JavaFX Intro

The methods used to display graphics and graphical user interfaces (GUI) in Java have gone through several evolutions since Java’s introduction in 1996. The first toolkit to display GUIs in Java was the Abstract Window Toolkit, or AWT. AWT was implemented using platform-specific code. The successor to AWT is Swing. Swing is written in Java which provides platform independence. Swing is complementary to AWT rather than a replacement. A typical Java programs written using Swing would incorporate libraries from both AWT and Swing. While there are still many Java programs written today using Swing, the most recent graphics toolkit for Java is 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. At the time of this writing JavaFX 8 is the latest version. JavaFX has several advantages over other graphical libraries, including hardware-accelerated graphics and a high-performance media engine. 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.

Due to the historical progression from AWT to Swing to JavaFX, you may find it helpful to learn a bit about AWT and Swing. Sometimes you will see references to the older toolkits in the context of a newer toolkit. Swing is covered in the online chapter.

A Sample JavaFX Application
Listing 1.2 contains a JavaFX application that draws a happy face. Let’s examine the code by going through it section by section.

Listing 1.2 Drawing a Happy Face

```java
import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.shape.ArcType;

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

    @Override
    public void start(Stage primaryStage) throws Exception {
        Group root = new Group();
```
Scene scene = new Scene(root);

Canvas canvas = new Canvas(400, 300);
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.strokeOval(100, 50, 200, 200);
gc.fillOval(155, 100, 10, 20);
gc.fillOval(230, 100, 10, 20);
gc.strokeArc(150, 160, 100, 50, 180, 180, ArcType.OPEN);

root.getChildren().add(canvas);
primaryStage.setTitle("HappyFace in JavaFX");
primaryStage.setScene(scene);
primaryStage.show();
}

Program Output

![HappyFace in JavaFX](image)

The section

```java
import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.shape.ArcType;
```

says that the application uses a number of library packages in the JavaFX library. They include classes for Application, Canvas, Scene, Group, Stage, GraphicsContext, and ArcType. These are all components of JavaFX that will be described in more detail later in the book.
The line

```java
public class HappyFace extends Application
```

begins the class definition for the program. It is named `HappyFace`. The words `extends Application` indicate that we are defining a JavaFX application, as opposed to some other kind of class. Although you need not worry about further details yet, we are using inheritance to create the class `HappyFace` based upon an existing class `Application`.

The application contains two methods – `main` and `start`. The `main` method is where a Java program normally begins.

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

In the sample program shown in Listing 1.1 most of our program code was entered in the `main` method. This is how we will write most of our Java programs. However, a JavaFX application is different. A JavaFX program begins execution in the `start` method. The `main` method is ignored in a correctly deployed JavaFX application. However, it is common to include `main` and a call to `launch` as a fallback, which will end up launching the JavaFX program and the `start` method.

For a JavaFX application, programs begin in the `start` method.

```java
@Override
public void start(Stage primaryStage) throws Exception
```

For now, you can ignore the `@Override` and the `throws Exception` code. What you do need to know is that this method is invoked automatically when the JavaFX application is run. JavaFX uses the metaphor of a stage and scenes, just like the stage and scene of a theater.

The next four lines sets up a canvas on a scene for you to draw simple graphics.

```java
Group root = new Group();
Scene scene = new Scene(root);
Canvas canvas = new Canvas(400, 300);
GraphicsContext gc = canvas.getGraphicsContext2D();
```

At this point we can now use drawing operations on the canvas. The method invocation

```java
gc.strokeOval(100, 50, 200, 200);
```

draws the big circle that forms the outline of the face. The first two numbers tell where on the screen the circle is drawn. The method `strokeOval`, as you may have guessed, draws ovals. The last two numbers give the width and height of the oval. To obtain a circle, you make the width and height the same size, as we have done here. The units for these numbers are called `pixels`, and we will describe them shortly.

The two method invocations

```java
gc.fillOval(155, 100, 10, 20);
gc.fillOval(230, 100, 10, 20);
```
draw the two eyes. The eyes are “real” ovals that are taller than they are wide. Also notice that the method is called `fillOval`, not `strokeOval`, which means it draws an oval that is filled in.

The next invocation

```java
gc.strokeArc(150, 160, 100, 50, 180, 180, ArcType.OPEN);
```

draws the mouth. We will explain the meaning of all these arguments in the next section.

Finally, the block

```java
root.getChildren().add(canvas);
primaryStage.setTitle("HappyFace in JavaFX");
primaryStage.setScene(scene);
primaryStage.show();
```

sets a title for the window and does some bookkeeping to set the stage and display the window.

### Adding Labels to a JavaFX Application

A label in a JavaFX application is little more than a quoted string, but it is handled in the same way that many other components, such as buttons, are handled. Thus, labels provide another way to add text to a JavaFX application, but serve as an introduction to expand to other types of components. We will add other components in the same way that we add labels.

Listing 5.25 contains a JavaFX application that displays the text `Hello Out there!`, but this program does not use `fillText` to create the displayed text; it uses labels instead. To access the label class we import `javafx.scene.control.Label`. In our program we add two labels to the program. We could have added only one label containing all the text, but we wanted an example of adding more than one component to the program.

Typically, you will add multiple items to a JavaFX application. When you do so, you need to specify how the components should be arranged visually in the window. In Listing 5.25 we did this using a vertical box layout. We access this class by importing `javafx.scene.layout.VBox`. Then in the code we use:

```java
VBox root = new VBox();
```

This statement says to create a vertical box layout to arrange our components vertically, one on top of the other. The first component we add to the layout will be displayed at the top. The second component will be displayed beneath it, and so on. There are other layouts we will discuss later in the text, but this is a simple layout to begin with.

Labels are created and added to the scene in two separate steps. You create a label as an object of the class `Label`, as illustrated in these lines:

```java
Label label1, label2;
label1 = new Label("Hello");
```

We can optionally set the font or other properties for the individual label, as in:
labell.setFont(Font.font("Times New Roman", 24));

Finally, the label is added to the layout with:

root.getChildren().add(label1);

LISTING 5.25   Adding Labels to a JavaFX Application

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;

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

    @Override
    public void start(Stage primaryStage) throws Exception
    {
        VBox root = new VBox();
        Label label1, label2;
        label1 = new Label("Hello");
        label1.setFont(Font.font("Times New Roman", 24));
        label2 = new Label("Out there!");
        label2.setFont(Font.font("Courier New", 36));
        root.getChildren().add(label1);
        root.getChildren().add(label2);

        Scene scene = new Scene(root, 300, 100);
        primaryStage.setTitle("Label Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

Program Output

Hello
Out there!
Event-Driven Programming

In this section we explain how to assign actions to a control like a button. JavaFX applications use events and event handlers. An event is an object that represents some user action that elicits a response, such as clicking a button with the mouse. When an object generates an event, it is said to fire the event. For example, when the user clicks a button, the button fires an event. In a JavaFX application, every object that can fire events can have one or more listener objects that receive events automatically. A listener object has methods that specify what will happen when the listener receives events of various kinds. These methods are called event handlers.

You the programmer specify what objects are the listener objects for any given object that might fire an event. You also define the event-handler methods. The relationship between an event-firing object, such as a button, and its event-handling listener is illustrated in Figure 8.6.

![Figure 8.6 Event Firing and an Event Listener](image)

We actually haven’t presented anything new. Event objects, listeners, and event handlers are all implemented using a Java interface as discussed in section 8.4. In our case, the interface is named EventHandler. This interface requires us to implement the method named handler. The listener object is the class that implements EventHandler. The event object will be an object of class ActionEvent. It is created by JavaFX when an event occurs, such as a button is clicked. This object contains information about the type of event that occurred and the object that initiated the event (e.g., the button that was clicked). The process is diagrammed in Figure 8.7.
Next we present three options to select the class that will be the listener. Regardless of the option, the listener class must implement `EventHandler<ActionEvent>`. The `<ActionEvent>` notation is discussed in Chapter 12 and essentially says that we are making an event handler that will handle items of type `ActionEvent`. We can make our class in a new, separate class, whose sole purpose is to handle the event. This is the “normal” technique we have been using to create a class that implements an interface. An alternate technique is to make the main GUI class itself implement the interface. Finally, a third alternative technique is to create an anonymous inner class.

**Event Handling in a Separate Class**

Listing 8.20 creates a simple JavaFX application with two buttons in a VBox layout pane. The listener object for the buttons is the class `HandleButtonClick` which is provided in Listing 8.21. To associate the button click event with a `HandleButtonClick` object we use the `setOnAction` method. The following example creates a new `HandleButtonClick` object and associates it with `btnSunny`:

```java
HandleButtonClick clickEvent = new HandleButtonClick();
btnSunny.setOnAction(clickEvent);
```

This is telling JavaFX that the `HandleButtonClick` class will handle any processing when `btnSunny` is clicked.

Since `HandleButtonClick` is a regular class, we can overload the constructor. In Listing 8.21 we have created a second constructor that takes a `String` message and stores it internally so that the message will be output when the event occurs. We can create an instance of `HandleButtonClick` using this constructor and associate it with a button all in one line:

```java
btnCloudy.setOnAction(new HandleButtonClick("It is cloudy."));
```
Note that `HandleButtonClick` is not just any class. The class we use to set the action must implement the interface `EventHandler<ActionEvent>`. This interface requires that you override the method

```java
public void handle(ActionEvent event)
```

In Listing 8.21 our implementation of the `handle` method simply outputs the message variable, which is “It is sunny!” by default. When the application is run, clicking the “Sunny” button will output “It is sunny!” to the console while clicking the “Cloudy” button will output “It is cloudy.” to the console.

**LISTING 8.20  Event Handling in a Separate Class**

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.VBox;
import javafx.scene.control.Button;

/**
 * Simple demonstration of programming buttons in a JavaFX application.
 * This version outputs a message when clicked.
 */
public class ButtonDemo extends Application {

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

    @Override
    public void start(Stage primaryStage) throws Exception {
        VBox root = new VBox();
        Button btnSunny;
        Button btnCloudy;
        btnSunny = new Button("Sunny");
        btnCloudy = new Button("Cloudy");

        // Create an event object to handle the button click.
        // The "handle" method in HandleButtonClick will be invoked when the button is clicked.
        HandleButtonClick clickEvent = new HandleButtonClick();
        btnSunny.setOnAction(clickEvent);

        // We can also create the HandleButtonClick object without
        // a named reference by creating it inside the call to setOnAction
        btnCloudy.setOnAction(new HandleButtonClick("It is cloudy.");

        root.getChildren().add(btnSunny);
        root.getChildren().add(btnCloudy);

        Scene scene = new Scene(root, 300, 100);
        primaryStage.setTitle("Button Event Handling Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;

/**
 * This class handles a button click and outputs a message.
 * The handle method is invoked when the button is clicked.
 */
public class HandleButtonClick implements EventHandler<ActionEvent> {
    private String message;
    public HandleButtonClick() {
        message = "It is sunny!";
    }
    public HandleButtonClick(String customMessage) {
        message = customMessage;
    }
    @Override
    public void handle(ActionEvent event) {
        System.out.println(message);
    }
}

Event Handling in the Main GUI Application Class

One minor inconvenience to the approach in Listing 8.20 is that we have to go through some overhead to define a class to handle our button click event. Another inconvenience is that the HandleButtonClick class doesn’t have access to any of the member variables in the ButtonDemo1 class. It is common for a control to need to interact with other controls in the application. We could work around this problem by passing in a reference to other controls we may need to access in the constructor to HandleButtonClick, but this is also somewhat inconvenient if we have to send a large number of arguments.

Another approach is to make the HandleButtonClick a private inner class within the ButtonDemo1 class. This will allow the HandleButtonClick class to access member variables in
ButtonDemo1. To do this we simply move the HandleButtonClick class inside ButtonDemo1, at the same level as the methods in ButtonDemo1, and change public to private. Inner classes are described in Chapter 12.

An alternate approach is to make the ButtonDemo class itself implement the EventHandler<ActionEvent> interface. This means that we will need to add the handle method into the ButtonDemo class. Since this method is inside ButtonDemo it will have access to any member variables that are part of the class, such as other UI controls.

In Listing 8.22 we have taken this approach and modified ButtonDemo1 into ButtonDemo2:

```java
public class ButtonDemo2 extends Application implements EventHandler<ActionEvent>
{
    private Button btnSunny;
    private Button btnCloudy;

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

This does bring up a new complication. Since we now only have a single object (the current object) to handle both button clicks, if we want to distinguish which button was clicked we now have to make that determination within the handle method. In our previous version we could distinguish which button was clicked by associating different instances of HandleButtonClick to each button. One way to do this is to invoke the getText() method, which retrieves the text in the button. We can then check the text value and act accordingly.

In Listing 8.22 the handle method disables the other button when one is clicked. To use the getText() method we typecast back to a Button object. For a safety check, we only perform these steps if the source object that caused the event is a Button object. We can get the source object from the event argument.

LISTING 8.22  Event Handling in the Main GUI Application Class

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.VBox;
import javafx.scene.control.Button;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public class ButtonDemo2 extends Application implements EventHandler<ActionEvent>
{
    private Button btnSunny;
    private Button btnCloudy;

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

@Override
public void handle(ActionEvent event)
{
    // This method can access the member variables
    // which reference the other GUI controls
    if (event.getSource() instanceof Button)
    {
        Button btnClicked = (Button) event.getSource();
        if (btnClicked.getText().equals("Sunny"))
        {
            // Disable the cloudy button if sunny clicked
            btnCloudy.setDisable(true);
        }
        else if (btnClicked.getText().equals("Cloudy"))
        {
            // Disable the sunny button if cloudy clicked
            btnSunny.setDisable(true);
        }
    }
}

@Override
public void start(Stage primaryStage) throws Exception
{
    VBox root = new VBox();
    btnSunny = new Button("Sunny");
    btnCloudy = new Button("Cloudy");

    btnSunny.setOnAction(this);
    btnCloudy.setOnAction(this);

    root.getChildren().add(btnSunny);
    root.getChildren().add(btnCloudy);

    Scene scene = new Scene(root, 300, 100);
    primaryStage.setTitle("Button Demo 2");
    primaryStage.setScene(scene);
    primaryStage.show();
}

Program Output
Event Handling in an Anonymous Inner Class

Finally, we show one more way to make an event handler. This technique has the benefit of giving the event handler access to any member variables in the main GUI class but also allows us to create different instances of the listener object. The technique is to create what is called an anonymous inner class. This is a class with no name that we declare and instantiate at the same time. It can be used to quickly create a class with methods that override any parent methods that we are required to implement for an interface.

The format to create an anonymous inner class that implements interface EventHandler<ActionEvent> is:

```
new EventHandler<ActionEvent>() {
   // Define any member variables or methods for the class
}
```

We can plug this into the setOnAction method where we flesh out the handle method. In this example, we change the text in a label in the ButtonDemo class to “It is sunny!”.

```
btnSunny.setOnAction(new EventHandler<ActionEvent>() {
   @Override
   public void handle(ActionEvent event) {
      lblMessage.setText("It is sunny!");
   }
});
```

Listing 8.23 illustrates this technique by creating separate anonymous inner classes for the two buttons. The text in the label is changed depending upon which button is clicked.
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.layout.VBox;
import javafx.scene.control.Button;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.control.Label;

/**
Event handling with an anonymous inner class.
*/
public class ButtonDemo extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        VBox root = new VBox();
        Button btnSunny;
        Button btnCloudy;
        Label lblMessage;
        btnSunny = new Button("Sunny");
        btnCloudy = new Button("Cloudy");
        lblMessage = new Label("Click a button.");

        // Create an anonymous inner class to handle btnSunny
        btnSunny.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                lblMessage.setText("It is sunny!");
            }
        });

        // Create an anonymous inner class to handle btnCloudy
        btnCloudy.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                lblMessage.setText("It is cloudy!");
            }
        });

        root.getChildren().add(btnSunny);
    }
}
Lambda Functions and Event Handlers

Functional programming with lambda expressions was introduced in Java 8. A lambda expression is a nameless function. In functional programming, a function is the same thing as a method. Related concepts include closures, anonymous functions, and function literals. As a nameless function, a lambda expression is essentially a little chunk of code that you can pass around as data but have it treated like a function with parameters. Lambda expressions provide a neat way to implement a class that normally has only one function and to make it easy to modify methods on the spot rather than go through the work of defining a method to perform a specialized task. Additionally, lambda expressions help Java parallelize itself to run more efficiently on multi-core or parallel machines.

The format to define a lambda expression looks like this:

```
parameters -> body
```

The arrow separates the parameters from the body. In many cases the body is short and just a single line of code. If it were longer, than a traditional method may make more sense. Here is a lambda expression with a function that takes no parameters and returns the number 44:

```
() -> { return 44; }
```

Here is a lambda expression that returns the sum of two integers x and y:

```
(int x, int y) -> { return (x+y); }
```

In many cases Java can infer the type of the parameters, in which case we can leave the data type off. We can also simply provide an expression on the right side and it automatically becomes the return value without requiring the keyword return. The following is equivalent to the previous example:

```
(x, y) -> x+y
```
As an example to motivate the use of lambda functions, consider the anonymous inner class that we wrote for the button handlers of Listing 8.23. The handler for btnSunny was:

```java
btnSunny.setOnAction(new EventHandler<ActionEvent>(){
    @Override
    public void handle(ActionEvent event){
        lblMessage.setText("It is sunny!");
    }
});
```

In this case we need to create a new class to implement `EventHandler<ActionEvent>` and define the `handle` method. We can do the same thing in a much more compact format using lambda functions. The equivalent event handler becomes:

```java
btnSunny.setOnAction(e -> {
    lblMessage.setText("It is sunny!");
});
```

We even don't need to specify the `ActionEvent` data type for argument `e` because Java can infer it from the only type valid from the context from which we are calling `setOnAction`. The lambda format is the simplest of all and lets us directly insert the method where needed. Listing 11.10 is the JavaFX program from Listing 8.23 converted to use lambda functions.

**LISTING 11.10  Handling Events with Lambda Functions**

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

/**
 Event handling with lambda functions. This program implements Listing 8.23 using lambda functions.
 */
public class ButtonDemoLambda extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        VBox root = new VBox();
    }
}
```
Button btnSunny;
Button btnCloudy;
Label lblMessage;
btnSunny = new Button("Sunny");
btnCloudy = new Button("Cloudy");
lblMessage = new Label("Click a button.");

btnSunny.setOnAction(e ->
{
    lblMessage.setText("It is sunny!");
});
btnCloudy.setOnAction(e ->
{
    lblMessage.setText("It is cloudy!");
});

root.getChildren().add(btnSunny);
root.getChildren().add(btnCloudy);
root.getChildren().add(lblMessage);

Scene scene = new Scene(root, 300, 100);
primaryStage.setTitle("Lambda Button Demo");
primaryStage.setScene(scene);
primaryStage.show();

Program Output

The program output is identical to that of Listing 8.23 except for the window title.

These examples should give you an idea of what Java lambda expressions look like and what they can do. While there is definitely a learning curve, lambda expressions will allow you to write code that is more concise while enabling parallel processing. Java 8's new syntax supports both functional programming and object-oriented programming in a way that reaps the benefits of both styles.

Building JavaFX Applications with the Scene Builder

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. Oracle no longer releases binary executables of the Scene Builder (you have to build it from source code) but free binaries can be downloaded from Gluon Labs at http://gluonhq.com/labs/scene-builder. Consult your IDE's documentation if configuration is needed to integrate the JavaFX Scene Builder. At the time of this writing, the latest version of the Gluon Scene Builder is 8.2.
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 with a text field, button, and label inside a VBox follows. While you could manually create the file, it is normally generated by the Scene Builder.

```xml
<?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. Figure 12.13 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.

FIGURE 12.13  UI Design with Scene Builder

The AnchorPane allows us to 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 in Figure 12.14. 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.
To load a saved FXML file created by the Scene Builder we use the `FXMLLoader` class. Listing 12.14 shows how to load a FXML file named `FXMLDocument.fxml`. Since the layout details are in the FXML file, very little coding is needed in the application class.

**LISTING 12.14  JavaFX Application Class for FXMLDocument.fxml**

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

public class JavaFXApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
    }
```
Next we need a Controller class to respond to events. A class named JavaFXAppController.java is shown in Listing 12.15 that implements a button handler to read the number entered in the text field, increment it by one, and output the result in the label. 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.

Listing 12.15  JavaFX Controller Class for JavaFXApp.fxml

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

public class JavaFXAppController implements Initializable {

    @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));
    }
}
```
```java
@Override
class 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 Listing 12.15 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.

Figure 12.15 depicts the process to link the controller of Listing 12.15 to the FXML file constructed by the Scene Builder. Once the linkages are made the Java programs can be compiled and the main application run to produce output such as that shown in the Program Output of Figure 12.15.

**FIGURE 12.15  Linking the Controller in the Scene Builder**

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
SELF-TEST QUESTIONS

31. What is the purpose of the FXML file?

32. What do you add to the controller to link an instance variable to the corresponding control in the FXML file?

Where to Go From here

In this book 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.