FillTransition(Duration.millis(500));
       fill.setToValue(Color.BLUE); // Transition Blue

       TranslateTransition translate =
         new TranslateTransition(Duration.millis(500));
       translate.setToY(200);  // Move circle to Y=200

         // Run fill and translate transitions
       ParallelTransition transition = new
             ParallelTransition(c,
            fill, translate);
       transition.setCycleCount(Timeline.INDEFINITE);
       transition.setAutoReverse(true);
       transition.play();

       Scene scene = new Scene(root, 500, 300,
                               Color.BLACK);

       stage.setTitle("JavaFX Circle Demo");
       stage.setScene(scene);
       stage.show();
    }

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

*** Show in class – individual transitions first, then parallel
(constructor takes Shape as second argument after transition time)

Upon running the program above you will see the circle move vertically from the top to bottom while changing colors from green to blue. Note that the program is responsive! JavaFX handles threading so the application does not lock-up. If you add other UI controls like buttons or textboxes to the scene then they will be active while the circle is animated.

JavaFX allows the programmer to attach event handlers to UI controls in a manner similar to Swing. The code below shows how to attach an event handler to a button. When the button is clicked the value entered in the textfield is read into an integer, incremented by one, and output into the label. The program uses a `VBox` pane which vertically stacks each node that is added to the scene.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class JavaFXEvent extends Application
{
    public void start(Stage primaryStage)
    {
        TextField txt = new TextField();
        txt.setText("0");
        txt.setFont(new Font(20));

        Label lbl = new Label();
        lbl.setFont(new Font(25));

        Button btn = new Button();
        btn.setFont(new Font(20));
        btn.setText("Click to add one");

        btn.setOnAction(new EventHandler<ActionEvent>()
        {
            public void handle(ActionEvent event)
            {
                int val =
                   Integer.parseInt(txt.getText());
                val++;
                lbl.setText(Integer.toString(val));
            }
        });

        VBox root = new VBox(); // Vertical layout
        root.getChildren().add(txt);
        root.getChildren().add(btn);
        root.getChildren().add(lbl);

        Scene scene = new Scene(root, 350, 200);
        primaryStage.setTitle("JavaFX Event Handler Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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



When we cover the section on functional programming then you can simplify the event handling code by using the following lambda expression:

```
        btn.setOnAction(e ->
        {
           int val = Integer.parseInt
               (txt.getText());
           val++;
           lbl.setText
               (Integer.toString(val));;
        });
```

Building complex interfaces can be tedious and difficult to visualize when directly coding the layout panes. To assist with UI development Oracle has released the JavaFX Scene Builder. If you are using an IDE then the Scene Builder may already installed on your system. The Scene Builder can be freely downloaded from http://www.oracle.com/technetwork/java/javase/downloads/sb2download-2177776.html. The last binary version of the Scene Builder is 2.0 or you can build it from source via OpenJFX.

The Scene Builder allows the programmer or UI designer to graphically construct the interface and quickly test the layout of UI controls. When using the Scene Builder a JavaFX application will typically be split up into at least three separate files, each handling a different aspect of the program:

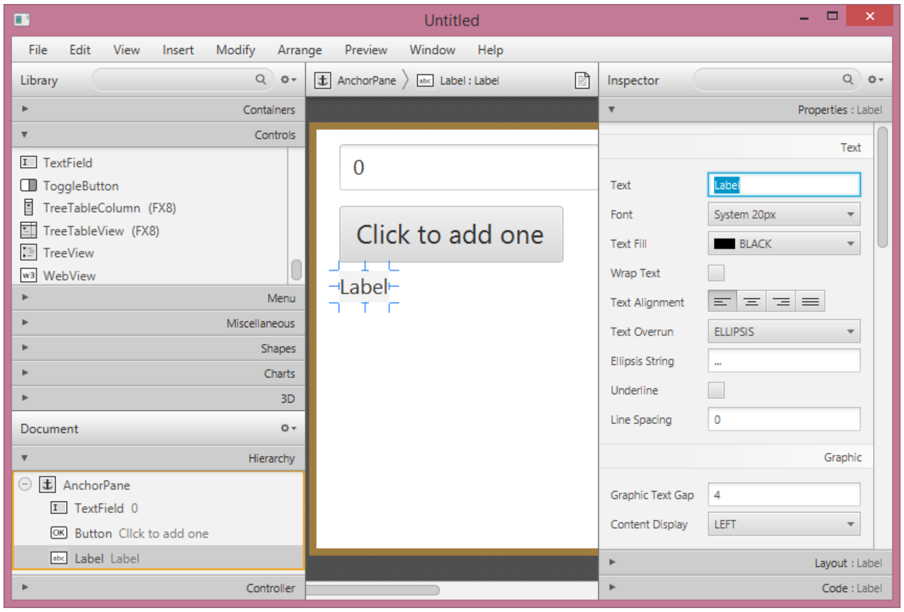
- FXML file. This is an XML file created by the Scene Builder that describes the layout of nodes in the scene. A sample FXML file similar to the previous program follows. While you could manually create the file, it is normally generated by the Scene Builder.

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.text.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>

<VBox maxHeight="-Infinity" maxWidth="-
Infinity" minHeight="-Infinity" minWidth="-
Infinity" prefHeight="200.0" prefWidth="350.0"
xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1">
   <children>
      <TextField fx:id="txt" text="0">
         <font>
            <Font size="20.0" />
         </font>
      </TextField>
      <Button fx:id="btn"
          mnemonicParsing="false"
               text="Click to add one">
         <font>
            <Font size="25.0" />
```

```
               </font>
            </Button>
            <Label fx:id="lbl" text="23">
               <font>
                  <Font size="20.0" />
               </font>
            </Label>
         </children>
      </VBox>
```

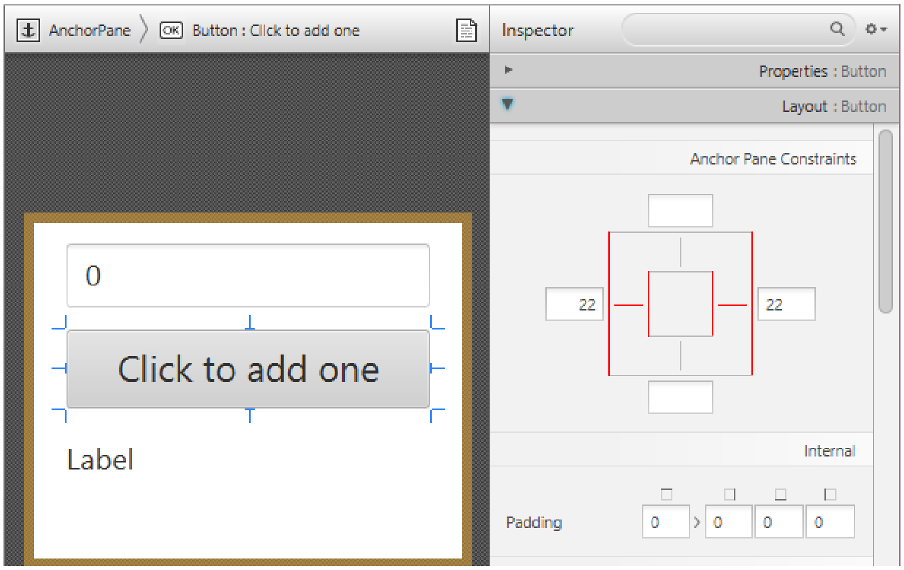
- Application file. This is the JavaFX Java source code that contains the `start` method. When used with an FXML file, the `start` method merely loads the FXML file using the `FXMLLoader` class.
- Controller file. This file contains a class that implements `javaFX.fxml.Initializable` and contains event handlers that respond to UI controls.

If you are using an IDE that includes the Scene Builder, then consult your IDE's documentation on how to create a new JavaFX FXML Application project. Otherwise, you can directly launch the Scene Builder application after downloading and installing it. The figure below shows the Scene Builder after dragging an `AnchorPane` from the "Containers" section to the middle of the window, followed by dragging a `TextField`, `Button`, and `Label` from the "Controls" section. You can select a control by either clicking it on the form or by selecting it by name under "Hierarchy" within the "Document" section on the bottom left. The latter is useful for "invisible" controls such as a label with no text. Once a control is selected you can edit properties, such as the text or font size, in the "Properties" section in the "Inspector" window on the right.

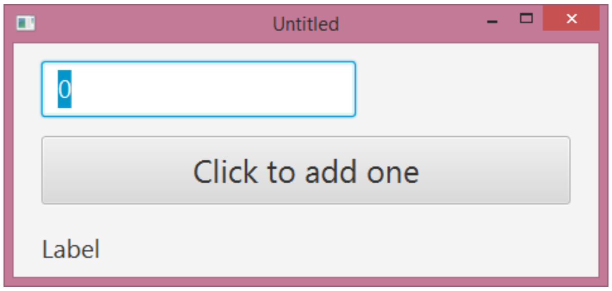


Text: Label
Font: System 20px
Text Fill: BLACK
Wrap Text
Text Alignment
Text Overrun: ELLIPSIS
Ellipsis String: ...
Underline
Line Spacing: 0
Graphic Text Gap: 4
Content Display: LEFT

Since we are using an `AnchorPane`, we can anchor sides of a control to edges of the pane. This is useful if the window is resized. For example, if we want the button to fit the entire width of the window when it is resized then we would anchor the left and right edges. This is illustrated below. The button has been selected and under the “Layout” section of the “Inspector,” anchors have been set on the left and right sides. You can see test the result using the “Preview” command from the main menu.



Anchoring a Button using Scene Builder 2.0



To load a saved FXML file created by the Scene Builder we use the `FXMLLoader` class. The code below shows how to load a FXML file named `JavaFXApp.FXML`. Since the layout details are in the FXML file, very little coding is needed in the application class.

// Loader JavaFX App

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

```
public class JavaFXApplication3 extends Application {

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

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Next we need a `Controller` class to respond to events. A class named `JavaFXAppController.java` is shown below that implements the button handler. This class implements `Initializable` and must have an `initialize` method that can be used to initialize the controller.

To link variables defined in the `JavaFXAppController` class to UI controls created in the Scene Builder, place the @FXML annotation before the variable definition. This injects the necessary values from the FXML loader.

// JavaFX Controller Class for `JavaFXApp.fxml`

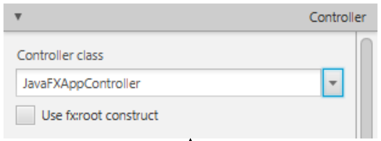
```
import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.TextField;
import javafx.scene.control.Label;
import javafx.scene.control.Button;

public class JavaFXAppController implements
Initializable
{
      // The @FXML annotation looks up the
      // corresponding ID in the FXML file so
      // these variables map to the controls in
      // the UI
```

```
    @FXML
    private Label lblNumber;
    @FXML
    private Button btnClick;
    @FXML
    private TextField txtNumber;

    @FXML
    private void handleButtonAction(ActionEvent event)
    {
      int val = Integer.parseInt
            (txtNumber.getText());
        val++;
        lblNumber.setText
                (Integer.toString(val));
    }
    public void initialize(URL url, ResourceBundle rb)
    {
        // Required by Initializable interface
        // Called to initialize a controller after the
        // root element has been processed
    }
}
```

Finally, we need to link the controller to the FXML file. Back in the Scene Builder, select the Java file containing the controller in the “Controller” section located at the bottom left side of the Scene Builder. In our example, the controller is named `JavaFXController.java`.

Next, select each UI control that has a corresponding variable defined in the controller. To link the controls, select the variable name from the “Code” section of the “Inspector” on the right. You can also select a method for an event handler. For example, in the code above we named the label variable `lblNumber`. In the Scene Builder the same name should be entered in the `fx:id` field for the label on the form.

The process to link the controller to the FXML file constructed by the Scene Builder is shown below. Once the linkages are made the Java programs can be compiled and the main application run.

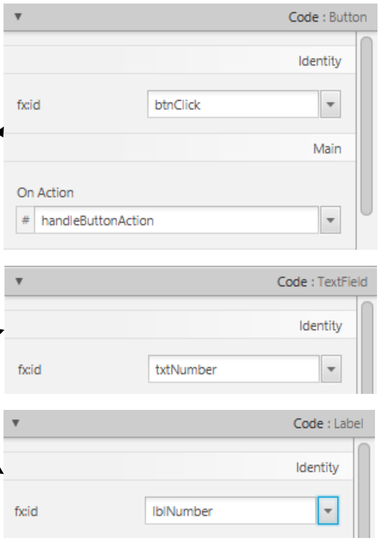

Controller

Controller class: JavaFXAppController

Use fx:root construct

1. Set controller class from the menu on bottom left

2. Select each UI control and then from the Inspector/Code section on the right, set fx:id to the corresponding name in the controller, and set the method to handle an event


Code : Button

Identity

fx:id: btnClick

Main

On Action: # handleButtonAction

Code : TextField

Identity

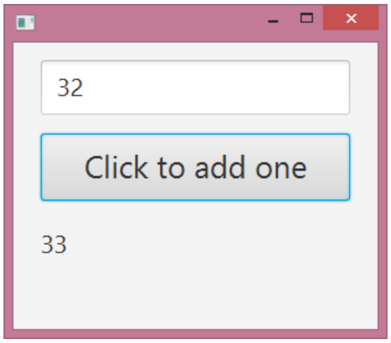
fx:id: txtNumber

Code : Label

Identity

fx:id: lblNumber

After compiling and running JavaFXApp


32

Click to add one

33

In this section we have presented only a small sip of what JavaFX can do. JavaFX provides a structure and APIs for visual effects, animation, graphics, media, and the construction of graphical user interfaces. In addition, JavaFX supports declarative programming, separates controlling code from display code through FXML, and offers a Scene Builder application to assist with the construction of complex user interfaces. For additional reading about JavaFX visit the Oracle JavaFX overview page at http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html and the JavaFX documentation website at http://docs.oracle.com/javase/8/javase-clienttechnologies.htm.