JavaFX

Getting Started with JavaFX

Release 2.2.40

E20473-09

September 2013

Get started with JavaFX 2 by creating simple applications that introduce you to layouts, CSS, FXML, visual effects, animation, and deployment.

JavaFX Getting Started with JavaFX, Release 2.2.40

E20473-09

Primary Author: Gail Chappell

Contributing Author: Jasper Potts, Nancy Hildebrandt

Contents

Part I

About This Tutorial

This collection of tutorials is designed to get you started with common JavaFX tasks, including working with layouts, controls, style sheets, and visual effects.

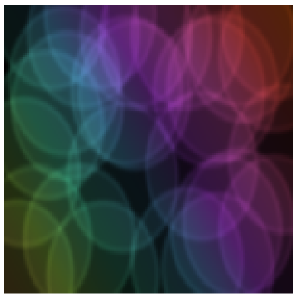
Hello World, JavaFX Style

Form Design in JavaFX

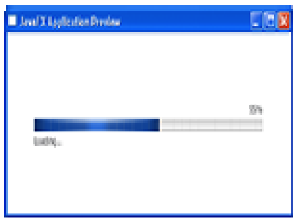
Fancy Design with CSS

User Interface Design with FXML

Animated Shapes and Visual Effects



Deployment Quickstart



1

Hello World, JavaFX Style

The best way to teach you what it is like to create and build a JavaFX application is with a “Hello World” application. An added benefit of this tutorial is that it enables you to test that your JavaFX technology is properly installed.

The tool used in this tutorial is NetBeans IDE 7.3. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 2. See the System Requirements for details.

Construct the Application

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX Application**. Click **Next**.
3. Name the project **HelloWorld** and click **Finish**.

   NetBeans opens the `HelloWorld.java` file and populates it with the code for a basic Hello World application, as shown in Example 1–1.

Example 1–1 Hello World

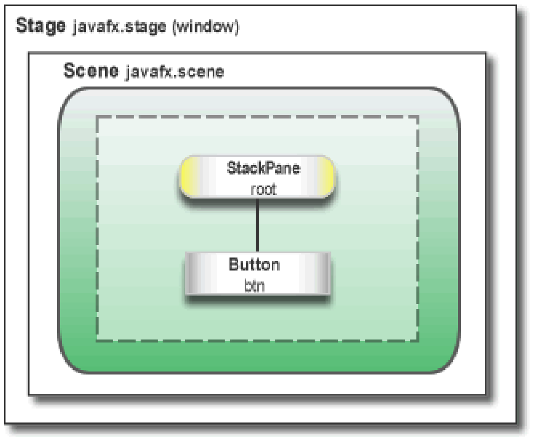
```
package helloworld;

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

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();
    }
}
```

Here are the important things to know about the basic structure of a JavaFX application:

- The main class for a JavaFX application extends the `javafx.application.Application` class. The `start()` method is the main entry point for all JavaFX applications.
- 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. Example 1–1 creates the stage and scene and makes the scene visible in a given pixel size.
- 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.
- 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.
- The `main()` method is not required for JavaFX applications when the JAR file for the application is created with the JavaFX Packager tool, which embeds the JavaFX Launcher in the JAR file. However, it is useful to include the `main()` method so 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. Also, Swing applications that embed JavaFX code require the `main()` method.

Figure 1–1 shows the scene graph for the Hello World application. For more information on scene graphs see Working with the JavaFX Scene Graph.

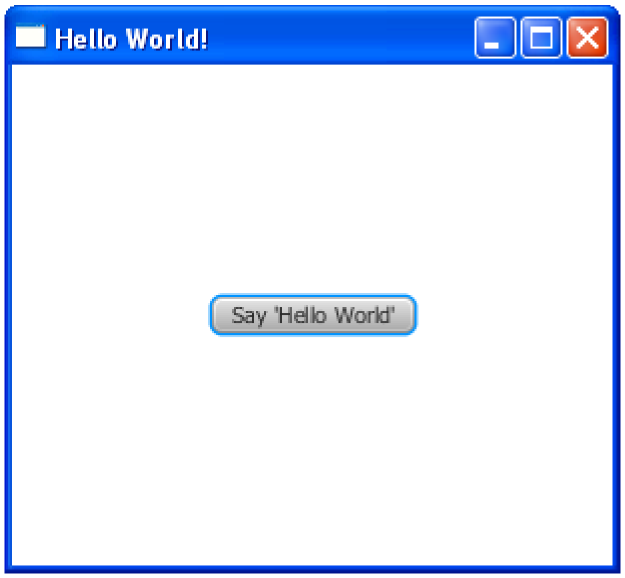
Figure 1–1 Hello World Scene Graph



Run the Application

1. In the Projects window, right-click the **HelloWorld** project node and choose **Run**.
2. Click the Say Hello World button.
3. Verify that the text "Hello World!" is printed to the NetBeans output window. Figure 1–2 shows the Hello World application, JavaFX style.

Figure 1–2 Hello World, JavaFX style



Where to Go Next

This concludes the basic Hello World tutorial, but continue reading for more lessons on developing JavaFX applications:

- Creating a Form in JavaFX teaches the basics of screen layout, how to add controls to a layout, and how to create input events.
- Fancy Forms with JavaFX CSS provides simple style tricks for enhancing your application, including adding a background image and styling buttons and text.
- Using FXML to Create a User Interface shows an alternate method for creating the login user interface. FXML is an XML-based language that provides the structure for building a user interface separate from the application logic of your code.
- Animation and Visual Effects in JavaFX shows how to bring an application to life by adding timeline animation and blend effects.
- Deploying Your First JavaFX Application describes how to run your application outside NetBeans IDE.

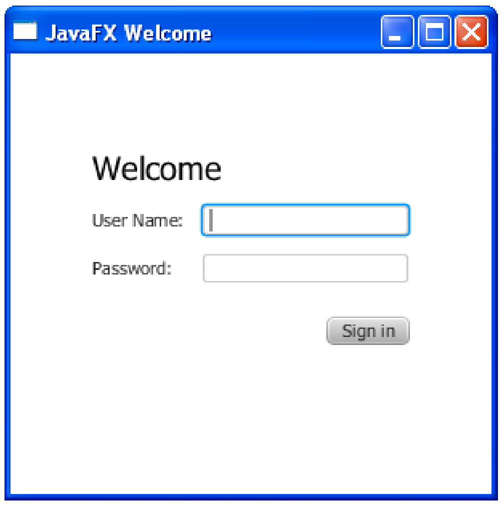
2

Creating a Form in JavaFX

Creating a form is a common activity when developing an application. This tutorial teaches you the basics of screen layout, how to add controls to a layout pane, and how to create input events.

In this tutorial, you will use JavaFX to build the login form shown in Figure 2–1.

Figure 2–1 Login Form



The tool used in this Getting Started tutorial is NetBeans IDE. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 2. See the System Requirements for details.

Create the Project

Your first task is to create a JavaFX project in NetBeans IDE and name it Login:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX Application**. Click **Next**.
3. Name the project **Login** and click **Finish**.

   When you create a JavaFX project, NetBeans IDE provides a Hello World application as a starting point, which you have already seen if you followed the Hello World tutorial.

4. Remove the `start()` method that NetBeans IDE generated and replace it with the code in Example 2–1.

Example 2–1 Application Stage

```
@Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Welcome");

        primaryStage.show();
    }
```

Tip: After you add sample code into a NetBeans project, press Ctrl (or Cmd) + Shift + I to import the required packages. When there is a choice of import statements, choose the one that starts with `javafx`.

Create a GridPane Layout

For the login form, use a `GridPane` layout because it enables you to create a flexible grid of rows and columns in which to lay out controls. You can place controls in any cell in the grid, and you can make controls span cells as needed.

The code to create the `GridPane` layout is in Example 2–2. Add the code before the line `primaryStage.show();`

Example 2–2 GridPane with Gap and Padding Properties

```
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);
```

Example 2–2 creates a `GridPane` object and assigns it to the variable named `grid`. The alignment property changes the default position of the grid from the top left of the scene to the center. The gap properties manage the spacing between the rows and columns, while the padding property manages the space around the edges of the grid pane. The insets are in the order of top, right, bottom, and left. In this example, there are `25` pixels of padding on each side.

The scene is created with the grid pane as the root node, which is a common practice when working with layout containers. Thus, as the window is resized, the nodes within the grid pane are resized according to their layout constraints. In this example, the grid pane remains in the center when you grow or shrink the window. The padding properties ensure there is a padding around the grid pane when you make the window smaller.

This 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

Looking at Figure 2–1, you can see that the form requires the title “Welcome “and text and password fields for gathering information from the user. The code for creating these controls is in Example 2–3. Add this code after the line that sets the grid padding property.

Example 2–3 Controls

```
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);
```

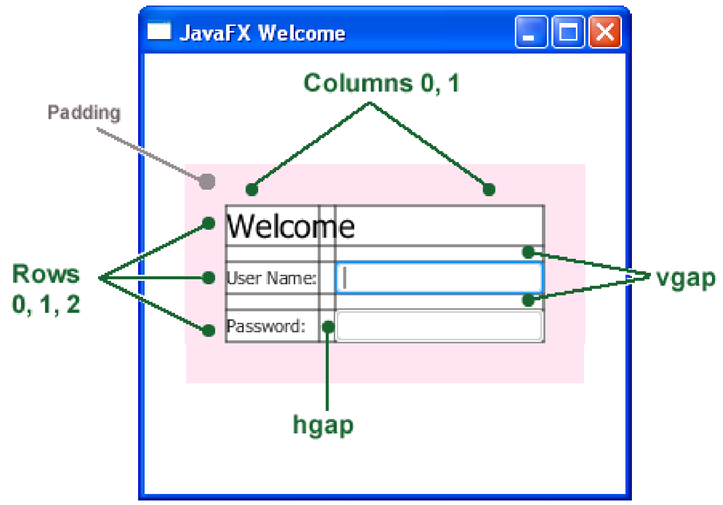
The first line creates a `Text` object that cannot be edited, sets the text to `Welcome`, and assigns it to a variable named `scenetitle`. The next line uses the `setFont()` method to set the font family, weight, and size of the `scenetitle` variable. Using an inline style is appropriate where the style is bound to a variable, but a better technique for styling the elements of your user interface is by using a style sheet. In the next tutorial, Fancy Forms with JavaFX CSS, you will replace the inline style with a style sheet.

The `grid.add()` method adds the `scenetitle` variable to the layout `grid`. The numbering for columns and rows in the grid starts at zero, and `scenetitle` is added in column 0, row 0. The last two arguments of the `grid.add()` method set the column span to 2 and the row span to 1.

The next lines create a `Label` object with text `User Name` at column 0, row 1 and a `Text Field` object that can be edited. The text field is added to the grid pane at column 1, row 1. A password field and label are created and added to the grid pane in a similar fashion.

When working with a grid pane, you can display the grid lines, which is useful for debugging purposes. In this case, you can add `grid.setGridLinesVisible(true)` after the line that adds the password field. Then, when you run the application, you see the lines for the grid columns and rows as well as the gap properties, as shown in Figure 2–2.

Figure 2–2 Login Form with Grid Lines



Add a Button and Text

The final two controls required for the application are a `Button` control for submitting the data and a `Text` control for displaying a message when the user presses the button.

First, create the button and position it on the bottom right, which is a common placement for buttons that perform an action affecting the entire form. The code is in Example 2–4. Add this code before the code for the scene.

Example 2–4 Button

```
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 first line creates a button named `btn` with the label `Sign in`, and the second line creates an `HBox` layout pane named `hbBtn` with spacing of `10` pixels. The `HBox` pane sets an alignment for the button that is different from the alignment applied to the other controls in the grid pane. The `alignment` property has a value of `Pos.BOTTOM_RIGHT`, which positions a node at the bottom of the space vertically and at the right edge of the space horizontally. The button is added as a child of the `HBox` pane, and the `HBox` pane is added to the grid in column 1, row 4.

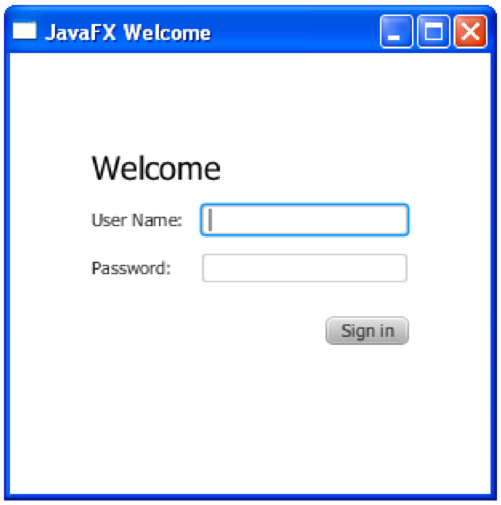
Now, add a `Text` control for displaying the message, as shown in Example 2–5. Add this code before the code for the scene.

Example 2–5 Text

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

Figure 2–3 shows the form now. You will not see the text message until you work through the next section of the tutorial, Add Code to Handle an Event.

Figure 2–3 Login Form with Button



Add Code to Handle an Event

Finally, make the button display the text message when the user presses it. Add the code in Example 2–6 before the code for the scene.

Example 2–6 Button Event

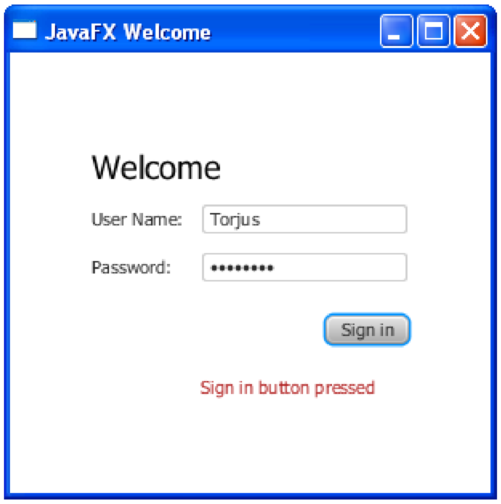
```
btn.setOnAction(new EventHandler<ActionEvent>() {

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

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.

Run the Application

Right-click the **Login** project node in the Projects window, choose **Run**, and then click the Sign in button. Figure 2–4 shows the results. If you run into problems, then take a look at the code in the `Login.java` file.

Figure 2–4 Final Login Form



Where to Go from Here

This concludes the basic form tutorial, but you can continue reading the following tutorials on developing JavaFX applications.

- Fancy Forms with JavaFX CSS provides tips on how to add a background image and radically change the style of the text, label, and button in the login form.
- Using FXML to Create a User Interface shows an alternate method for creating the login user interface. FXML is an XML-based language that provides the structure for building a user interface separate from the application logic of your code.
- Working With Layouts in JavaFX explains the built-in JavaFX layout panes, and tips and tricks for using them.
- Deploying Your First JavaFX Application provides information on how to run your application outside NetBeans IDE.

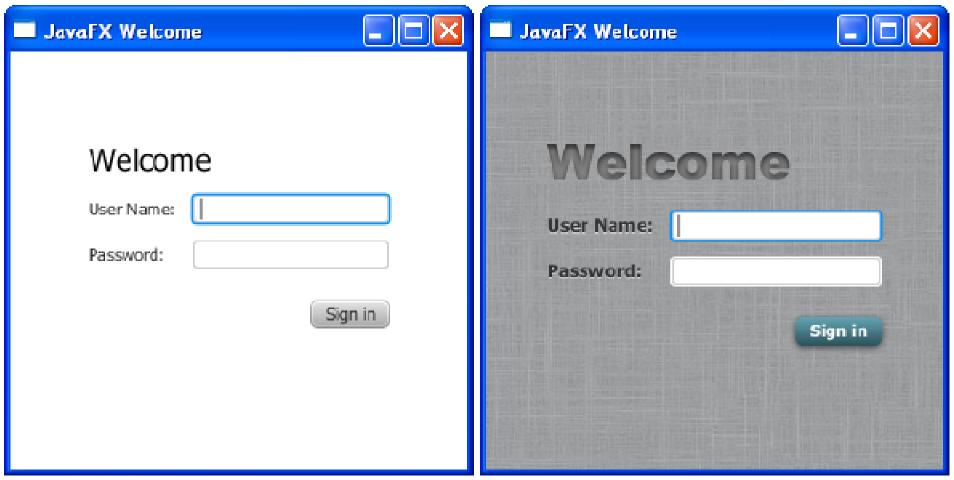
Also try out the JavaFX samples, which you can download from the JDK Demos and Samples section of the Java SE Downloads page at `http://www.oracle.com/technetwork/java/javase/downloads/`. The Ensemble sample contains examples of layouts and their source code.

3 Fancy Forms with JavaFX CSS

This tutorial is about making your JavaFX application look attractive by adding a Cascading Style Sheet (CSS). You develop a design, create a `.css` file, and apply the new styles.

In this tutorial, you will take a Login form that uses default styles for labels, buttons, and background color, and, with a few simple CSS modifications, turn it into a stylized application, as shown in Figure 3–1.

Figure 3–1 Login Form With and Without CSS



The tool used in this Getting Started tutorial is NetBeans IDE. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 2. See the System Requirements for details.

Create the Project

If you followed the Getting Started guide from the start, then you already created the Login project required for this tutorial. If not, download the Login project by right-clicking **Login.zip** and saving it to your file system.   Extract the files from the zip file, and then open the project in NetBeans IDE.

Create the CSS File

Your first task is to create a new CSS file and save it in the same directory as the main class of your application. After that, you must make the JavaFX application aware of the newly added Cascading Style Sheet.

1. In the NetBeans IDE Projects window, expand the **Login** project node and then the **Source Packages** directory node.
2. Right-click the **login** folder under the Source Packages directory and choose **New**, then **Other**.
3. In the New File dialog box, choose **Other**, then **Cascading Style Sheet**, and click **Next**.
4. Enter **Login** for the File Name text field and ensure the Folder text field value is `src\login`.
5. Click **Finish**.
6. In the `Login.java` file, initialize the `style sheets` variable of the `Scene` class to point to the Cascading Style Sheet by including the line of code shown in bold below so that it appears as shown in Example 3–1.

Example 3–1 Initialize the stylesheets Variable

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

This code looks for the style sheet in the `src\login` directory in the NetBeans project.

Add a Background Image

A background image helps make your form more attractive. For this tutorial, you add a gray background with a linen-like texture.

First, download the background image by right-clicking **background.jpg** and saving it to your file system. Then, copy the file into the `src\login` folder in the Login NetBeans project.

Now, add the code for the `background-image` property to the CSS file. Remember that the path is relative to the style sheet. So, in the code in Example 3–2, the `background.jpg` image is in the same directory as the `Login.css` file.

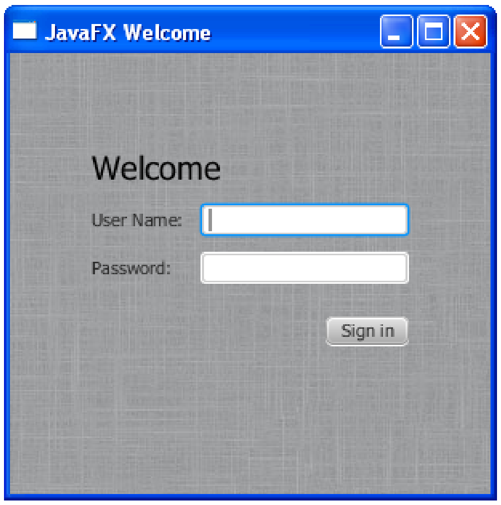
Example 3–2 Background Image

```
.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")`).

Figure 3–2 shows the login form with the new gray background.

Figure 3–2 Gray Linen Background



Style the Labels

The next controls to enhance are the labels. You will use the `.label` style class, which means the styles will affect all labels in the form. The code is in Example 3–3.

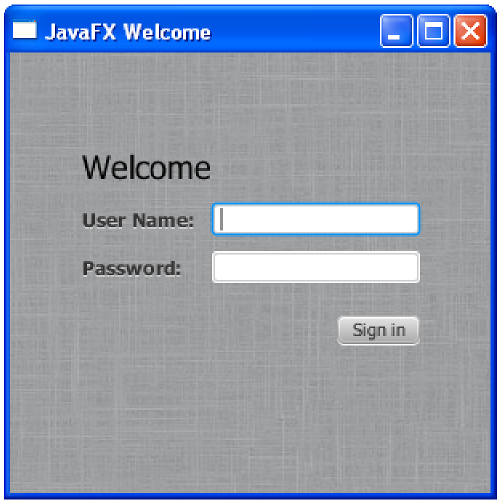
Example 3–3 Font Size, Fill, Weight, and Effect on Labels

```
.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 increases the font size and weight and applies a drop shadow of a gray color (#333333). The purpose of the drop shadow is to add contrast between the dark gray text and the light gray background. See the section on effects in the JavaFX CSS Reference Guide for details on the parameters of the drop shadow property.

The enhanced User Name and Password labels are shown in Figure 3–3.

Figure 3–3 Bigger, Bolder Labels with Drop Shadow



Style Text

Now, create some special effects on the two `Text` objects in the form: `scenetitle`, which includes the text `Welcome`, and `actiontarget`, which is the text that is returned when the user presses the Sign in button. You can apply different styles to `Text` objects used in such diverse ways.

1. In the `Login.java` file, 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);
   ```

   By switching to CSS over inline styles, you separate the design from the content. This approach makes it easier for a designer to have control over the style without having to modify content.

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 in Example 3–4.

Example 3–4 Text Effect

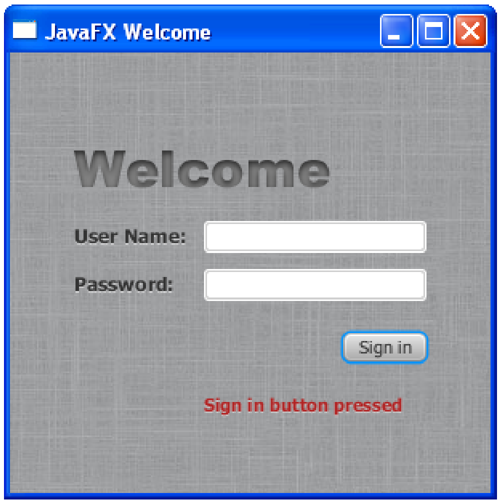
```
#welcome-text {
   -fx-font-size: 32px;
   -fx-font-family: "Arial Black";
   -fx-fill: #818181;
   -fx-effect: innershadow( three-pass-box , rgba(0,0,0,0.7) , 6, 0.0 , 0 , 2 );
}
#actiontarget {
  -fx-fill: FIREBRICK;
  -fx-font-weight: bold;
  -fx-effect: dropshadow( gaussian , rgba(255,255,255,0.5) , 0,0,0,1 );
}
```

The size of the Welcome text is increased to 32 points and the font is changed to Arial Black. The text fill color is set to a dark gray color (#818181) and an inner shadow effect is applied, creating an embossing effect. You can apply an inner shadow to any color by changing the text fill color to be a darker version of the background. See the section on effects in the JavaFX CSS Reference Guide for details about the parameters of inner shadow property.

The style definition for `actiontarget` is similar to what you have seen before.

Figure 3–4 shows the font changes and shadow effects on the two `Text` objects.

Figure 3–4 Text with Shadow Effects



Style the Button

The next step is to style the button, making it change style when the user hovers the mouse over it. This change will give users an indication that the button is interactive, a standard design practice.

First, create the style for the initial state of the button by adding the code in Example 3–5. 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.

Example 3–5 Drop Shadow for Button

```
.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 );
}
```

Now, 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 in Example 3–6.

Example 3–6 Button Hover Style

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

Figure 3–5 shows the initial and hover states of the button with its new blue-gray background and white bold text.

Figure 3–5 Initial and Hover Button States

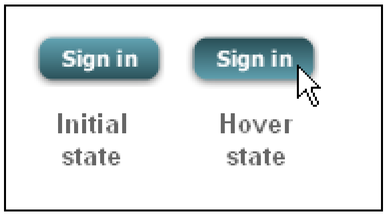
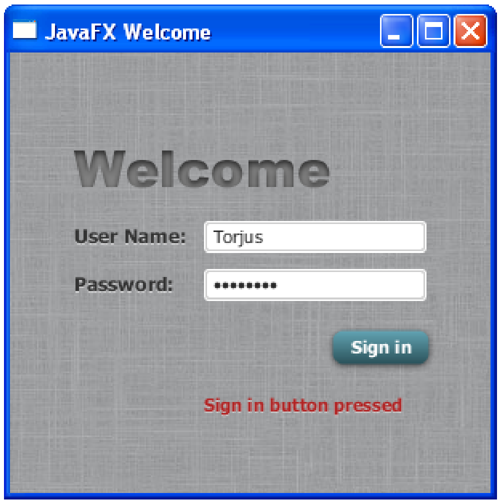


Figure 3–6 shows the final application.

Figure 3–6 Final Stylized Application



Where to Go from Here

Here are some things for you to try next:

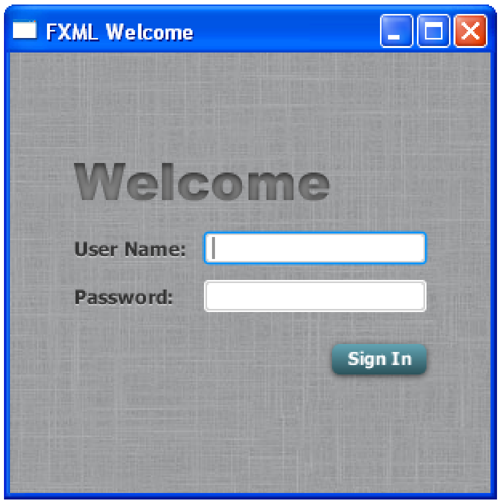
- See what you can create using CSS. Some documents that can help you are Skinning JavaFX Applications with CSS, Styling Charts with CSS, and the JavaFX CSS Reference Guide. Skinning with CSS and CSS Analyzer also provides information on how you can use the JavaFX Scene Builder tool to skin your JavaFX FXML layout.
- See Styling FX Buttons with CSS for examples of how to create common button styles using CSS.
- Try deploying your application outside NetBeans IDE. See Deploying Your First JavaFX Application.

4 Using FXML to Create a User Interface

This tutorial shows the benefits of using JavaFX FXML, which is an XML-based language that provides the structure for building a user interface separate from the application logic of your code.

If you started this document from the beginning, then you have seen how to create a login application using just JavaFX. Here, you use FXML to create the same login user interface, separating the application design from the application logic, thereby making the code easier to maintain. The login user interface you build in this tutorial is shown in Figure 4–1.

Figure 4–1 Login User Interface



This tutorial uses NetBeans IDE. Ensure that the version of NetBeans IDE that you are using supports JavaFX 2.2. See the System Requirements for details.

Set Up the Project

Your first task is to set up a JavaFX FXML project in NetBeans IDE:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX FXML Application**. Click **Next**.
3. Name the project **FXMLExample** and click **Finish**.

NetBeans IDE opens an FXML project that includes the code for a basic Hello World application. The application includes three files:

- **FXMLExample.java.** This file takes care of the standard Java code required for an FXML application.
- **Sample.fxml.** This is the FXML source file in which you define the user interface.
- **SampleController.java.** This is the controller file for handling the mouse and keyboard input.

4. Rename SampleController.java to FXMLExampleController.java so that the name is more meaningful for this application.
   a. In the Projects window, right-click **SampleController.java** and choose **Refactor** then **Rename**.
   b. Enter **FXMLExampleController**, and click **Refactor**.
5. Rename Sample.fxml to fxml_example.fxml.
   a. Right-click **Sample.fxml** and choose **Rename**.
   b. Enter **fxml_example** and click **OK**.

Load the FXML Source File

The first file you edit is the `FXMLExample.java` file. This file includes the code for setting up the application main class and for defining the stage and scene. More specific to FXML, the file uses the `FXMLLoader` class, which is responsible for loading the FXML source file and returning the resulting object graph.

Make the changes shown in bold in Example 4–1.

Example 4–1 FXMLExample.java

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

    Scene scene = new Scene(root, 300, 275);

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

A good practice is to set the height and width of the scene when you create it, in this case 300 by 275; otherwise the scene defaults to the minimum size needed to display its contents.

Modify the Import Statements

Next, edit the `fxml_example.fxml` file. This file specifies the user interface that is displayed when the application starts. The first task is to modify the import statements so your code looks like Example 4–2.

Example 4–2 XML Declaration and Import Statements

```
<?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.*?>
```

As in Java, class names can be fully qualified (including the package name), or they can be imported using the import statement, as shown in Example 4–2. If you prefer, you can use specific import statements that refer to classes.

Create a GridPane Layout

The Hello World application generated by NetBeans uses an `AnchorPane` layout. For the login form, you will use a `GridPane` layout because it enables you to create a flexible grid of rows and columns in which to lay out controls.

Remove the `AnchorPane` layout and its children and replace it with the `GridPane` layout in Example 4–3.

Example 4–3   GridPane Layout

```
<GridPane fx:controller="fxmlexample.FXMLExampleController"
    xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
<padding><Insets top="25" right="25" bottom="10" left="25"/></padding>

</GridPane>
```

In this application, the `GridPane` layout is the root element of the FXML document and as such has two attributes. The `fx:controller` attribute is required when you specify controller-based event handlers in your markup. The `xmlns:fx` attribute is always required and specifies the `fx` namespace.

The remainder of the code controls the alignment and spacing of the grid pane. The alignment property changes the default position of the grid from the top left of the scene to the center. The `gap` properties manage the spacing between the rows and columns, while the `padding` property manages the space around the edges of the grid pane.

As the window is resized, the nodes within the grid pane are resized according to their layout constraints. In this example, the grid remains in the center when you grow or shrink the window. The padding properties ensure there is a padding around the grid when you make the window smaller.

Add Text and Password Fields

Looking back at Figure 4–1, you can see that the login form requires the title “Welcome” and text and password fields for gathering information from the user. The code in Example 4–4 is part of the `GridPane` layout and must be placed above the `</GridPane>` statement.

Example 4–4   Text, Label, TextField, and Password Field Controls

```
    <Text text="Welcome"
        GridPane.columnIndex="0" GridPane.rowIndex="0"
        GridPane.columnSpan="2"/>

    <Label text="User Name:"
```

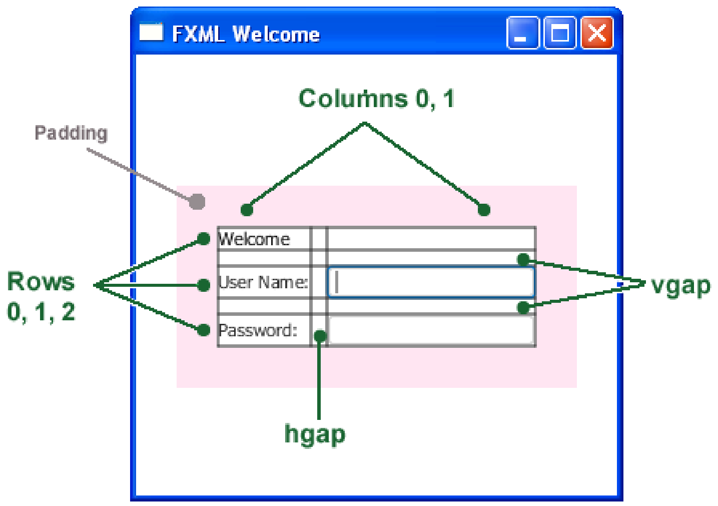
```
        GridPane.columnIndex="0" GridPane.rowIndex="1"/>

    <TextField
        GridPane.columnIndex="1" GridPane.rowIndex="1"/>

    <Label text="Password:"
        GridPane.columnIndex="0" GridPane.rowIndex="2"/>

    <PasswordField fx:id="passwordField"
        GridPane.columnIndex="1" GridPane.rowIndex="2"/>
```

The first line creates a `Text` object and sets its text value to `Welcome`. The `GridPane.columnIndex` and `GridPane.rowIndex` attributes correspond to the placement of the `Text` control in the grid. The numbering for rows and columns in the grid starts at zero, and the location of the `Text` control is set to (0,0), meaning it is in the first column of the first row. The `GridPane.columnSpan` attribute is set to 2, making the Welcome title span two columns in the grid. You will need this extra width later in the tutorial when you add a style sheet to increase the font size of the text to 32 points.

The next lines create a `Label` object with text `User Name` at column 0, row 1 and a `TextField` object to the right of it at column 1, row 1. Another `Label` and `PasswordField` object are created and added to the grid in a similar fashion.

When working with a grid layout, you can display the grid lines, which is useful for debugging purposes. In this case, set the `gridLinesVisible` property to `true` by adding the statement `<gridLinesVisible>true</gridLinesVisible>` right after the `<padding></padding>` statement. Then, when you run the application, you see the lines for the grid columns and rows as well as the gap properties, as shown in Figure 4–2.

Figure 4–2 Login Form with Grid Lines



Add a Button and Text

The final two controls required for the application are a `Button` control for submitting the data and a `Text` control for displaying a message when the user presses the button. The code is in Example 4–5. Add this code before `</GridPane>`.

Example 4–5 HBox, Button, and Text

```
<HBox spacing="10" alignment="bottom_right"
        GridPane.columnIndex="1" GridPane.rowIndex="4">
        <Button text="Sign In"
        onAction="#handleSubmitButtonAction"/>
</HBox>

<Text fx:id="actiontarget"
       GridPane.columnIndex="1" GridPane.rowIndex="6"/>
```

An `HBox` pane is needed to set an alignment for the button that is different from the default alignment applied to the other controls in the `GridPane` layout. The `alignment` property is set to `bottom_right`, which positions a node at the bottom of the space vertically and at the right edge of the space horizontally. The `HBox` pane is added to the grid in column 1, row 4.

The `HBox` pane has one child, a `Button` with `text` property set to `Sign in` and an `onAction` property set to `handleSubmitButtonAction()`. While FXML is a convenient way to define the structure of an application's user interface, it does not provide a way to implement an application's behavior. You implement the behavior for the `handleSubmitButtonAction()` method in Java code in the next section of this tutorial, Add Code to Handle an Event.

Assigning an `fx:id` value to an element, as shown in the code for the `Text` control, creates a variable in the document's namespace, which you can refer to from elsewhere in the code. While not required, defining a controller field helps clarify how the controller and markup are associated.

Add Code to Handle an Event

Now make the `Text` control display a message when the user presses the button. You do this in the `FXMLExampleController.java` file. Delete the code that NetBeans IDE generated and replace it with the code in Example 4–6.

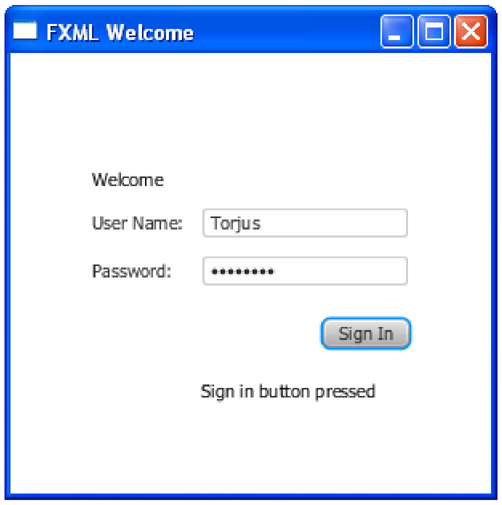
Example 4–6 FXMLExampleController.java

```
package fxmlexample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class FXMLExampleController {
    @FXML private Text actiontarget;

    @FXML protected void handleSubmitButtonAction(ActionEvent event) {
        actiontarget.setText("Sign in button pressed");
    }

}
```

The `@FXML` annotation is used to tag nonpublic controller member fields and handler methods for use by FXML markup. The `handleSubmtButtonAction` method sets the `actiontarget` variable to `Sign in button pressed` when the user presses the button.

You can run the application now to see the complete user interface. Figure 4–3 shows the results when you type text in the two fields and click the Sign in button. If you have any problems, then you can compare your code against the FXMLLogin example.

Figure 4–3 FXML Login Window



Use a Scripting Language to Handle Events

As an alternative to using Java code to create an event handler, you can create the handler with any language that provides a JSR 223-compatible scripting engine. Such languages include JavaScript, Groovy, Jython, and Clojure.

Optionally, you can try using JavaScript now.

1. In the file `fxml_example.fxml`, add the JavaScript declaration after the XML doctype declaration.

```
<?language javascript?>
```

2. In the `Button` markup, change the name of the function so the call looks as follows:

```
onAction="handleSubmitButtonAction(event);"
```

3. Remove the `fx:controller` attribute from the `GridPane` markup and add the JavaScript function in a `<script>` tag directly under it, as shown in Example 4–7.

Example 4–7 JavaScript in FXML

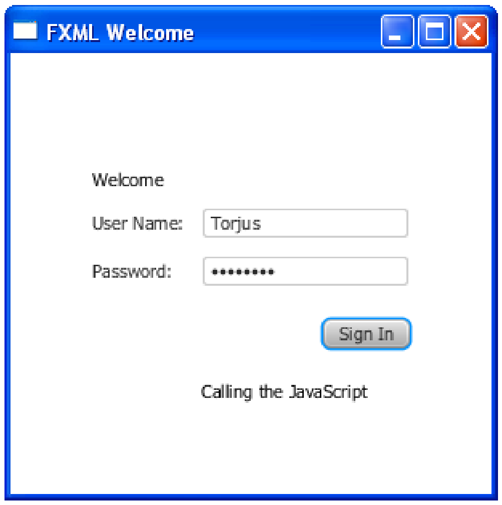
```
<GridPane xmlns:fx="http://javafx.com/fxml"
          alignment="center" hgap="10" vgap="10">
     <fx:script>
         function handleSubmitButtonAction() {
             actiontarget.setText("Calling the JavaScript");
         }
     </fx:script>
```

Alternatively, you can put the JavaScript functions in an external file (such as `fxml_example.js`) and include the script like this:

```
<fx:script source="fxml_example.js"/>
```

The result is in Figure 4–4.

Figure 4–4 Login Application Using JavaScript



If you are considering using a scripting language with FXML, then note that an IDE might not support stepping through script code during debugging.

Style the Application with CSS

The final task is to make the login application look attractive by adding a Cascading Style Sheet (CSS).

1. Create a style sheet.
   a. In the Project window, right-click the login folder under Source Packages and choose **New**, then **Other**.
   b. In the New File dialog box, choose **Other**, then **Cascading Style Sheet** and click **Next**.
   c. Enter **Login** and click **Finish**.
   d. Copy the contents of the `Login.css` file attached to this document into your CSS file. For a description of the classes in the CSS file, see Fancy Forms with JavaFX CSS.
2. Download the gray, linen-like image for the background in the `background.jpg` file and add it to the fxmlexample folder.
3. Open the `fxml_example.fxml` file and add a stylesheets element before the end of the markup for the `GridPane` layout as shown in Example 4–8.

Example 4–8 Style Sheet

```
  <stylesheets>
    <URL value="@Login.css" />
  </stylesheets>

</GridPane>
```

The @ symbol before the name of the style sheet in the URL indicates that the style sheet is in the same directory as the FXML file.

4. To use the root style for the grid pane, add a style class to the markup for the `GridPane` layout as shown in Example 4–9.

Example 4–9 Style the GridPane

```
<GridPane fx:controller="fxmlexample.FXMLExampleController"
    xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10"
    styleClass="root">
```

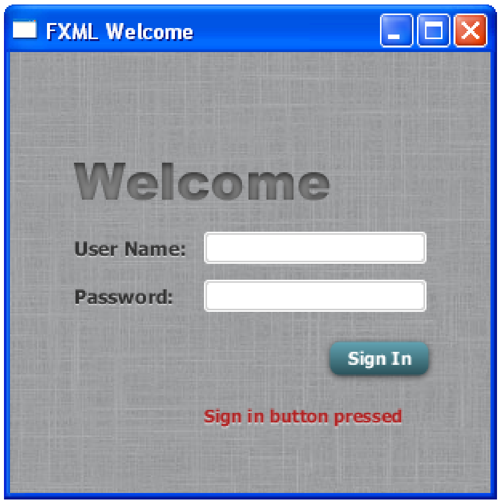
5. Create a `welcome-text` ID for the Welcome `Text` object so it uses the style `#welcome-text` defined in the CSS file, as shown in Example 4–10.

Example 4–10 Text ID

```
<Text id="welcome-text" text="Welcome"
        GridPane.columnIndex="0" GridPane.rowIndex="0"
        GridPane.columnSpan="2"/>
```

6. Run the application. Figure 4–5 shows the stylized application.

Figure 4–5 Stylized Login Application



For information about how to run your application outside NetBeans IDE, see Deploying Your First JavaFX Application.

Where to Go from Here

Now that you are familiar with FXML, look at Introduction to FXML, which provides more information on the elements that make up the FXML language. The document is included in the javafx.fxml package in the API documentation at `http://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html`.

You can also try out the JavaFX Scene Builder tool by opening the `fxml_example.fxml` file in Scene Builder and making modifications. This tool provides a visual layout environment for designing the UI for JavaFX applications and automatically generates the FXML code for the layout. Note that the FXML file might be reformatted when saved. See Getting Started with JavaFX Scene Builder for more information on this tool. The Skinning with CSS and CSS Analyzer section of the JavaFX Scene Builder User Guide also give you information on how you can skin your FXML layout.

5

Animation and Visual Effects in JavaFX

You can use JavaFX to quickly develop applications with rich user experiences. In this Getting Started tutorial, you will learn to create animated objects and attain complex effects with very little coding.

Figure 5–1 shows the application to be created.

Figure 5–1 Colorful Circles Application

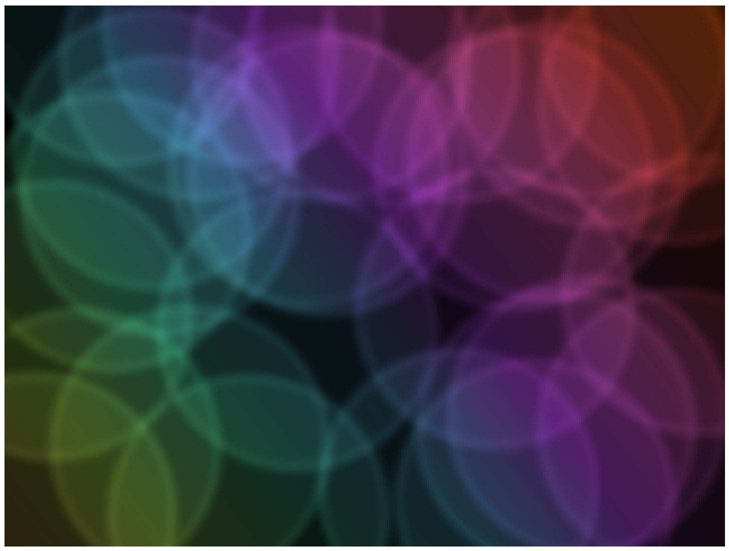
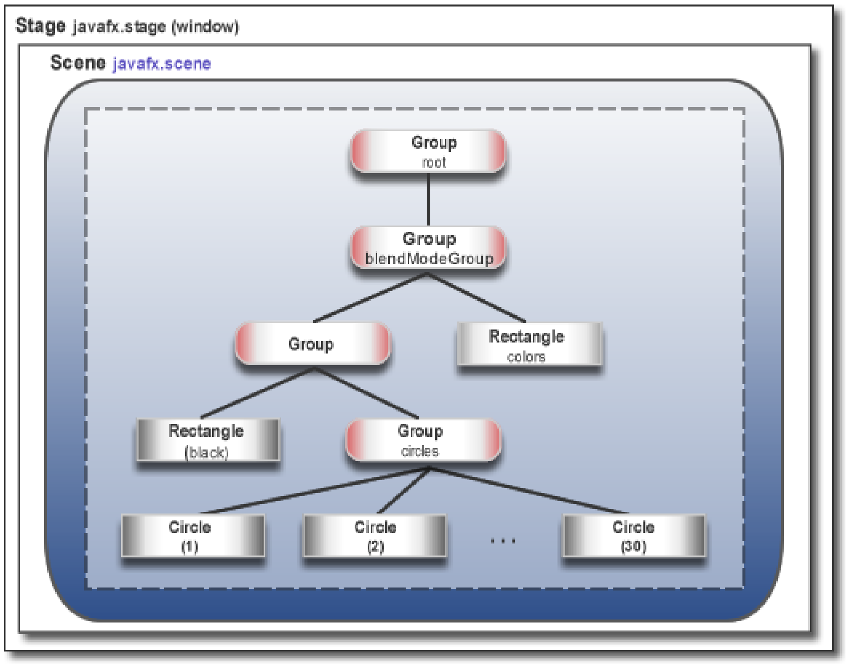


Figure 5–2 shows the scene graph for the `ColorfulCircles` application. Nodes that branch are instantiations of the `Group` class, and the nonbranching nodes, also known as leaf nodes, are instantiations of the `Rectangle` and `Circle` classes.

Figure 5–2 Colorful Circles Scene Graph



The tool used in this Getting Started tutorial is NetBeans IDE. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 2. See the System Requirements for details.

Set Up the Application

Set up your JavaFX project in NetBeans IDE as follows:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX Application**. Click **Next**.
3. Name the project **ColorfulCircles** and click **Finish**.
4. Open the ColorfulCircles.java file, copy the import statements, and paste them into the source file for your project, overwriting the import statements that NetBeans IDE generated.

   Or, you can generate the import statements as you work your way through the tutorial by using either the code completion feature or the Fix Imports command from the Source menu in NetBeans IDE. When there is a choice of import statements, choose the one that starts with `javafx`.

Set Up the Project

Delete the `ColorfulCircles` class from the source file that NetBeans IDE generated and replace it with the code in Example 5–1.

Example 5–1 Basic Application

```
public class ColorfulCircles extends Application {

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

    @Override
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 800, 600, Color.BLACK);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}
```

For the ColorfulCircles application, it is appropriate to use a group node as the root node for the scene. The size of the group is dictated by the size of the nodes within it. For most applications, however, you want the nodes to track the size of the scene and change when the stage is resized. In that case, you use a resizable layout node as the root, as described in Creating a Form in JavaFX.

You can compile and run the ColorfulCircles application now, and at each step of the tutorial, to see the intermediate results. If you run into problems, then take a look at the code in the ColorfulCircles.java file. At this point, the application is a simple black window.

Add Graphics

Next, create 30 circles by adding the code in Example 5–2 before the `primaryStage.show()` line.

Example 5–2 30 Circles

```
Group circles = new Group();
for (int i = 0; i < 30; i++) {
   Circle circle = new Circle(150, Color.web("white", 0.05));
   circle.setStrokeType(StrokeType.OUTSIDE);
   circle.setStroke(Color.web("white", 0.16));
   circle.setStrokeWidth(4);
   circles.getChildren().add(circle);
}
root.getChildren().add(circles);
```

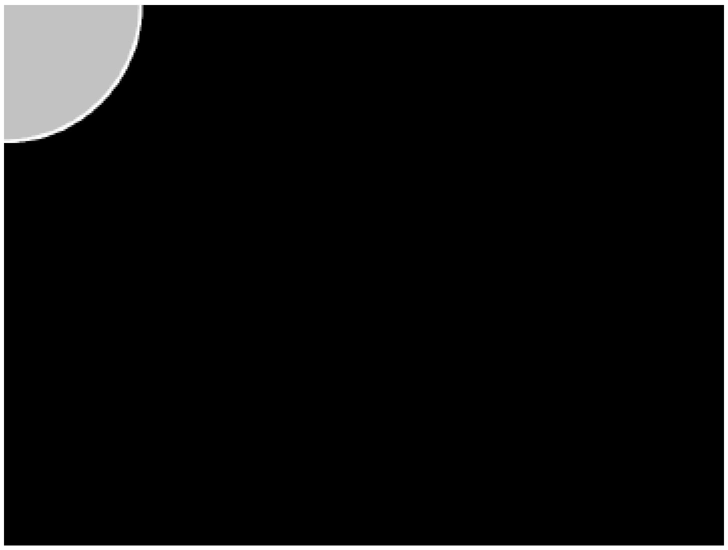
This code creates a group named `circles`, then uses a `for` loop to add 30 circles to the group. Each circle has a radius of `150`, fill color of `white`, and opacity level of `5` percent, meaning it is mostly transparent.

To create a border around the circles, the code includes the `StrokeType` class. A stroke type of `OUTSIDE` means the boundary of the circle is extended outside the interior by the `StrokeWidth` value, which is `4`. The color of the stroke is `white`, and the opacity level is `16` percent, making it brighter than the color of the circles.

The final line adds the `circles` group to the root node. This is a temporary structure. Later, you will modify this scene graph to match the one shown in Figure 5–2.

Figure 5–3 shows the application. Because the code does not yet specify a unique location for each circle, the circles are drawn on top of one another, with the upper left-hand corner of the window as the center point for the circles. The opacity of the overlaid circles interacts with the black background, producing the gray color of the circles.

Figure 5–3 Circles



Add a Visual Effect

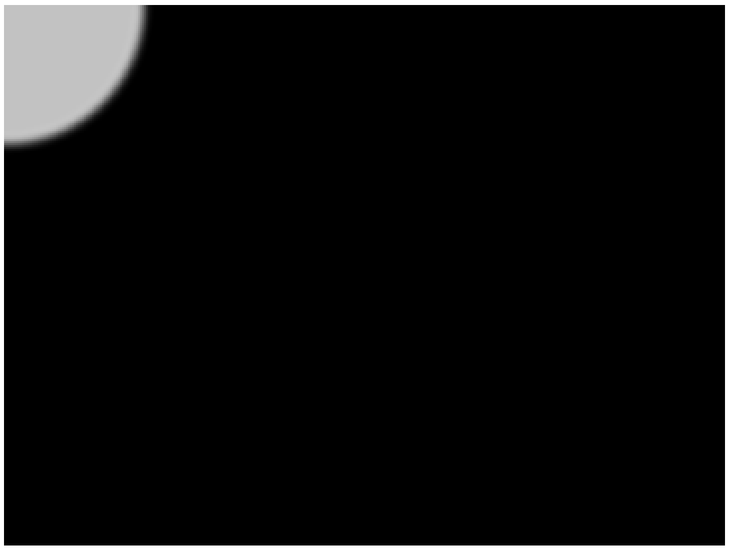
Continue by applying a box blur effect to the circles so that they appear slightly out of focus. The code is in Example 5–3. Add this code before the `primaryStage.show()` line.

Example 5–3 Box Blur Effect

```
circles.setEffect(new BoxBlur(10, 10, 3));
```

This code sets the blur radius to `10` pixels wide by `10` pixels high, and the blur iteration to `3,` making it approximate a Gaussian blur. This blurring technique produces a smoothing effect on the edge of the circles, as shown in Figure 5–4.

Figure 5–4 Box Blur on Circles



Create a Background Gradient

Now, create a rectangle and fill it with a linear gradient, as shown in Example 5–4.

Add the code before the `root.getChildren().add(circles)` line so that the gradient rectangle appears behind the circles.

Example 5–4 Linear Gradient

```
Rectangle colors = new Rectangle(scene.getWidth(), scene.getHeight(),
     new LinearGradient(0f, 1f, 1f, 0f, true, CycleMethod.NO_CYCLE, new
         Stop[]{
            new Stop(0, Color.web("#f8bd55")),
            new Stop(0.14, Color.web("#c0fe56")),
            new Stop(0.28, Color.web("#5dfbc1")),
            new Stop(0.43, Color.web("#64c2f8")),
            new Stop(0.57, Color.web("#be4af7")),
            new Stop(0.71, Color.web("#ed5fc2")),
            new Stop(0.85, Color.web("#ef504c")),
            new Stop(1, Color.web("#f2660f")),}));
colors.widthProperty().bind(scene.widthProperty());
colors.heightProperty().bind(scene.heightProperty());
root.getChildren().add(colors);
```

This code creates a rectangle named `colors`. The rectangle is the same width and height as the scene and is filled with a linear gradient that starts in the lower left-hand corner (0, 1) and ends in the upper right-hand corner (1, 0). The value of `true` means the gradient is proportional to the rectangle, and `NO_CYCLE` indicates that the color cycle will not repeat. The `Stop[]` sequence indicates what the gradient color should be at a particular spot.

The next two lines of code make the linear gradient adjust as the size of the scene changes by binding the width and height of the rectangle to the width and height of the scene. See Using JavaFX Properties and Bindings for more information on binding.

The final line of code adds the `colors` rectangle to the root node.

The gray circles with the blurry edges now appear on top of a rainbow of colors, as shown in Figure 5–5.

Figure 5–5 Linear Gradient

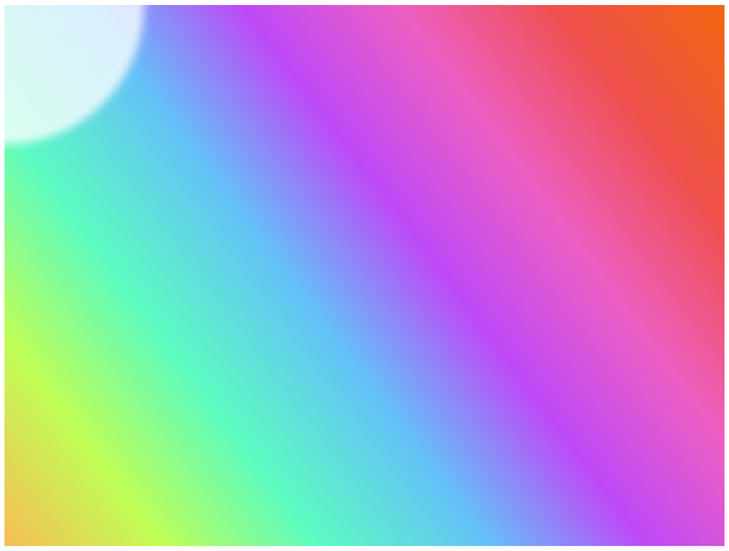
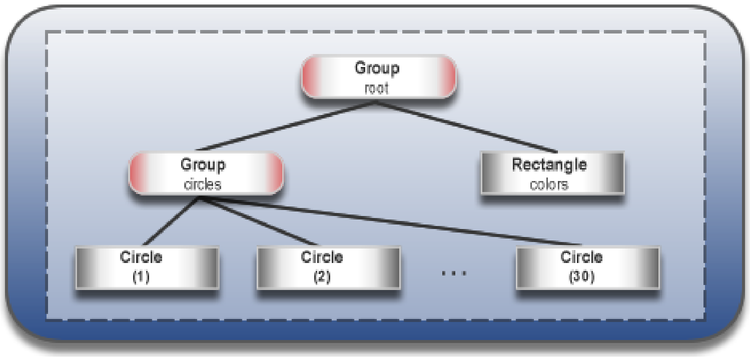


Figure 5–6 shows the intermediate scene graph. At this point, the `circles` group and `colors` rectangle are children of the root node.

Figure 5–6 Intermediate Scene Graph



Apply a Blend Mode

Next, add color to the circles and darken the scene by adding an overlay blend effect. You will remove the `circles` group and the linear gradient rectangle from the scene graph and add them to the new overlay blend group.

1. Locate the following two lines of code:

```
root.getChildren().add(colors);
root.getChildren().add(circles);
```

2. Replace the two lines of code from Step 1 with the code in Example 5–5.

Example 5–5 Blend Mode

```
Group blendModeGroup =
    new Group(new Group(new Rectangle(scene.getWidth(), scene.getHeight(),
        Color.BLACK), circles), colors);
```

```
colors.setBlendMode(BlendMode.OVERLAY);
root.getChildren().add(blendModeGroup);
```

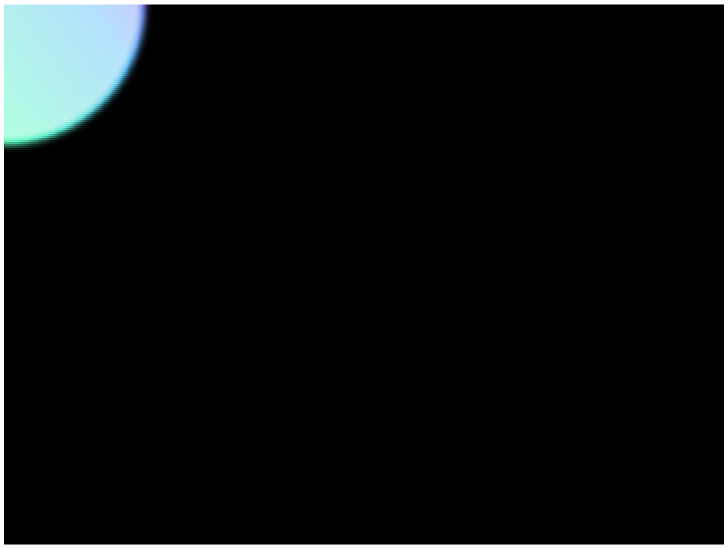
The group `blendModeGroup` sets up the structure for the overlay blend. The group contains two children. The first child is a new (unnamed) `Group` containing a new (unnamed) black rectangle and the previously created `circles` group. The second child is the previously created `colors` rectangle.

The `setBlendMode()` method applies the overlay blend to the `colors` rectangle. The final line of code adds the `blendModeGroup` to the scene graph as a child of the root node, as depicted in Figure 5–2.

An overlay blend is a common effect in graphic design applications. Such a blend can darken an image or add highlights or both, depending on the colors in the blend. In this case, the linear gradient rectangle is used as the overlay. The black rectangle serves to keep the background dark, while the nearly transparent circles pick up colors from the gradient, but are also darkened.

Figure 5–7 shows the results. You will see the full effect of the overlay blend when you animate the circles in the next step.

Figure 5–7 Overlay Blend



Add Animation

The final step is to use JavaFX animations to move the circles:

1. If you have not done so already, add `import static java.lang.Math.random;` to the list of import statements.
2. Add the animation code in Example 5–6 before the `primaryStage.show()` line.

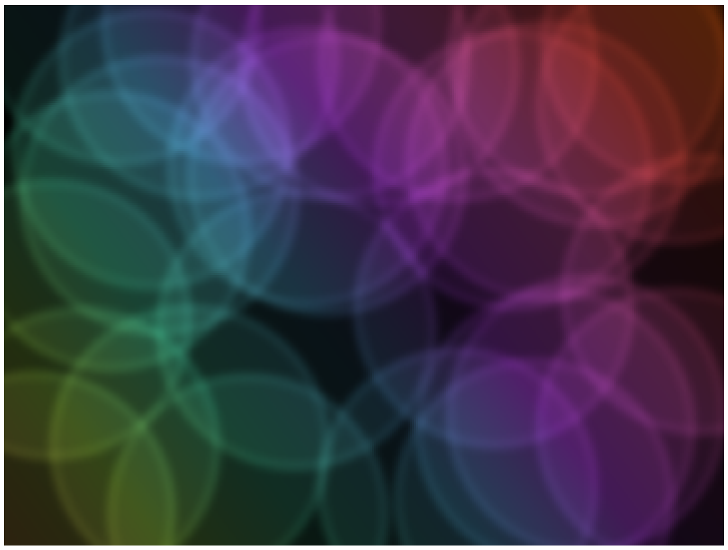
Example 5–6 Animation

```
Timeline timeline = new Timeline();
for (Node circle: circles.getChildren()) {
    timeline.getKeyFrames().addAll(
        new KeyFrame(Duration.ZERO, // set start position at 0
            new KeyValue(circle.translateXProperty(), random() * 800),
            new KeyValue(circle.translateYProperty(), random() * 600)
            ),
            new KeyFrame(new Duration(40000), // set end position at 40s
                new KeyValue(circle.translateXProperty(), random() * 800),
                new KeyValue(circle.translateYProperty(), random() * 600)
            )
        );
}
// play 40s of animation
timeline.play();
```

Animation is driven by a timeline, so this code creates a timeline, then uses a `for` loop to add two key frames to each of the 30 circles. The first key frame at 0 seconds uses the properties `translateXProperty` and `translateYProperty` to set a random position of the circles within the window. The second key frame at 40 seconds does the same. Thus, when the timeline is played, it animates all circles from one random position to another over a period of 40 seconds.

Figure 5–8 shows the 30 colorful circles in motion, which completes the application. For the complete source code, see the ColorfulCircles.java file.

Figure 5–8 Animated Circles



Where to Go from Here

Here are several suggestions about what to do next:

- Try deploying your application outside NetBeans IDE. See Deploying Your First JavaFX Application.
- Try the JavaFX samples, which you can download from the JDK Demos and Samples section of the Java SE Downloads page at `http://www.oracle.com/technetwork/java/javase/downloads/`. Especially take a look at the Ensemble application, which provides sample code for animations and effects.
- Read `Creating Transitions and Timeline Animation in JavaFX`. You will find more details on timeline animation as well as information on fade, path, parallel, and sequential transitions.

- See `Creating Visual Effects in JavaFX` for additional ways to enhance the look and design of your application, including reflection, lighting, and shadow effects.
- Try the JavaFX Scene Builder tool to create visually interesting applications. This tool provides a visual layout environment for designing the UI for JavaFX applications and generates FXML code. You can use the Properties panel or the Modify option of the menu bar to add effects to the UI elements. See the `Properties Panel` section or `Table 3-3` in the `Menu Bar` section of the `JavaFX Scene Builder User Guide` for information.

6

Deploying Your First JavaFX Application

This topic shows how to deploy the samples from any of the Getting Started with JavaFX tutorials.

If you develop your JavaFX application in NetBeans IDE, it is packaged automatically and is easy to deploy.

This page contains the following sections.

- Deployment Modes
- Packaging the Application in NetBeans IDE
- Running the Application Outside NetBeans IDE
- Deploying the Packaged Files
- Other Ways to Package JavaFX Applications

Deployment Modes

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

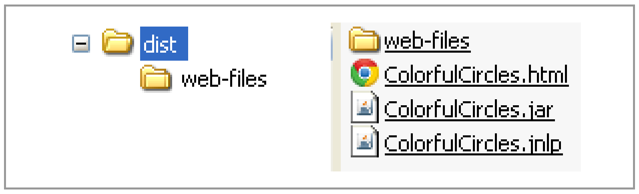
Packaging the Application in NetBeans IDE

When you run your application in NetBeans IDE or use the Clean and Build command, your application is packaged for all modes of JavaFX deployment, using options that are set as project properties. By default, applications are packaged into the following files as shown in Figure 6–1:

- A JAR file, which contains the compiled class files and images.
- A JNLP file, which contains a deployment descriptor for the two web modes (Web Start and embedded in browser).
- An HTML file, which contains basic code for running both the Web Start application and the embedded application using the Deployment Toolkit.
- A web-files folder, which contains an offline set of files from the Java Deployment Toolkit to assist with starting up your application. For more information about

using the Java Deployment Toolkit in JavaFX applications, see "Deployment in the Browser" at
`http://docs.oracle.com/javafx/2/deployment/deployment_toolkit.htm`

Figure 6–1 Example of Default Application Packaging



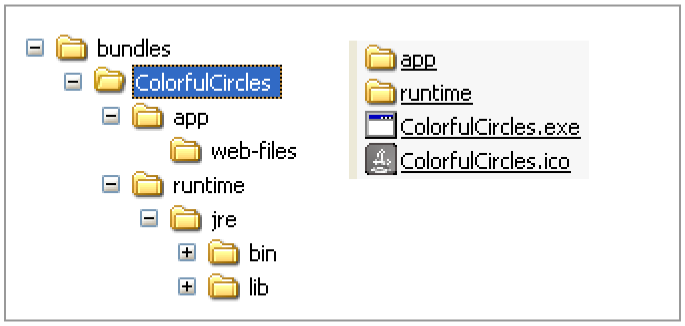
Optionally, you can package your application as a self-contained application in NetBeans IDE by adding the task shown in Example 6–1 to the build.xml file.

Example 6–1 Changes to the build.xml File for Packaging Self-Contained Applications

```
<project name="ColorfulCircles" default="default" basedir="."
        xmlns:fx="javafx:com.sun.javafx.tools.ant">
    target name="-post-jfx-deploy">
        <fx:deploy width="${javafx.run.width}" height="${javafx.run.height}"
                   nativeBundles="all"
                   outdir="${basedir}/${dist.dir}" outfile="${application.title}">
            <fx:application name="${application.title}"
                            mainClass="${javafx.main.class}"/>
            <fx:resources>
                <fx:fileset dir="${basedir}/${dist.dir}" includes="*.jar"/>
            </fx:resources>
            <fx:info title="${application.title}" vendor="${application.vendor}"/>
        </fx:deploy>
    </target>
</project>
```

A self-contained application runs similar to the way in which a native application runs. The self-contained application package contains your application, the JRE, the JavaFX runtime, and a platform-specific application launcher. An example of a package is shown in Figure 6–2. For additional information, see `Self-Contained Application Packaging`.

Figure 6–2 Example of a Self-Contained Application Package



Sizing the Application Window

With standalone and Web Start mode, the window is sized around the width and height values you specify for the scene in the code. If you do not specify a width and height in the code, the window automatically sizes itself to fit the application.

Applications that are embedded in a web page require that you set width and height values in the NetBeans project's properties, even if you specify width and height values in the code. You can even specify different width and height values in the project properties from what is specified in the code, so you can experiment to see what looks best in the browser.

To specify application width and height in NetBeans project properties:

1. In the Projects pane, right-click your NetBeans project and choose **Properties**.
2. Select the **Run** category.
3. In the section entitled Web Start and Browser Application Properties, specify width and height values.
4. If you want to test the application in a browser when you run it inside NetBeans IDE, choose **in Browser** in the Run field.
5. Click **OK** to close the Properties dialog box.
6. Right-click the project and choose **Clean and Build**.
7. If you chose to run the application in the browser in the project properties, you can right-click the project and choose **Run**. Otherwise, test the HTML file outside the browser by following the instructions in Running the Application Outside NetBeans IDE.

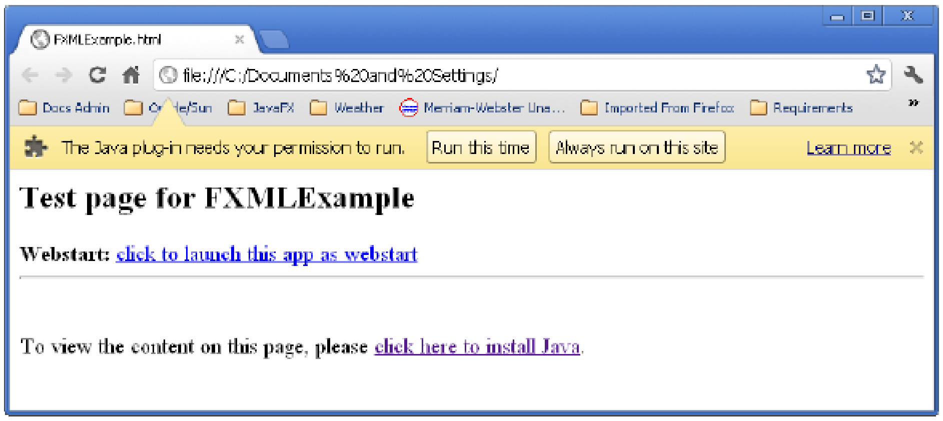
Running the Application Outside NetBeans IDE

To run the packaged application outside NetBeans IDE, go to the dist subdirectory of your application project directory and do the following:

- To run the application in standalone mode, double-click the JAR file.
- To run as a Web Start application, either click the link in the HTML page in your browser or double-click the JNLP file. The advantage of using the link in the browser is that the web page contains logic that uses the Deployment Toolkit, which checks to ensure that the minimum required version of both the Java and JavaFX Runtimes are installed on the user's system.
- To run the application in a web page, open the HTML file in a browser.

Note: If you open the HTML file in Google Chrome, you might have to click either **Run this time** or **Always run on this site** to enable the Java plug-in, as shown in Figure 6–3. Then reload the page to run the application.

Figure 6–3 Plug-In Message in Google Chrome



The Java plug-in needs your permission to run.

Test page for FXMLExample

Webstart: click to launch this app as webstart

To view the content on this page, please click here to install Java.

Note: For applications that include FXML markup, NetBeans IDE adds a digital signature to the JAR file by default to ensure that it will run on the web. For more information about packaging FXML applications and FXML deployment, see `http://docs.oracle.com/javafx/2/fxml_get_started/fxml_deployment.htm`

- To run a self-contained application package, go to the folder that contains the application. If you are running on a Windows or Linux platform, click the launcher file for the application. If you are running on a Mac platform, double click the application folder.

Deploying the Packaged Files

With the Java 7 Runtime and later, you can move the deployment files to other locations without changing any configuration properties in the deployment descriptor (which means the contents of the JNLP file). The files required for each deployment mode are shown in Table 6–1.

Table 6–1 Files Required for Each Deployment Mode

Deployment Mode	Files Required
Standalone	JAR
Run in Browser	JAR, JNLP, HTML
Web Start	JAR, JNLP, HTML
Self-Contained Application	Folder that contains the application and supporting files

Other Ways to Package JavaFX Applications

Instead of using NetBeans IDE to package your deployment files, you can also use the JavaFX Packager tool or a custom Ant task. All of these tools generate a default HTML file with application launch descriptors that you manually copy into your web page, but you can optionally generate the application launch descriptors directly into your own HTML pages using an input template. You can also customize many other aspects of application startup and execution. For more information, see *Deploying JavaFX*

Applications at
http://docs.oracle.com/javafx/2/deployment/jfxpub-deployment.htm