THE EXPERT'S VOICE® IN JAVA

JavaFX 2.0

Introduction by Example

DEVELOP HARDWARE-ACCELERATED, RICH-CLIENT APPLICATIONS USING JAVA

Carl Dea

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

JavaFX 2.0 is Java's next generation graphical user interface (GUI) toolkit for developers to rapidly build rich cross-platform applications. Built from the ground up, JavaFX takes advantage of modern GPUs through hardware-accelerated graphics while providing well-designed programming interfaces enabling developers to combine graphics, animation, and UI controls. The new JavaFX 2.0 is a pure Java language application programming interface (API).

The key architectural strategies provided by JavaFX 2.0 API are the reuse of existing Java libraries and the bridging of communication between other languages that run on the JVM (Visage, Jython, Groovy, JRuby, and Scala).

Nandini Ramani of Oracle plainly states the intended direction of JavaFX the platform in the following excerpt from the screencast, *Introducing JavaFX 2.0*:

> *"The industry is moving toward multi-core/multi-threading [type] platforms with GPUs. JavaFX 2.0 leverages these attributes to improve execution efficiency and UI design flexibility. Our initial goal is to give architects and developers of enterprise applications a set of tools and APIs to help them build better data driven business applications."*
>
> —Nandini Ramani
> Oracle Corp.
> VP of Development, Java Client Platform

Some History

In 2005, Sun Microsystems acquired the company SeeBeyond, at which a certain software engineer by the name of Chris Oliver created a graphics-rich scripting language known as F3 (Form Follows Function). F3 was later unveiled by Sun Microsystems at the 2007 JavaOne conference as JavaFX.

On April 20, 2009 Oracle Corporation announced the acquisition of Sun Microsystems, making Oracle the new steward of JavaFX. At JavaOne 2010, Oracle announced the JavaFX roadmap. As part of the road map, Oracle revealed its plans to phase out the JavaFX script language and re-create JavaFX for the Java language and platform.

As promised based on the 2010 roadmap, JavaFX 2.0 SDK was released at JavaOne October 3, 2011. Oracle also announced its commitment to take steps to release JavaFX as an open-source product, thus allowing the community to help move the platform forward. Open-sourcing JavaFX will increase its adoption, enable a quicker turnaround time on bug fixes, and generate new enhancements.

Table 0-1 shows the overall history of the major JavaFX releases.

Table 0-1. *Historical Timeline of Major JavaFX Releases*

Release Date	Version	Platform	Description
December 4, 2008	1.0	Windows and MacOS	JavaFX Script language, Production Suite, Media Playback
February 12, 2009	1.1	Windows and MacOS	New mobile development
June 2, 2009	1.2	Windows, MacOS, Linux, Solaris	Skinnable UI controls, Charting API, and performance improvements
April 22. 2010	1.3	Windows, MacOS, Linux, Solaris	JavaFX Composer, TV Emulator, Mobile Emulator
October 3, 2011	2.0	Windows, MacOS	Rewritten for the Java Language

Approach in This Book

The title of the book says it all: *JavaFX 2.0 Introduction by Example*. In this book, you will be learning the new JavaFX 2.0 capabilities by following practical recipe examples. These recipes will, in turn, provide you with the knowledge needed to create your own rich client applications. In the same tone with Java's mantra "Write once, run anywhere," JavaFX also preserves this same sentiment. Because JavaFX 2.0 is written entirely in Java the language, you will feel right at home.

Most of the recipes can be compiled and run under Java 6. However, some recipes will take advantage of Java 7's language enhancements, so Java 7 will be required. While working through this book with JavaFX 2.0 and Java 7, you will realize that the new APIs and language enhancements will help you to become a more productive developer. Having said this, I encourage you to explore all of Java 7's new capabilities. To delve deeper into the new capabilities of Java 7, I recommend the book, *Java 7 Recipes*. On an added note, the recipes in this book can also be found in *Java 7 Recipes*.

This book covers JavaFX 2.0's fundamentals, graphics and animations, audio and video, and the Web. The fundamentals include how to install prerequisite software (JavaFX 2.0, NetBeans 7.1) and create simple user interfaces. You will also learn the basics of the scene graph, text nodes and font styles, shapes, colors, layouts, menus, UI controls, simple styling (CSS styling), binding expressions, background processes, keyboard shortcuts, and dialog boxes. Next, in graphics and animations you will encounter image handling, drag-and-drop operations, animation APIs, and UI theming (Look 'n' Feel). After graphics and animations, you will learn audio and video. This section will include creating an MP3 player, using a video player, responding to media events, handling media marker events, and synchronizing an animation with media events. Finally, you will be using JavaFX 2.0 to interoperate with web technologies such as HTML5, JavaScript, and XML. In this section, you will be learning how to embed JavaFX into a web page, rendering and dynamically manipulating HTML5 content, creating a weather application to respond to HTML events, and creating an RSS feed application using an embedded database (Derby).

Who This Book Is For

If you are a Java developer who desires to take your client-side applications to the next level, you will find this book your guide to help you begin creating usable and aesthetically pleasing user interfaces. If you want a particular platform that is not listed in the preceding table, don't be too concerned because by the time you read this, JavaFX 2.0 should be available on your favorite OS.

How This Book Is Structured

This book is arranged in a natural progression that moves forward from beginner- to intermediate-level concepts. For the Java developer, none of the concepts mentioned in this book should be extremely difficult to figure out. This book's recipes are presented in a problem-solution format. After a brief description of a practical and real-world problem, a step-by-step solution will explain which techniques will be best suited to solve the problem. Each recipe can be easily adapted to meet your own needs when developing a game, media player, or your usual enterprise application. The more experienced Java UI developer you are, the more freedom you have to jump around to different chapters and recipes throughout the book. However, any Java developer can naturally progress through the book and learn the skills needed to enhance everyday GUI applications.

Downloading the Code

Source code is available for the examples in this book. You can download that code from this book's catalog page on the Apress web site. The URL is `http://www.apress.com/9781430242574`. The code will be in a `.zip` file that is organized by chapter.

References

Following are some online resources that will prove helpful as you begin your journey:

Introducing JavaFX 2.0, by Nandini Ramani: http://medianetwork.oracle.com/video/player/1191127359001

Chris Oliver's weblog: http://blogs.oracle.com/chrisoliver/entry/f3

JavaFX Roadmap: http://javafx.com/roadmap/

OpenJDK Discussion About JavaFX, by Richard Bair: http://fxexperience.com/2011/10/openjdk-discussion-about-javafx/

JavaFX on Wikipedia: http://en.wikipedia.org/wiki/JavaFX

CHAPTER 1

JavaFX Fundamentals

The JavaFX 2.0 API is Java's next generation GUI toolkit for developers to build rich cross-platform applications. JavaFX 2.0 is based on a scene graph paradigm (retained mode) as opposed to the traditional immediate mode style rendering. JavaFX's scene graph is a tree-like data structure that maintains vector-based graphic nodes. The goal of JavaFX is to be used across many types of devices such as mobile devices, smartphones, TVs, tablet computers, and desktops.

Before the creation of JavaFX, the development of rich Internet applications (RIAs) involved the gathering of many separate libraries and APIs to achieve highly functional applications. These separate libraries include Media, UI controls, Web, 3D, and 2D APIs. Because integrating these APIs together can be rather difficult, the talented engineers at Sun Microsystems (now Oracle) created a new set of JavaFX libraries that roll up all the same capabilities under one roof. JavaFX is the Swiss Army Knife of GUIs (see Figure 1-1). JavaFX 2.0 is a pure Java (language) API that allows developers to leverage existing Java libraries and tools.

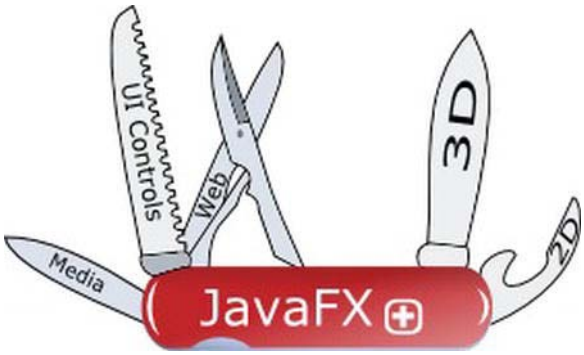

Figure 1-1. *JavaFX*

Depending on who you talk to, you will likely encounter different definitions of "user experience" (or in the UI world, UX). But one fact still remains; the users will always demand better content and increased usability from GUI applications. In light of this fact, developers and designers often work together to craft applications to fulfill this demand. JavaFX provides a toolkit that helps both the developer and designer (in some cases, they happen to be the same person) to create functional yet esthetically pleasing applications. Another thing to acknowledge is that if you are developing a game, media player, or the usual enterprise application, JavaFX will not only assist in developing richer UIs but you'll also find that the APIs are extremely well designed to greatly improve developer productivity (I'm all about the user of the API's perspective).

Although this book doesn't go through an exhaustive study of all of JavaFX 2.0's capabilities, you will find common use cases that can help you build richer applications. Hopefully, these recipes can lead

you in the right direction by providing practical and real-world examples. I also would like to encourage you to explore other resources to gain further insight into JavaFX. I highly recommend the book *Pro JavaFX Platform* (Apress, 2009) and the soon to released *Pro JavaFX 2.0 Platform* (Apress, 2012), which is an invaluable resource. These books go in depth to help you create professional grade applications.

So without further ado, let's get started, shall we?

1-1. Installing Required Software

Problem

You want to start developing JavaFX applications, but you don't know what software is required to be installed.

Solution

You'll need to install the following software in order to get started with JavaFX:

- Java 7 JDK or greater
- JavaFX 2.0 SDK
- NetBeans IDE 7.1 or greater

■ **Note** As of this writing, things are subject to change. To see additional requirements, refer to `http://download.oracle.com/javafx/2.0/system_requirements/jfxpub-system_requirements.htm`.

As of this writing, things are subject to change. By the time you read this; you will likely find JavaFX able to run on your favorite OS. For this recipe, I assume that Java 7 is already installed so I won't detail those installation steps. Following are steps to install all other required software components:

1. Download JavaFX 2.0 and NetBeans IDE 7.1.x from the following locations:
   - JavaFX 2.0 SDK: `http://www.oracle.com/technetwork/java/javafx/downloads/index.html`
   - NetBeans 7.1 beta SDK: `http://netbeans.org`
2. Install JavaFX 2.0 SDK. The screen in Figure 1-2 will appear once you've launched the JavaFX SDK Setup executable.

Once you have launched the JavaFX SDK setup executable you will see the start of the wizard in Figure 1-2.

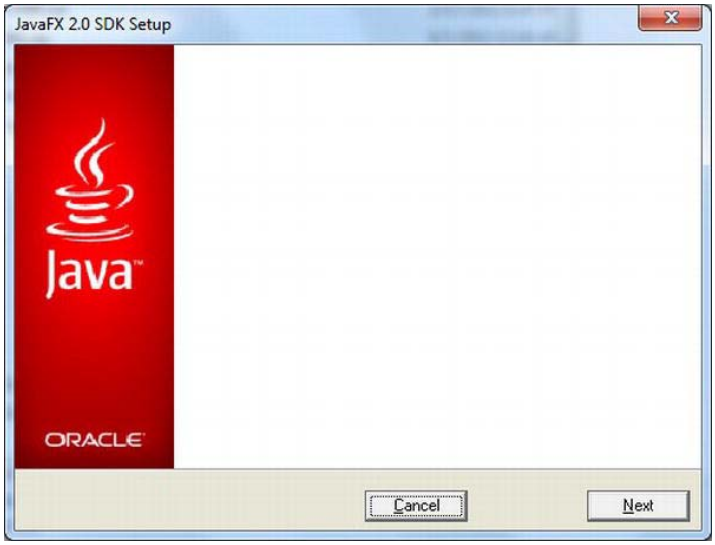

Figure 1-2. *JavaFX 2.0 SDK Setup Wizard*

3. Next, you can specify the home directory of the JavaFX SDK by clicking the Browse button. Figure 1-3 shows the default location for the JavaFX SDK's home directory. You might want to jot this location down in order to configure your CLASSPATH in Step 6.

Figure 1-3 displays Setup Options, which allow you to specify the JavaFX 2.0 SDK's home directory.

Figure 1-3. *JavaFX SDK home directory*

4. After you click Next, the components will install and the screen shown in Figure 1-4 will appear.

Figure 1-4 displays the progress indicator installing the last components before completing.

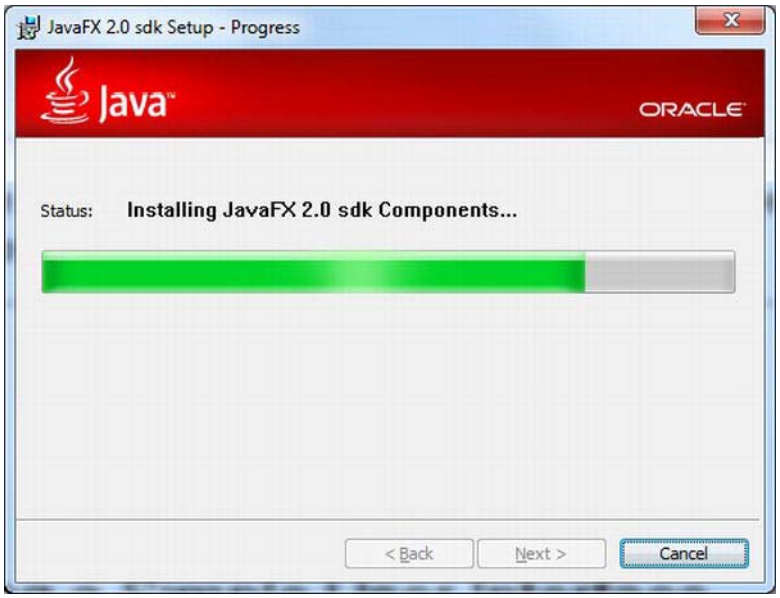

Figure 1-4. *Completing the install*

5. Install the NetBeans IDE, which includes the JavaFX 2.0 plug-in.

When installing, you will follow the default wizard screens. For additional instructions, you may refer to `http://netbeans.org/community/releases/71/install.html`.

6. Configuring your environment variable CLASSPATH to include the JavaFX runtime library. The name and location of the runtime library is at `<JavaFX SDK Home directory>\rt\lib\jfxrt.jar`. (Linux uses the forward slash: /).

How It Works

This recipe shows how to install Java FX 2.0 and the NetBeans IDE onto the Windows platform. You may need to modify your steps slightly when installing JavaFX 2.0 on other operating systems as they become available. Although the steps described here are for NetBeans, you can also develop using other IDEs such as Eclipse, IntelliJ, or vi. While most of the example recipes were created using the NetBeans IDE, you can also compile and run JavaFX applications using the command-line prompt.

To compile and run JavaFX applications using the command-line prompt you will need to configure your CLASSPATH. After you have followed the wizards to install the prerequisite software you will need to set your environment's `CLASSPATH` variable to include the JavaFX runtime library `<JavaFX SDK Home directory>/rt/lib/jfxrt.jar` (Step 6). Setting this library will later assist in compiling and running JavaFX-based applications on the command-line. The following code configures your CLASSPATH environment variable based on your platform:

Setting CLASSPATH on Windows Platforms

```
set JAVAFX_HOME=C:\Program Files (x86)\Oracle\JavaFX 2.0 SDK
set JAVA_HOME=C:\Program Files (x86)\Java\jdk1.7.0
set CLASSPATH=%JAVAFX_HOME%\rt\lib\jfxrt.jar;.
```

Setting CLASSPATH on UNIX/Linux/Mac OS platforms

```
# bash environments
export JAVAFX_HOME=<JavaFX SDK Home>
export CLASSPATH=$CLASSPATH:$JAVAFX_HOME/rt/lib/jfxrt.jar

#csh environments
setenv JAVAFX_HOME <JavaFX SDK Home>
setenv CLASSPATH ${CLASSPATH}:${JAVAFX_HOME}/rt/lib/jfxrt.jar
```

In recipe 1-2 you will learn how to create a simple Hello World application. Once your Hello World application is created, you will be able to compile and run a JavaFX-based application.

1-2. Creating a Simple User Interface

Problem

You want to create, code, compile, and run a simple JavaFX Hello World application.

Solution #1

Develop a JavaFX HelloWorld application using the JavaFX project creation wizard in the NetBeans IDE.

CREATING A JAVAFX HELLO WORLD APPLICATION IN NETBEANS

To quickly get started with creating, coding, compiling, and running a simple JavaFX HelloWorld application using the NetBeans IDE, follow these steps:

Launch NetBeans IDE.

1) On the File menu, select New Project.

2) Under Choose Project and Categories, select the JavaFX folder.

3) Under Projects, select Java FX Application, and click Next.

4) Specify **HelloWorldMain** for your project name.

5) Change or accept the defaults for the Project Location and Project Folder fields.

6) Make sure the Create Application Class check box option is selected. Click Finish.

7) In the NetBeans IDE on the Projects tab, select the newly created project. Open the Project Properties dialog box to verify that the Source/Binary format settings are JDK 7. Click Sources under Categories.

8) While still in the Project Properties dialog box, under Categories, select Libraries to verify that the Java 7 and JavaFX platform are configured properly. Click the Manage Platforms button. Make sure a tab showing JavaFX libraries appears. Figure 1-5 depicts the JavaFX tab detailing its SDK home, Runtime, and Javadoc directory locations. Once verified, click the Close button.

Figure 1-5 shows the Java Platform Manager window containing JavaFX as a managed platform included with JDK 7.

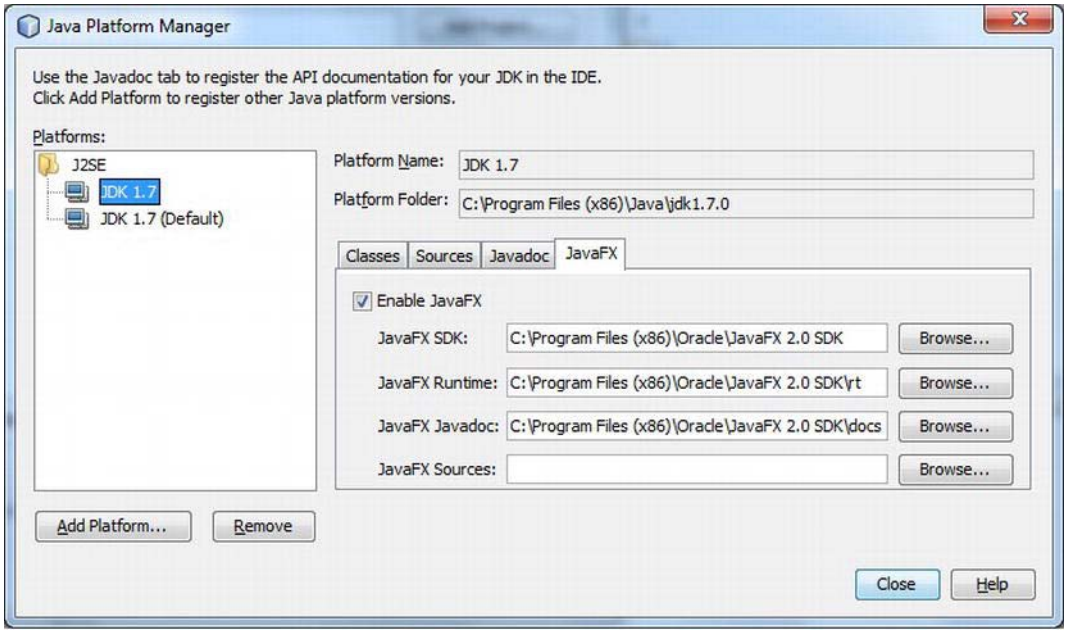

Figure 1-5. *Java Platform Manager*

9) After closing the Java Platform Manager window, click OK to close the Project Properties window.

10) To run and test your JavaFX Hello World application, access the Run menu, and select Run Main Project or hit the F6 key.

Shown in Figure 1-6 is a simple JavaFX Hello World application launched from the NetBeans IDE.

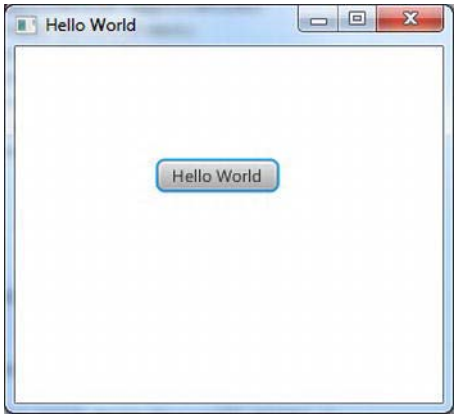

Figure 1-6. *JavaFX Hello World launched from the NetBeans IDE*

Solution #2

Use your favorite editor to code your JavaFX Hello World application. Once the Java file is created you will use the command-line prompt to compile and run your JavaFX application. Following are the steps to create a JavaFX Hello World application to be compiled and run on the command-line prompt.

CREATING A JAVAFX HELLO WORLD APPLICATION IN ANOTHER IDE

To quickly get started:

1. Copy and paste the following code into your favorite editor and save the file as `HelloWorldMain.java`.

The following source code is a JavaFX Hello World application:

```
package helloworldmain;

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

/**
 *
 * @author cdea
 */
public class HelloWorldMain extends Application {
```

```
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");
        Group root = new Group();
        Scene scene = new Scene(root, 300, 250);
        Button btn = new Button();
        btn.setLayoutX(100);
        btn.setLayoutY(80);
        btn.setText("Hello World");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            public void handle(ActionEvent event) {
                System.out.println("Hello World");
            }
        });
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

2. After saving the file named `HelloWorldMain.java`, on the command-line prompt you will navigate to the directory location of the file.

3. Compile the source code file `HelloWorldMain.java` using the Java compiler `javac`:

```
javac -d . HelloWorldMain.java
```

4. Run and test your JavaFX Hello World application. Assuming you are located in the same directory as the `HelloWorldMain.java` file, type the following command to run your JavaFX Hello World application from the command-line prompt:

```
java helloworldmain.HelloWorldMain
```

Shown in Figure 1-7 is a simple JavaFX Hello World application launched from the command-line prompt.

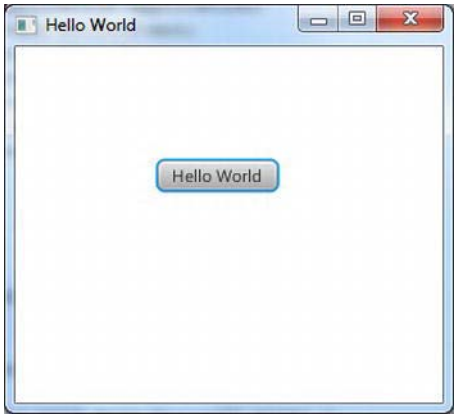

Figure 1-7. *JavaFX Hello World launched from the command-line prompt*

How It Works

Following are descriptions of the two solutions. Both solutions require prerequisite software. (I cover how to install required software in recipe 1-1.) In Solution #1 you will be creating a JavaFX application using the NetBeans IDE. Solution #2 allows you to choose your favorite editor and use the command-line prompt to compile and execute JavaFX programs.

Solution #1

To create a simple JavaFX Hello World application, using the NetBeans you will use the JavaFX project creation wizard as specified in Steps 1 through 7. In Steps 8 through 10, you will verify two settings to ensure that the project is configured to compile and run JavaFX 2.0 applications properly. Finally, in Step 11 you will run the JavaFX Hello World application by selecting the Run Main Project menu option.

You shouldn't encounter any difficulty when following Steps 1 through 7. However, Steps 8 through 10 address a minor NetBeans bug that has to do with setting your project source/binary format to JDK 7 and making sure that the managed platform includes the JavaFX runtime libraries. If you are not experiencing this issue, the NetBeans team may have already corrected the problem. To be on the safe side, it wouldn't hurt to follow Steps 8 through 10 to verify your configurations before you begin.

Solution #2

To create a simple JavaFX Hello World application using your favorite IDE, follow Steps 1 and 2. To compile and run your Hello World program on the command line, follow Steps 3 and 4.

Once the source code is entered into your favorite editor and the source file has been saved, you will want to compile and run your JavaFX program. Open the command-line prompt window and navigate to the directory location of the Java file named `HelloWorldMain.java`.

Here I would like to point out the way you compile the file using the command `javac -d . HelloWorldMain.java`. You will notice the `-d .` before the file name. This lets the Java compiler know where to put class files based on their package name. In this scenario, the `HelloWorldMain` package statement is `helloworldmain`, which will create a subdirectory under the current directory. When finished compiling, your directory structure should resemble the following:

```
|projects
        |helloworld
              |HelloWorldMain.java
              | helloworldmain
                    |HelloWorldMain.class
```

With the preceding directory structure in mind, the following commands will compile and run our JavaFX Hello World application:

```
cd /projects/helloworld

javac -d . HelloWorldMain.java

java helloworldmain.HelloWorldMain
```

■ **Note** There are many ways to package and deploy JavaFX applications. To learn more, please see "Learning how to deploy and package JavaFX applications" at `http://blogs.oracle.com/thejavatutorials/entry/javafx_2_0_beta_packager`. For in-depth JavaFX deployment strategies, see Oracle's "Deploying JavaFX Applications" at `http://download.oracle.com/javafx/2.0/deployment/deployment_toolkit.htm`.

In both solutions you'll notice in the source code that JavaFX applications extend the `javafx.application.Application` class. The `Application` class provides application life cycle functions such as launching and stopping during runtime. This also provides a mechanism for Java applications to launch JavaFX GUI components in a threadsafe manner. Keep in mind that synonymous to Java Swing's event dispatch thread, JavaFX will have its own JavaFX application thread.

In our `main()` method's entry point we launch the JavaFX application by simply passing in the command line arguments to the `Application.launch()` method. Once the application is in a ready state, the framework internals will invoke the `start()` method to begin. When the `start()` method is invoked, a JavaFX `javafx.stage.Stage` object is available for the developer to use and manipulate.

You'll notice that some objects are oddly named, such as `Stage` or `Scene`. The designers of the API have modeled things similar to a theater or a play in which actors perform in front of an audience. With this same analogy, in order to show a play, there are basically one-to-many scenes that actors perform in. And, of course, all scenes are performed on a stage. In JavaFX the `Stage` is equivalent to an application window similar to Java Swing API `JFrame` or `JDialog`. You may think of a `Scene` object as a content pane capable of holding zero-to-many `Node` objects. A `Node` is a fundamental base class for all scene graph nodes to be rendered. Commonly used nodes are UI controls and `Shape` objects. Similar to a tree data structure, a scene graph will contain children nodes by using a container class `Group`. We'll learn more about the `Group` class later when we look at the `ObservableList`, but for now we can think of them as Java `List`s or `Collection`s that are capable of holding `Node`s.

Once the child nodes have been added, we set the `primaryStage`'s (`Stage`) scene and call the `show()` method on the `Stage` object to show the JavaFX window.

One last thing: in this chapter most of the example applications will be structured the same as this example in which recipe code solutions will reside inside the `start()` method. Having said this, most of the recipes in this chapter will follow the same pattern. In other words, for the sake of brevity, much of

the boiler plate code will not be shown. To see the full source listings of all the recipes, please download the source code from the book's web site.

1-3: Drawing Text

Problem

You want to draw text onto the JavaFX scene graph.

Solution

Create `Text` nodes to be placed on the JavaFX scene graph by utilizing the `javafx.scene.text.Text` class. As `Text` nodes are to be placed on the scene graph, you decide you want to create randomly positioned `Text` nodes rotated around their (x, y) positions scattered about the scene area.

The following code implements a JavaFX application that displays `Text` nodes scattered about the scene graph with random positions and colors:

```
package javafx2introbyexample.chapter1.recipe1_03;

import java.util.Random;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.stage.Stage;

/**
 *
 * @author cdea
 */
public class DrawingText extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chapter 1-3 Drawing Text");
        Group root = new Group();
        Scene scene = new Scene(root, 300, 250, Color.WHITE);
        Random rand = new Random(System.currentTimeMillis());
        for (int i = 0; i < 100; i++) {
            int x = rand.nextInt((int) scene.getWidth());
            int y = rand.nextInt((int) scene.getHeight());
```

```
            int red = rand.nextInt(255);
            int green = rand.nextInt(255);
            int blue = rand.nextInt(255);

            Text text = new Text(x, y, "JavaFX 2.0");

            int rot = rand.nextInt(360);
            text.setFill(Color.rgb(red, green, blue, .99));
            text.setRotate(rot);
            root.getChildren().add(text);
        }

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Figure 1-8 shows random `Text` nodes scattered about the JavaFX scene graph.

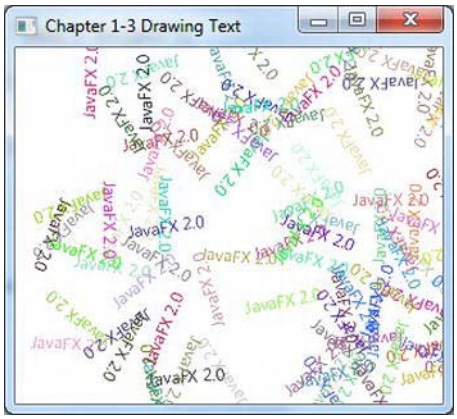

Figure 1-8. *Drawing text*

How It Works

To draw text in JavaFX you will be creating a `javafx.scene.text.Text` node to be placed on the scene graph (`javafx.scene.Scene`). In this example you'll notice text objects with random colors and positions scattered about the `Scene` area.

First, we create a loop to generate random (x, y) coordinates to position `Text` nodes. Second, we create random color components between (0–255 rgb) to be applied to the `Text` nodes. Third, the rotation angle (in degrees) is a randomly generated value between (0–360 degrees) to cause the text to be slanted. The following code creates random values that will be assigned to a `Text` node's position, color, and rotation:

```
int x = rand.nextInt((int) scene.getWidth());
int y = rand.nextInt((int) scene.getHeight());
int red = rand.nextInt(255);
```

```
int green = rand.nextInt(255);
int blue = rand.nextInt(255);
int rot = rand.nextInt(360);
```

Once the random values are generated, they will be applied to the `Text` nodes, which will be drawn onto the scene graph. The following code snippet applies position (x, y), color (rgb), and rotation (angle in degrees) onto the `Text` node:

```
Text text = new Text(x, y, "JavaFX 2.0");
text.setFill(Color.rgb(red, green, blue, .99));
text.setRotate(rot);

root.getChildren().add(text);
```

You will begin to see the power of the scene graph API by its ease of use. `Text` nodes can be easily manipulated as if they were `Shapes`. Well, actually they really are `Shapes`. Defined in the inheritance hierarchy, `Text` nodes extend from the `javafx.scene.shape.Shape` class and are therefore capable of doing interesting things such as being filled with colors or rotated about an angle. Although the text is colorized, they still tend to be somewhat boring. However, in the next recipe we will demonstrate how to change a text's font.

1-4: Changing Text Fonts

Problem

You want to change text fonts and add special effect to `Text` nodes.

Solution

Create a JavaFX application that uses the following classes to set the text font and apply effects on `Text` nodes:

- `javafx.scene.text.Font`
- `javafx.scene.effect.DropShadow`
- `javafx.scene.effect.Reflection`

The code that follows sets the font and applies effects to `Text` nodes. We will be using the Serif, SanSerif, Dialog, and Monospaced fonts along with the drop shadow and reflection effects:

```
package javafx2introbyexample.chapter1.recipe1_04;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.effect.Reflection;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
```

```
import javafx.stage.Stage;

/**
 * Changing Text Fonts
 * @author cdea
 */
public class ChangingTextFonts extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chapter 1-4 Changing Text Fonts");
        Group root = new Group();
        Scene scene = new Scene(root, 550, 250, Color.WHITE);

        // Serif with drop shadow
        Text text2 = new Text(50, 50, "JavaFX 2.0: Intro. by Example");
        Font serif = Font.font("Serif", 30);
        text2.setFont(serif);
        text2.setFill(Color.RED);
        DropShadow dropShadow = new DropShadow();
        dropShadow.setOffsetX(2.0f);
        dropShadow.setOffsetY(2.0f);
        dropShadow.setColor(Color.rgb(50, 50, 50, .588));
        text2.setEffect(dropShadow);
        root.getChildren().add(text2);


        // SanSerif
        Text text3 = new Text(50, 100, "JavaFX 2.0: Intro. by Example");
        Font sanSerif = Font.font("SanSerif", 30);
        text3.setFont(sanSerif);
        text3.setFill(Color.BLUE);
        root.getChildren().add(text3);

        // Dialog
        Text text4 = new Text(50, 150, "JavaFX 2.0: Intro. by Example");
        Font dialogFont = Font.font("Dialog", 30);
        text4.setFont(dialogFont);
        text4.setFill(Color.rgb(0, 255, 0));
        root.getChildren().add(text4);

        // Monospaced
        Text text5 = new Text(50, 200, "JavaFX 2.0: Intro. by Example");
        Font monoFont = Font.font("Monospaced", 30);
        text5.setFont(monoFont);
```

```
        text5.setFill(Color.BLACK);
        root.getChildren().add(text5);

        Reflection refl = new Reflection();
        refl.setFraction(0.8f);
        text5.setEffect(refl);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Figure 1-9 shows the JavaFX application setting various font styles and applying effects (drop shadow and reflection) to the `Text` nodes.

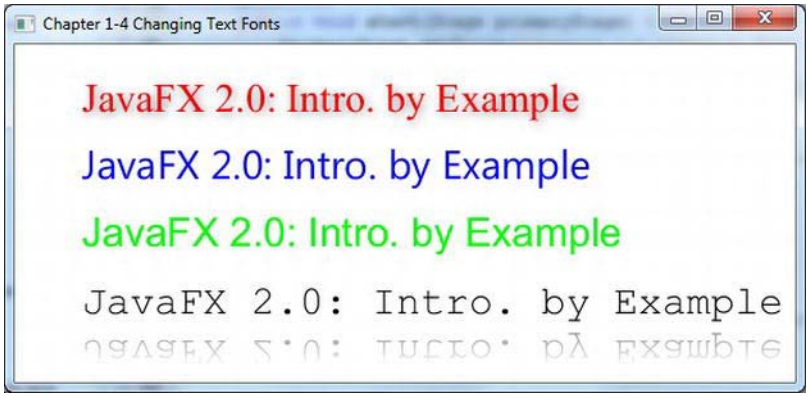

Figure 1-9. *Changing text fonts*

How It Works

In this recipe, I basically used JavaFX's scene graph to display `Text` nodes. JavaFX takes a retained mode approach, in which nodes use vector-based graphics. Vector-based graphics allow you to scale shapes and apply effects without issues of pixilation (jaggies). In each `Text` node you can create and set the font to be rendered onto the scene graph. Here is the code to create and set the font on a `Text` node:

```
Text text2 = new Text(50, 50, "JavaFX 2.0: Intro. by Example");
Font serif = Font.font("Serif", 30);
text2.setFont(serif);
```

The drop shadow is a real effect (`DropShadow`) object and actually applied to a single `Text` node instance. The `DropShadow` object is set to be positioned based on an x and y offset in relation to the `Text` node. Also we can set the color of the shadow; here we set it to gray with a .588 opacity. Following is an example of setting a `Text` node's effect property with a drop shadow effect (`DropShadow`):

```
DropShadow dropShadow = new DropShadow();
dropShadow.setOffsetX(2.0f);
```

```
dropShadow.setOffsetY(2.0f);
dropShadow.setColor(Color.rgb(50, 50, 50, .588));
text2.setEffect(dropShadow);
```

Although this is about setting text fonts, we applied effects to `Text` nodes. I've added yet another effect (just kicking it up a notch). While creating the last `Text` node using the monospaced font, I applied the popular reflection effect. Here it is, set so that .8 or 80 percent of the reflection will be shown. The reflection values range from zero (0%) to one (100%). The following code snippet implements a reflection of 80% with a float value of 0.8f:

```
Reflection refl = new Reflection();
refl.setFraction(0.8f);

text5.setEffect(refl);
```

1-5. Creating Shapes

Problem

You want to create shapes to be placed on the scene graph.

Solution

Use JavaFX's `Arc`, `Circle`, `CubicCurve`, `Ellipse`, `Line`, `Path`, `Polygon`, `Polyline`, `QuadCurve`, `Rectangle`, `SVGPath`, and `Text` classes in the `javafx.scene.shape.*` package. You may also use builder classes associated with each shape in the `javafx.builders.*` package.

The following code draws various complex shapes. The first complex shape involves a cubic curve drawn in the shape of a sine wave. The next shape, which I would like to call the *ice cream cone*, uses the path class that contains path elements (`javafx.scene.shape.PathElement`). The third shape is a Quadratic Bézier curve (`QuadCurve`) forming a smile. Our final shape is a delectable donut. We create this donut shape by subtracting two ellipses (one smaller and one larger):

```
// CubicCurve
CubicCurve cubicCurve = CubicCurveBuilder.create()
        .startX(50).startY(75)          // start pt (x1,y1)
        .controlX1(80).controlY1(-25)   // control pt1
        .controlX2(110).controlY2(175)  // control pt2
        .endX(140).endY(75)             // end pt (x2,y2)
        .strokeType(StrokeType.CENTERED).strokeWidth(1)
        .stroke(Color.BLACK)
        .strokeWidth(3)
        .fill(Color.WHITE)
        .build();
root.getChildren().add(cubicCurve);

// Ice cream
Path path = new Path();

MoveTo moveTo = new MoveTo();
```

```
moveTo.setX(50);
moveTo.setY(150);

QuadCurveTo quadCurveTo = new QuadCurveTo();
quadCurveTo.setX(150);
quadCurveTo.setY(150);
quadCurveTo.setControlX(100);
quadCurveTo.setControlY(50);

LineTo lineTo1 = new LineTo();
lineTo1.setX(50);
lineTo1.setY(150);

LineTo lineTo2 = new LineTo();
lineTo2.setX(100);
lineTo2.setY(275);

LineTo lineTo3 = new LineTo();
lineTo3.setX(150);
lineTo3.setY(150);
path.getElements().add(moveTo);
path.getElements().add(quadCurveTo);
path.getElements().add(lineTo1);
path.getElements().add(lineTo2);
path.getElements().add(lineTo3);
path.setTranslateY(30);
path.setStrokeWidth(3);
path.setStroke(Color.BLACK);

root.getChildren().add(path);

// QuadCurve create a smile
QuadCurve quad =QuadCurveBuilder.create()
        .startX(50)
        .startY(50)
        .endX(150)
        .endY(50)
        .controlX(125)
        .controlY(150)
        .translateY(path.getBoundsInParent().getMaxY())
        .strokeWidth(3)
        .stroke(Color.BLACK)
        .fill(Color.WHITE)
        .build();

root.getChildren().add(quad);

// outer donut
Ellipse bigCircle = EllipseBuilder.create()
        .centerX(100)
        .centerY(100)
        .radiusX(50)
```

```
                .radiusY(75/2)
                .translateY(quad.getBoundsInParent().getMaxY())
                .strokeWidth(3)
                .stroke(Color.BLACK)
                .fill(Color.WHITE)
                .build();

        // donut hole
        Ellipse smallCircle = EllipseBuilder.create()
                .centerX(100)
                .centerY(100)
                .radiusX(35/2)
                .radiusY(25/2)
                .build();

        // make a donut
        Shape donut = Path.subtract(bigCircle, smallCircle);
        // orange glaze
        donut.setFill(Color.rgb(255, 200, 0));

        // add drop shadow
        DropShadow dropShadow = new DropShadow();
        dropShadow.setOffsetX(2.0f);
        dropShadow.setOffsetY(2.0f);
        dropShadow.setColor(Color.rgb(50, 50, 50, .588));
        donut.setEffect(dropShadow);

        // move slightly down
        donut.setTranslateY(quad.getBoundsInParent().getMinY() + 30);

        root.getChildren().add(donut);
```

Figure 1-10 displays the sine wave, ice cream cone, smile, and donut shapes that we have created using JavaFX:

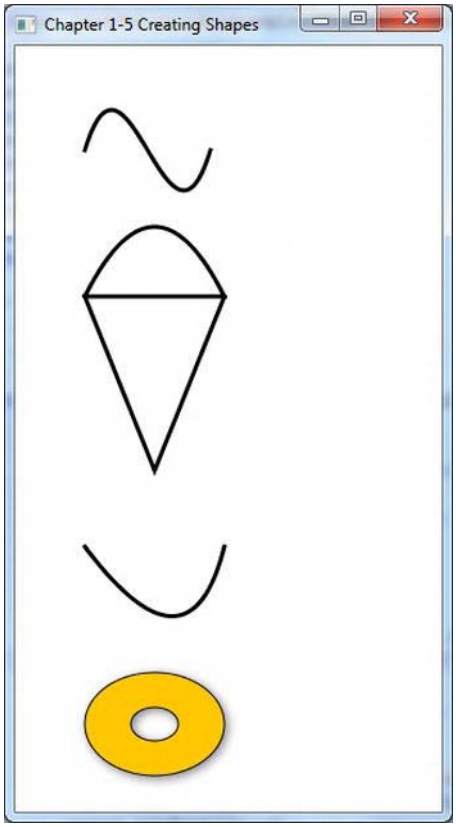

Figure 1-10. *Creating shapes*

How It Works

The first shape is a `javafx.scene.shape.CubicCurve` class. To create a cubic curve, you simply look for the appropriate constructor to be instantiated. A cubic curve's main parameters that you will be setting are its start X, start Y, control point 1 X, control point 1 Y, control point 2 X, control point 2 Y, end X, end Y. The startX`, startY, endX, endY` parameters are the starting point and ending point of a curve. The controlX1, controlY1, controlX2, controlY2 denote control point 1 and control point 2. A control point is a point that pulls the curve towards the direction of the point itself. In our example, we simply have a `control point 1` above to pull the curve upward to form a hill and `control point 2` below to pull the curve downward to form a valley. The following illustration (Figure 1-11) depicts a Cubic Curve with control points influencing the curve:

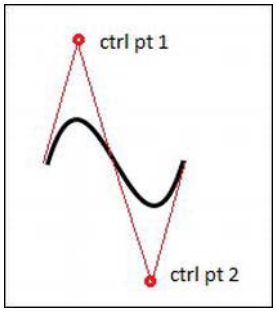

Figure 1-11. *Cubic Curve*

The following code snippet is used to create a `javafx.scene.shape.CubicCurve` instance:

```
CubicCurve cubicCurve = new CubicCurve();
cubicCurve.setStartX(50);     // start pt (x1,y1)
cubicCurve.setStartY(75);
cubicCurve.setControlX1(80);  // control pt1
cubicCurve.setControlY1(-25);
cubicCurve.setControlX2(110); // control pt2
cubicCurve.setControlY2(175);
cubicCurve.setEndX(140);      // end pt (x2,y2)
cubicCurve.setEndY(75);
```

But, right off the bat in the source code listing in the solution section, you'll notice that I didn't use the usual `new CubicCurve()` constructor like the previous snippet, but instead I use a class having a suffix of `Builder` on the end of it. `Builder` classes are convenience classes that follow a design pattern called the Builder pattern. `Builder` classes provide a way to method chain invocations by enabling the developer to specify attributes in an ad hoc way (declarative). This makes code more readable and less verbose, thus increasing developer productivity. When using this facility while developing graphics applications, you may also find that coding tends to be more expressive and reminiscent of declarative type languages (Visage, Groovy, Scala, and Python).

Back to `CubicCurveBuilder`; we begin with the `create()` method that will instantiate a `Builder` class. Next is specifying a cubic curve's attributes in any order. Similar to mutators or setter methods, you simply pass a single value to the method. The convention is that the `set` prefixed on the method is removed, and the method returns the `this` pointer of the builder object instance. By returning itself it allows you to continue to use the dot notation to specify parameters, thus the method chaining behavior. Once finished with specifying values on the `Builder` class, a call to the `build()` method will return an instance of the desired class (in this case, the `CubicCurve` class).

The ice cream cone shape is created using the `javafx.scene.shape.Path` class. As each path element is created and added to the `Path` object, each element is *not* considered a graph node (`javafx.scene.Node`). This means they do not extend from the `javafx.scene.shape.Shape` class and cannot be a child node in a `scene` graph to be displayed. When looking at the Javadoc, you will notice that a `Path` class extends from the `Shape` class that extends from the (`javafx.scene.Node`) class, and therefore a `Path` is a graph node, but path elements do not extend from the `Shape` class. Path elements actually extend from the `javafx.scene.shape.PathElement` class, which is only used in the context of a

Path object. So you won't be able to instantiate a LineTo class to be put in the scene graph. Just remember that the classes with To as a suffix is a path element, not a real Shape node. For example, the MoveTo and LineTo object instances are Path elements added to a Path object, not shapes that can be added to the scene. Shown following are Path elements added to a Path object to draw an ice cream cone:

```
// Ice cream
Path path = new Path();

MoveTo moveTo = new MoveTo();
moveTo.setX(50);
moveTo.setY(150);

...// Additional Path Elements created.
LineTo lineTo1 = new LineTo();
lineTo1.setX(50);
lineTo1.setY(150);

...// Additional Path Elements created.

path.getElements().add(moveTo);
path.getElements().add(quadCurveTo);
path.getElements().add(lineTo1);
```

Rendering the QuadCurve (smile) object I used the QuadCurveBuilder class similar to the CubicCurveBuilder class, and you'll notice the simplicity of creating such a shape. This is similar to the cubic curve example described above in the first shape. Instead of two control points you only have one control point. Shown below (Figure 1-12) is a QuadCurve shape with a control point below its starting and ending points:

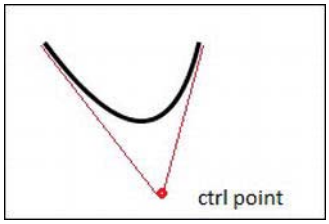

Figure 1-12. *Quadratic Curve*

Once your builder class is complete you will finish things off by invoking the build() method. Shown below is a quadratic curve with a stroke thickness of three pixels filled with the color white:

```
// QuadCurve create a smile
QuadCurve quad =QuadCurveBuilder.create()
        .startX(50)
        .startY(50)
        .endX(150)
        .endY(50)
```

```
        .controlX(125)
        .controlY(150)
        .strokeWidth(3)
        .stroke(Color.BLACK)
        .fill(Color.WHITE)
        .build();
```

Last is our tasty donut shape with a drop shadow effect. When creating the donut, we begin by creating two circular ellipses. By subtracting the smaller ellipse (donut hole) from the larger ellipse area, a newly derived `Shape` object is created and returned using the `Path.subtract()` method. Following is the code snippet that creates the donut shape using the `Path.subtract()` method:

```
// outer donut
Ellipse bigCircle = ...//Outer shape area

// donut hole
Ellipse smallCircle = ...// Inner shape area

// make a donut
Shape donut = Path.subtract(bigCircle, smallCircle);
```

Next, is applying a drop shadow effect onto our donut. A common technique is to draw the shape filled black while the original shape is laid on top slightly offset to appear as a shadow. However, in JavaFX we draw it once and use the `setEffect()` method to apply a `DropShadow` object instance. To cast the shadow offset call the `setOffsetX()` and `setOffsetY()` methods.

One last thing to point out is that all shapes are drawn to be positioned underneath one another. As each shape was created, you'll notice their translateY property was set to reposition or shift the shape from its original position. For example, if a shape's upper left bounding box point is created at (100, 100) and you want it to be moved to (101, 101) the translateX and translateY properties would be set to one.

1-6. Assigning Colors to Objects

Problem

You want to fill your shapes with simple colors and gradient colors.

Solution

In JavaFX, all shapes can be filled with simple colors and gradient colors. The following are the main classes used to fill shape nodes:

- `javafx.scene.paint.Color`
- `javafx.scene.paint.LinearGradient`
- `javafx.scene.paint.Stop`
- `javafx.scene.paint.RadialGradient`

The following code uses the preceding classes to add radial and linear gradient colors as well as transparent (alpha channel level) colors to our shapes. We will be using an ellipse, rectangle, and rounded rectangle in our recipe. A solid black line (as depicted in Figure 1-13) also appears in our recipe to demonstrate the transparency of our shape's color.

```
primaryStage.setTitle("Chapter 1-6 Assigning Colors To Objects");
Group root = new Group();
Scene scene = new Scene(root, 350, 300, Color.WHITE);


Ellipse ellipse = new Ellipse(100, 50 + 70/2, 50, 70/2);
RadialGradient gradient1 = RadialGradientBuilder.create()
        .focusAngle(0)
        .focusDistance(.1)
        .centerX(80)
        .centerY(45)
        .radius(120)
        .proportional(false)
        .cycleMethod(CycleMethod.NO_CYCLE)
        .stops(new Stop(0, Color.RED), new Stop(1, Color.BLACK))
        .build();

ellipse.setFill(gradient1);
root.getChildren().add(ellipse);

Line blackLine = LineBuilder.create()
        .startX(170)
        .startY(30)
        .endX(20)
        .endY(140)
        .fill(Color.BLACK)
        .strokeWidth(10.0f)
        .translateY(ellipse.prefHeight(-1) + ellipse.getLayoutY() + 10)
        .build();

root.getChildren().add(blackLine);


Rectangle rectangle = RectangleBuilder.create()
        .x(50)
        .y(50)
        .width(100)
        .height(70)
        .translateY(ellipse.prefHeight(-1) + ellipse.getLayoutY() + 10)
        .build();

LinearGradient linearGrad = LinearGradientBuilder.create()
        .startX(50)
        .startY(50)
        .endX(50)
        .endY(50 + rectangle.prefHeight(-1) + 25)
```

```
            .proportional(false)
            .cycleMethod(CycleMethod.NO_CYCLE)
            .stops( new Stop(0.1f, Color.rgb(255, 200, 0, .784)),
                    new Stop(1.0f, Color.rgb(0, 0, 0, .784)))
            .build();

    rectangle.setFill(linearGrad);
    root.getChildren().add(rectangle);

    Rectangle roundRect = RectangleBuilder.create()
            .x(50)
            .y(50)
            .width(100)
            .height(70)
            .arcWidth(20)
            .arcHeight(20)
            .translateY(ellipse.prefHeight(-1) +
                        ellipse.getLayoutY() +
                        10 +
                        rectangle.prefHeight(-1) +
                        rectangle.getLayoutY() + 10)
            .build();



    LinearGradient cycleGrad = LinearGradientBuilder.create()
            .startX(50)
            .startY(50)
            .endX(70)
            .endY(70)
            .proportional(false)
            .cycleMethod(CycleMethod.REFLECT)
            .stops(new Stop(0f, Color.rgb(0, 255, 0, .784)),
                   new Stop(1.0f, Color.rgb(0, 0, 0, .784)))
            .build();

    roundRect.setFill(cycleGrad);
    root.getChildren().add(roundRect);

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

Figure 1-13 displays the various types of colorized fills that can be applied onto shapes.

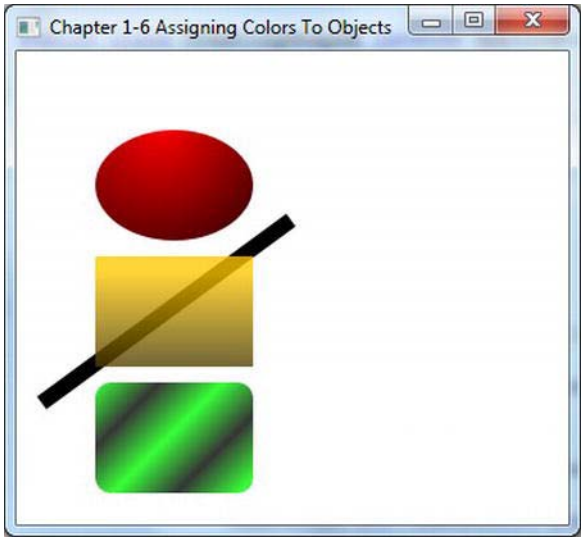

Figure 1-13. *Color shapes*

How It Works

Figure 1-13 shows shapes displayed from top to bottom starting with an ellipse, rectangle, and a rounded rectangle having colored gradient fills. When drawing the eclipse shape you will be using a radial gradient that appears as if it were a 3D spherical object. Next, you will be creating a rectangle filled with a yellow semitransparent linear gradient. A thick black line shape was drawn behind the yellow rectangle to demonstrate the rectangle's semitransparent color. Last, you will implement a rounded rectangle filled with a green-and-black reflective linear gradient resembling 3D tubes in a diagonal direction.

The amazing thing about colors with gradients is that they can often make shapes appear three-dimensional. Gradient paint allows you to interpolate between two or more colors, which gives depth to the shape. JavaFX provides two types of gradients: a radial (`RadialGradient`) and a linear (`LinearGradient`) gradient. For our ellipse shape you will be using a radial gradient (`RadialGradient`).

I created Table 1-1 from the JavaFX 2.0 Javadoc definitions found for the `RadialGradient` class (`http://download.oracle.com/javafx/2.0/api/javafx/scene/paint/RadialGradient.html`).

Table 1-1. *RadialGradient Properties*

Property	Data Type	Description
`focusAngle`	`double`	Angle in degrees from the center of the gradient to the focus point to which the first color is mapped
`focusDistance`	`double`	Distance from the center of the gradient to the focus point to which the first color is mapped

Property	Data Type	Description
centerX	double	X coordinate of the center point of the gradient's circle
centerY	double	Y coordinate of the center point of the gradient's circle
radius	double	Radius of the circle defining the extents of the color gradient
proportional	boolean	Coordinates and sizes are proportional to the shape which this gradient fills
cycleMethod	CycleMethod	Cycle method applied to the gradient
stops	List<Stop>	Gradient's color specification

In our recipe the focus angle is set to zero, distance is set to .1, center X and Y is set to (80,45), radius is set to 120 pixels, proportional is set to false, cycle method is set to the no cycle (`CycleMethod.NO_CYCLE`), and two color stop values set to red (`Color.RED`) and black (`Color.BLACK`). These settings give a radial gradient to our ellipse by starting with the color red with a center position of (80, 45) (upper left of the ellipse) that interpolates to the color black with a distance of 120 pixels (radius).

Next, you will be creating a rectangle having a yellow semitransparent linear gradient. For our yellow rectangle you will be using linear gradient (`LinearGradient`) paint.

I created Table 1-2 from the JavaFX 2.0 Javadoc definitions found for the `LinearGradient` class (`http://download.oracle.com/javafx/2.0/api/javafx/scene/paint/LinearGradient.html`).

Table 1-2. *LinearGradient Properties*

Property	Data Type	Description
startX	double	X coordinate of the gradient axis start point
startY	double	Y coordinate of the gradient axis start point
endX	double	X coordinate of the gradient axis end point
endY	double	Y coordinate of the gradient axis end point
proportional	boolean	Whether the coordinates are proportional to the shape which this gradient fills
cycleMethod	CycleMethod	Cycle method applied to the gradient
stops	List<Stop>	Gradient's color specification

To create a linear gradient paint you will specify the `startX`, `startY`, `endX`, and `endY` for the start and end points. The start and end point coordinates denote where the gradient pattern begins and stops.

To create the second shape (yellow rectangle) you will set the start X and Y to `(50, 50)`, end X and Y to (50, 75), proportional to false, cycle method to no cycle (`CycleMethod.NO_CYCLE`), and two color stop values to yellow (`Color.YELLOW`) and black (`Color.BLACK`) with an alpha transparency of .784.These settings give a linear gradient to our rectangle from top to bottom with a starting point of (50, 50) (top left of rectangle) that interpolates to the color black (bottom left of rectangle).

Finally, you'll notice a rounded rectangle with a repeating pattern of a gradient using green and black in a diagonal direction. This is a simple linear gradient paint that is the same as the linear gradient paint (`LinearGradient`) except that the start X, Y and the end X, Y are set in a diagonal position, and the cycle method is set to reflect (`CycleMethod.REFLECT`). When specifying the cycle method to reflect (`CycleMethod.REFLECT`), the gradient pattern will repeat or cycle between the colors. The following code snippet implements the rounded rectangle having a cycle method of reflect (`CycleMethod.REFLECT`):

```
LinearGradient cycleGrad = LinearGradientBuilder.create()
        .startX(50)
        .startY(50)
        .endX(70)
        .endY(70)
        .proportional(false)
        .cycleMethod(CycleMethod.REFLECT)
        .stops(new Stop(0f, Color.rgb(0, 255, 0, .784)),
               new Stop(1.0f, Color.rgb(0, 0, 0, .784)))
        .build();
```

1-7. Creating Menus

Problem

You want to create standard menus in your JavaFX applications.

Solution

Employ JavaFX's menu controls to provide standardized menuing capabilities such as check box menus, radio menus, submenus, and separators. The following are the main classes used to create menus.

- `javafx.scene.control.MenuBar`
- `javafx.scene.control.Menu`
- `javafx.scene.control.MenuItem`

The following code calls into play all the menuing capabilities listed previously. The example code will simulate a building security application containing menu options to turn on cameras, sound an alarm, and select contingency plans.

```
primaryStage.setTitle("Chapter 1-7 Creating Menus");
Group root = new Group();
```

```
Scene scene = new Scene(root, 300, 250, Color.WHITE);

MenuBar menuBar = new MenuBar();

// File menu - new, save, exit
Menu menu = new Menu("File");
menu.getItems().add(new MenuItem("New"));
menu.getItems().add(new MenuItem("Save"));
menu.getItems().add(new SeparatorMenuItem());
menu.getItems().add(new MenuItem("Exit"));

menuBar.getMenus().add(menu);

// Cameras menu - camera 1, camera 2
Menu tools = new Menu("Cameras");
tools.getItems().add(CheckMenuItemBuilder.create()
        .text("Show Camera 1")
        .selected(true)
        .build());

tools.getItems().add(CheckMenuItemBuilder.create()
        .text("Show Camera 2")
        .selected(true)
        .build());
menuBar.getMenus().add(tools);


// Alarm
Menu alarm = new Menu("Alarm");
ToggleGroup tGroup = new ToggleGroup();
RadioMenuItem soundAlarmItem = RadioMenuItemBuilder.create()
        .toggleGroup(tGroup)
        .text("Sound Alarm")
        .build();
RadioMenuItem stopAlarmItem = RadioMenuItemBuilder.create()
        .toggleGroup(tGroup)
        .text("Alarm Off")
        .selected(true)
        .build();

alarm.getItems().add(soundAlarmItem);
alarm.getItems().add(stopAlarmItem);

Menu contingencyPlans = new Menu("Contingent Plans");
contingencyPlans.getItems().add(new CheckMenuItem("Self Destruct in T minus 50"));
contingencyPlans.getItems().add(new CheckMenuItem("Turn off the coffee machine "));
contingencyPlans.getItems().add(new CheckMenuItem("Run for your lives! "));

alarm.getItems().add(contingencyPlans);
menuBar.getMenus().add(alarm);
```

```
menuBar.prefWidthProperty().bind(primaryStage.widthProperty());

root.getChildren().add(menuBar);
primaryStage.setScene(scene);
primaryStage.show();
```

Figure 1-14 shows a simulated building security application containing radio, checked, and submenu items.

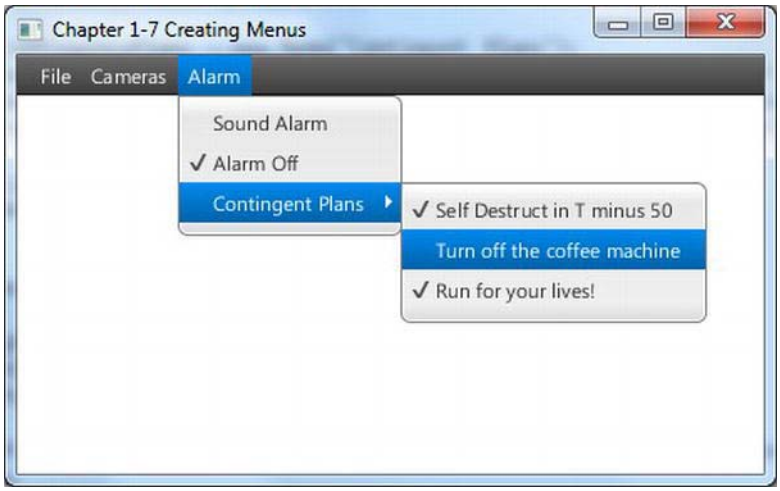

Figure 1-14. *Creating menus*

How It Works

Menus are standard ways on windowed platform applications to allow users to select options. Menus should also have the functionality of hot keys or keyboard equivalents. Often users will want to use the keyboard instead of the mouse to navigate the menu.

First, we create an instance of a `MenuBar` that will contain one to many menu (`MenuItem`) objects. The following code snippet creates a menu bar:

```
MenuBar menuBar = new MenuBar();
```

Secondly, we create menu (`Menu`) objects that contain one-to-many menu item (`MenuItem`) objects and other `Menu` objects making submenus. The following code snippet creates a menu:

```
Menu menu = new Menu("File");
```

Third, we create menu items to be added to `Menu` objects, such as menu (`MenuItem`), check (`CheckMenuItem`), and radio menu items (`RadioMenuItem`). Menu items can have icons in them. I don't showcase this in the recipe, but I encourage you to explore the various constructors for all menu items (`MenuItem`). When creating a radio menu item (`RadioMenuItem`), you should be aware of the `ToggleGroup`

class. The `ToggleGroup` class is also used on regular radio buttons (`RadioButtons`) to allow one selected option only. The following code creates radio menu items (`RadioMenuItems`) to be added to a `Menu` object:

```
// Alarm
Menu alarm = new Menu("Alarm");
ToggleGroup tGroup = new ToggleGroup();
RadioMenuItem soundAlarmItem = RadioMenuItemBuilder.create()
        .toggleGroup(tGroup)
        .text("Sound Alarm")
        .build();
RadioMenuItem stopAlarmItem = RadioMenuItemBuilder.create()
        .toggleGroup(tGroup)
        .text("Alarm Off")
        .selected(true)
        .build();

alarm.getItems().add(soundAlarmItem);
alarm.getItems().add(stopAlarmItem);
```

At times you may want some menu items separated with a visual line separator. To create a visual separator, create an instance of a `SeparatorMenuItem` class to be added to a menu via the `getItems()` method. The method `getItems()` returns an observable list of `MenuItem` objects (`ObservableList<MenuItem>`). As you will see later in recipe 1-11, you will learn about the ability to be notified when items in a collection are altered. The following code line adds a visual line separator (`SeparatorMenuItem`) to the menu:

```
menu.getItems().add(new SeparatorMenuItem());
```

Other menu items used are the check menu item (`CheckMenuItem`) and the radio menu item (`RadioMenuItem`), which are similar to their counterparts in JavaFX UI controls check box (`CheckBox`) and radio button (`RadioButton`), respectively.

Prior to our adding the menu bar to the scene, you will notice the bound property between the preferred width of the menu bar and the width of the `Stage` object via the `bind()` method. When binding these properties you will see the menu bar's width stretch when the user resizes the screen. Later you will see how binding works in recipe 1-10, "Binding Expressions."

The following code snippet shows the binding between the menu bar's `width` property and the stage's `width` property.

```
menuBar.prefWidthProperty().bind(primaryStage.widthProperty());

root.getChildren().add(menuBar);
```

1-8. Adding Components to a Layout

Problem

You want to create a simple form application by adding UI components to a layout similar to a grid-like display.

Solution

Use JavaFX's `javafx.scene.layout.GridPane` class. This source code implements a simple UI form containing a first and last name field controls using the grid pane layout node (`javafx.scene.layout.GridPane`):

```
GridPane gridpane = new GridPane();
gridpane.setPadding(new Insets(5));
gridpane.setHgap(5);
gridpane.setVgap(5);

Label fNameLbl = new Label("First Name");
TextField fNameFld = new TextField();
Label lNameLbl = new Label("First Name");
TextField lNameFld = new TextField();
Button saveButt = new Button("Save");

// First name label
GridPane.setHalignment(fNameLbl, HPos.RIGHT);
gridpane.add(fNameLbl, 0, 0);


// Last name label
GridPane.setHalignment(lNameLbl, HPos.RIGHT);
gridpane.add(lNameLbl, 0, 1);

// First name field
GridPane.setHalignment(fNameFld, HPos.LEFT);
gridpane.add(fNameFld, 1, 0);

// Last name field
GridPane.setHalignment(lNameFld, HPos.LEFT);
gridpane.add(lNameFld, 1, 1);

// Save button
GridPane.setHalignment(saveButt, HPos.RIGHT);
gridpane.add(saveButt, 1, 2);

root.getChildren().add(gridpane);
```

Figure 1-15 depicts a small form containing UI controls laid out using a grid pane layout node.

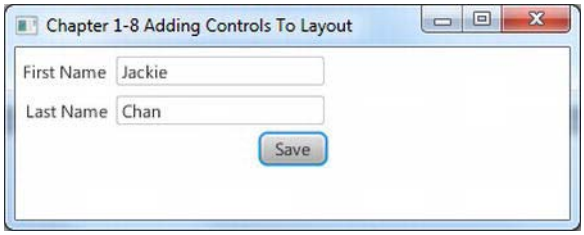

Figure 1-15. *Adding controls to a layout*

How It Works

One of the greatest challenges in building user interfaces is the laying out of controls onto the display area. When developing GUI applications it is ideal for an application to allow the user to move and adjust the size of their viewable area while maintaining a pleasant user experience. Similar to Java Swing, JavaFX layout has stock layouts that provide the most common ways to display UI controls on the scene graph. This recipe demonstrates the `GridPane` class. Before we begin I want explain two common layouts provided by JavaFX 2.0. These are the horizontal box (`HBox`) and the vertical box (`VBox`) layout nodes. These two common layouts will be used in later recipes to allow the scene graph to manage child nodes. HBox will contain child nodes that take the available horizontal space as nodes are added. VBox will contain child nodes that take the available vertical space as nodes are added.

First we create an instance of a `GridPane`. Next, we set the padding by using an instance of an `Inset` object. After setting the padding, we simply set the horizontal and vertical gap. The following code snippet instantiates a grid pane (`GridPane`) with padding, horizontal, and vertical gaps set to 5 (pixels):

```
GridPane gridpane = new GridPane();
gridpane.setPadding(new Insets(5));
gridpane.setHgap(5);
gridpane.setVgap(5);
```

The padding is the top, right, bottom, and left spacing around the region's content in pixels. When obtaining the preferred size, the padding will be included in the calculation. Setting the horizontal and vertical gaps relate to the spacing between UI controls within the cells.

Next is simply putting each UI control into its respective cell location. All cells are zero relative. Following is a code snippet that adds a save button UI control into a grid pane layout node (`GridPane`) at cell (1, 2):

```
gridpane.add(saveButt, 1, 2);
```

The layout also allows you to horizontally or vertically align controls within the cell. The following code statement right-aligns the save button:

```
GridPane.setHalignment(saveButt, HPos.RIGHT);
```

1-9. Generating Borders

Problem

You want to create and customize borders around an image.

Solution

Create an application to dynamically customized border regions using JavaFX's CSS styling API.

The following code creates an application that has a CSS editor text area and a border view region surrounding an image. By default the editor's text area will contain JavaFX styling selectors that create a dashed-blue line surrounding the image. You will have the opportunity to modify styling selector values in the CSS Editor by clicking the Bling! button to apply border settings.

```
primaryStage.setTitle("Chapter 1-9 Generating Borders");
Group root = new Group();
Scene scene = new Scene(root, 600, 330, Color.WHITE);

// create a grid pane
GridPane gridpane = new GridPane();
gridpane.setPadding(new Insets(5));
gridpane.setHgap(10);
gridpane.setVgap(10);

// label CSS Editor
Label cssEditorLbl = new Label("CSS Editor");
GridPane.setHalignment(cssEditorLbl, HPos.CENTER);
gridpane.add(cssEditorLbl, 0, 0);

// label Border View
Label borderLbl = new Label("Border View");
GridPane.setHalignment(borderLbl, HPos.CENTER);
gridpane.add(borderLbl, 1, 0);

// Text area for CSS editor
final TextArea cssEditorFld = new TextArea();
cssEditorFld.setPrefRowCount(10);
cssEditorFld.setPrefColumnCount(100);
cssEditorFld.setWrapText(true);
cssEditorFld.setPrefWidth(150);
GridPane.setHalignment(cssEditorFld, HPos.CENTER);
gridpane.add(cssEditorFld, 0, 1);

String cssDefault = "-fx-border-color: blue;\n"
        + "-fx-border-insets: 5;\n"
        + "-fx-border-width: 3;\n"
        + "-fx-border-style: dashed;\n";
```

```
cssEditorFld.setText(cssDefault);

// Border decorate the picture
final ImageView imv = new ImageView();
final Image image2 = new
Image(GeneratingBorders.class.getResourceAsStream("smoke_glass_buttons1.png"));
imv.setImage(image2);


final HBox pictureRegion = new HBox();
pictureRegion.setStyle(cssDefault);
pictureRegion.getChildren().add(imv);
gridpane.add(pictureRegion, 1, 1);


Button apply = new Button("Bling!");
GridPane.setHalignment(apply, HPos.RIGHT);
gridpane.add(apply, 0, 2);

apply.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        pictureRegion.setStyle(cssEditorFld.getText());
    }
});

root.getChildren().add(gridpane);
primaryStage.setScene(scene);
primaryStage.show();
```

Figure 1-16 illustrates the border customizer application.

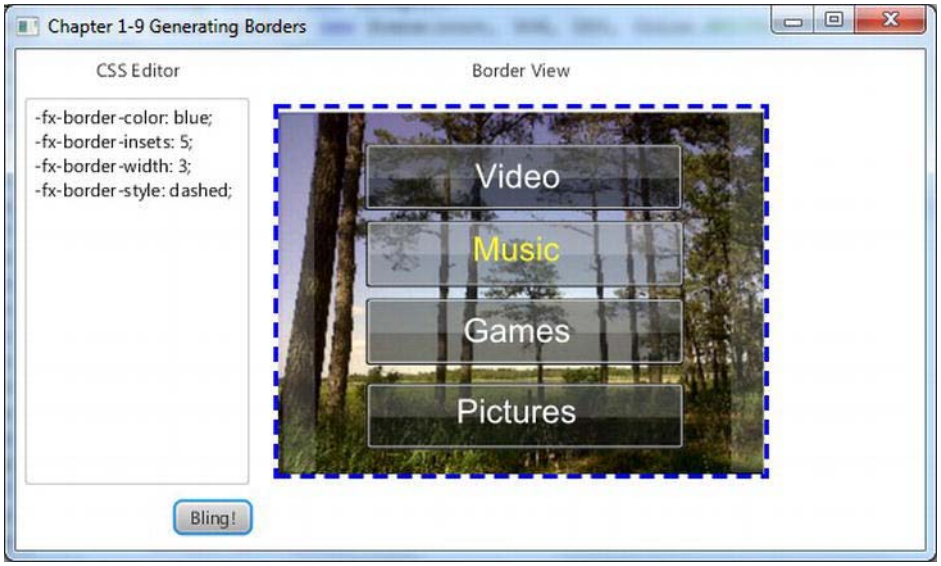

Figure 1-16. *Generating borders*

How It Works

JavaFX is capable of styling JavaFX nodes similar to Cascading Style Sheets (CSS) in the world of web development. This powerful API can alter a node's background color, font, border, and many other attributes essentially allowing a developer or designer to skin GUI controls using CSS.

This recipe allows a user to enter JavaFX CSS styles in the left text area and, by clicking the Bling! button below, to apply the style around the image shown to the right. Based on the type of node there are limitations to what styles you can set. To see a full listing of all style selectors refer to the JavaFX CSS Reference Guide:
`http://download.oracle.com/docs/cd/E17802_01/javafx/javafx/1.3/docs/api/javafx.scene/doc-files/cssref.html`.

In the first step of applying JavaFX CSS styles, you must determine what type of node you want to style. When setting attributes on various node types, you will discover that certain nodes have limitations. In our recipe the intent is to put a border around the `ImageView` object. Because `ImageView` is not extending from `Region` it doesn't have border style properties. So, to resolve this I simply create an `HBox` layout to contain the `imageView` and apply the JavaFX CSS against the `HBox`. Shown here is code to apply JavaFX CSS border styles to a horizontal box region (`HBox`) using the `setStyle()` method:

```
String cssDefault = "-fx-border-color: blue;\n"
    + "-fx-border-insets: 5;\n"
    + "-fx-border-width: 3;\n"
    + "-fx-border-style: dashed;\n";
```

```
final ImageView imv = new ImageView();

...// additional image view properties set

final HBox pictureRegion = new HBox();
pictureRegion.setStyle(cssDefault);
pictureRegion.getChildren().add(imv);
```

1-10. Binding Expressions

Problem

You want to synchronize changes between two values.

Solution

Use `javafx.beans.binding.*` and `javafx.beans.property.*` packages to bind variables. There is more than one scenario to consider when binding values or properties. This recipe demonstrates the following three binding strategies:

- Bidirectional binding on a Java Bean
- High-level binding using the Fluent API
- Low-level binding using `javafx.beans.binding.*` Binding objects

The following code is a console application implementing these three strategies. The console application will output property values based on various binding scenarios. The first scenario is a bidirectional binding between a String property variable and a String property owned by a domain object (`Contact`) such as the `firstName` property. The next scenario is a high-level binding using a fluent interface API to calculate the area of rectangle. The last scenario is using a low-level binding strategy to calculate the volume of a sphere. The difference between the high-and low-level binding is that the high level uses methods such as `multiply()`, `subtract()` instead of the operators * and -. When using low-level binding, you would use a derived `NumberBinding` class such as a `DoubleBinding` class. With a `DoubleBinding` class you will override its `computeValue()` method so that you can use the familiar operators such as * and - to formulate complex math equations:

```
package javafx2introbyexample.chapter1.recipe1_10;

import javafx.beans.binding.DoubleBinding;
import javafx.beans.binding.NumberBinding;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

/**
 * Binding Expressions
```

```
 * @author cdea
 */
public class BindingExpressions {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Chapter 1-10 Binding Expressions\n");

        System.out.println("Binding a Contact bean [Bi-directional binding]");
        Contact contact = new Contact("John", "Doe");
        StringProperty fname = new SimpleStringProperty();
        fname.bindBidirectional(contact.firstNameProperty());
        StringProperty lname = new SimpleStringProperty();
        lname.bindBidirectional(contact.lastNameProperty());

        System.out.println("Current - StringProperty values   : " + fname.getValue() + " " +
lname.getValue());
        System.out.println("Current - Contact values          : " + contact.getFirstName() + "
" + contact.getLastName());

        System.out.println("Modifying StringProperty values");
        fname.setValue("Jane");
        lname.setValue("Deer");

        System.out.println("After - StringProperty values   : " + fname.getValue() + " " +
lname.getValue());
        System.out.println("After - Contact values          : " + contact.getFirstName() + " "
+ contact.getLastName());


        System.out.println();
        System.out.println("A Area of a Rectangle [High level Fluent API]");

        // Area = width * height
        final IntegerProperty width = new SimpleIntegerProperty(10);
        final IntegerProperty height = new SimpleIntegerProperty(10);

        NumberBinding area = width.multiply(height);

        System.out.println("Current - Width and Height     : " + width.get() + " " +
height.get());
        System.out.println("Current - Area of the Rectangle: " + area.getValue());
        System.out.println("Modifying width and height");

        width.set(100);
        height.set(700);

        System.out.println("After - Width and Height     : " + width.get() + " " +
height.get());
        System.out.println("After - Area of the Rectangle: " + area.getValue());
```

```
        System.out.println();
        System.out.println("A Volume of a Sphere [low level API]");

        // volume = 4/3 * pi r^3
        final DoubleProperty radius = new SimpleDoubleProperty(2);

        DoubleBinding volumeOfSphere = new DoubleBinding() {
            {
                super.bind(radius);
            }

            @Override
            protected double computeValue() {
                return (4 / 3 * Math.PI * Math.pow(radius.get(), 3));
            }
        };

        System.out.println("Current - radius for Sphere: " + radius.get());
        System.out.println("Current - volume for Sphere: " + volumeOfSphere.get());
        System.out.println("Modifying DoubleProperty radius");

        radius.set(50);
        System.out.println("After - radius for Sphere: " + radius.get());
        System.out.println("After - volume for Sphere: " + volumeOfSphere.get());



    }
}

class Contact {

    private SimpleStringProperty firstName = new SimpleStringProperty();
    private SimpleStringProperty lastName = new SimpleStringProperty();

    public Contact(String fn, String ln) {
        firstName.setValue(fn);
        lastName.setValue(ln);
    }

    public final String getFirstName() {
        return firstName.getValue();
    }

    public StringProperty firstNameProperty() {
        return firstName;
    }

    public final void setFirstName(String firstName) {
        this.firstName.setValue(firstName);
    }
```

```
    public final String getLastName() {
        return lastName.getValue();
    }

    public StringProperty lastNameProperty() {
        return lastName;
    }

    public final void setLastName(String lastName) {
        this.lastName.setValue(lastName);
    }
}
```

The following output demonstrates the three binding scenarios:

```
Binding a Contact bean [Bi-directional binding]
Current - StringProperty values   : John Doe
Current - Contact values          : John Doe
Modifying StringProperty values
After - StringProperty values   : Jane Deer
After - Contact values          : Jane Deer

A Area of a Rectangle [High level Fluent API]
Current - Width and Height     : 10 10
Current - Area of the Rectangle: 100
Modifying width and height
After - Width and Height     : 100 700
After - Area of the Rectangle: 70000

A Volume of a Sphere [low level API]
Current - radius for Sphere: 2.0
Current - volume for Sphere: 25.132741228718345
Modifying DoubleProperty radius
After - radius for Sphere: 50.0
After - volume for Sphere: 392699.0816987241
```

How It Works

Binding has the idea of at least two values being synchronized. This means when a dependent variable changes the other variable changes. JavaFX provides many binding options that enable the developer to synchronize properties in domain objects and GUI controls. This recipe will demonstrate the three common binding scenarios.

One of the easiest ways to bind variables is a *bidirectional bind*. This scenario is often used when domain objects contain data that will be bound to a GUI form. In our recipe we create a simple contact (`Contact`) object containing a first name and last name. Notice the instance variables using the `SimpleStringProperty` class. Many of these classes, which end in `Property`, are `javafx.beans.Observable` classes that all have the ability to be bound. In order for these properties to be bound, they must be the same data type. In the preceding example we create first name and last name variables of type `SimpleStringProperty` outside the created `Contact` domain object. Once they have been created we bind them bidirectionally to allow changes to update on either end. So if you change the domain object, the other bound properties get updated. And when the outside variables are modified, the domain object's

properties get updated. The following demonstrates bidirectional binding against string properties on a domain object (`Contact`):

```
Contact contact = new Contact("John", "Doe");
StringProperty fname = new SimpleStringProperty();
fname.bindBidirectional(contact.firstNameProperty());
StringProperty lname = new SimpleStringProperty();
lname.bindBidirectional(contact.lastNameProperty());
```

Next up is how to bind numbers. Binding numbers is simple when using the new Fluent API. This high-level mechanism allows a developer to bind variables to compute values using simple arithmetic. Basically, a formula is "bound" to change its result based on changes to the variables it is bound to. Please look at the Javadoc for details on all the available methods and number types. In this example we simply create a formula for an area of a rectangle. The area (`NumberBinding`) is the binding, and its dependencies are the width and height (`IntegerProperty`) properties. When binding using the fluent interface API, you'll notice the `multiply()` method. According to the Javadoc, all property classes inherit from the `NumberExpressionBase` class, which contains the number-based fluent interface APIs. The following code snippet uses the fluent interface API:

```
// Area = width * height
final IntegerProperty width = new SimpleIntegerProperty(10);
final IntegerProperty height = new SimpleIntegerProperty(10);
NumberBinding area = width.multiply(height);
```

The last scenario on binding numbers is considered more of a low-level approach. This allows the developer to use primitives and more-complex math operations. Here we use a `DoubleBinding` class to solve the volume of a sphere given the radius. We begin by implementing the `computeValue()` method to perform our calculation of the volume. Shown is the low-level binding scenario to compute the volume of a sphere by overriding the `computeValue()` method:

```
final DoubleProperty radius = new SimpleDoubleProperty(2);

DoubleBinding volumeOfSphere = new DoubleBinding() {
        {
            super.bind(radius);
        }

        @Override
        protected double computeValue() {
                return (4 / 3 * Math.PI * Math.pow(radius.get(), 3));
        }
};
```

1-11. Creating and Working with Observable Lists

Problem

You want to create a GUI application containing two list view controls allowing the user pass items between the two lists.

Solution

Take advantage of JavaFX's `javafx.collections.ObservableList` and `javafx.scene.control.ListView` classes to provide a model-view-controller (MVC) mechanism that updates the UI's list view control whenever the back-end list is manipulated.

The following code creates a GUI application containing two lists that allow the user to send items contained in one list to be sent to the other. Here you will create a contrived application to pick candidates to be considered heroes. The user will pick potential candidates from the list on the left to be moved into the list on the right to be considered heroes. This demonstrates UI list controls' (`ListView`) ability to be synchronized with back-end store lists (`ObservableList`).

```
primaryStage.setTitle("Chapter 1-11 Creating and Working with ObservableLists");
Group root = new Group();
Scene scene = new Scene(root, 400, 250, Color.WHITE);

// create a grid pane
GridPane gridpane = new GridPane();
gridpane.setPadding(new Insets(5));
gridpane.setHgap(10);
gridpane.setVgap(10);

// candidates label
Label candidatesLbl = new Label("Candidates");
GridPane.setHalignment(candidatesLbl, HPos.CENTER);
gridpane.add(candidatesLbl, 0, 0);

Label heroesLbl = new Label("Heroes");
gridpane.add(heroesLbl, 2, 0);
GridPane.setHalignment(heroesLbl, HPos.CENTER);

// candidates
final ObservableList<String> candidates =
FXCollections.observableArrayList("Superman",
        "Spiderman",
        "Wolverine",
        "Police",
        "Fire Rescue",
        "Soldiers",
        "Dad & Mom",
        "Doctor",
        "Politician",
        "Pastor",
        "Teacher");
final ListView<String> candidatesListView = new ListView<String>(candidates);
candidatesListView.setPrefWidth(150);
candidatesListView.setPrefHeight(150);

gridpane.add(candidatesListView, 0, 1);

// heros
final ObservableList<String> heroes = FXCollections.observableArrayList();
```

```
final ListView<String> heroListView = new ListView<String>(heroes);
heroListView.setPrefWidth(150);
heroListView.setPrefHeight(150);

gridpane.add(heroListView, 2, 1);


// select heroes
Button sendRightButton = new Button(">");
sendRightButton.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent event) {
        String potential = candidatesListView.getSelectionModel().getSelectedItem();
        if (potential != null) {
            candidatesListView.getSelectionModel().clearSelection();
            candidates.remove(potential);
            heroes.add(potential);
        }
    }
});

// deselect heroes
Button sendLeftButton = new Button("<");
sendLeftButton.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent event) {
        String notHero = heroListView.getSelectionModel().getSelectedItem();
        if (notHero != null) {
            heroListView.getSelectionModel().clearSelection();
            heroes.remove(notHero);
            candidates.add(notHero);
        }
    }
});

VBox vbox = new VBox(5);
vbox.getChildren().addAll(sendRightButton,sendLeftButton);

gridpane.add(vbox, 1, 1);
GridPane.setConstraints(vbox, 1, 1, 1, 2,HPos.CENTER, VPos.CENTER);

root.getChildren().add(gridpane);
primaryStage.setScene(scene);
primaryStage.show();
```

Figure 1-17 depicts our hero selection application.

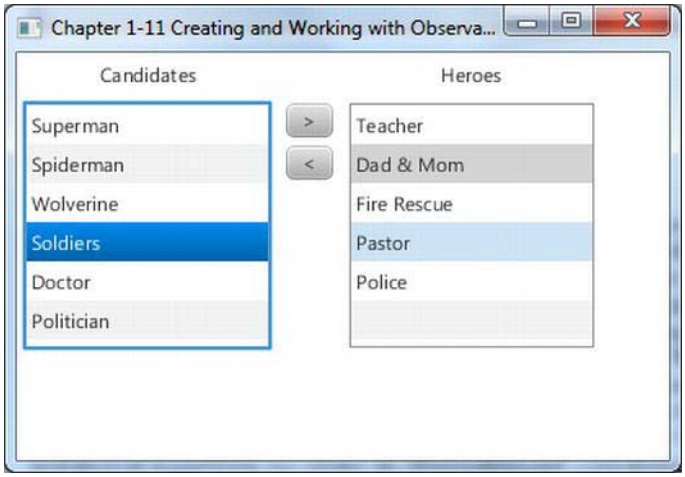

Figure 1-17. *ListViews and ObservableLists*

How It Works

When dealing with Java `collections` you'll notice there are so many useful container classes that represent all kinds of data structures. One commonly used collection is the `java.util.ArrayList` class. When building applications with domain objects that contain an `ArrayList,` a developer can easily manipulate objects inside the collection. But, in the past (back in the day), when using Java Swing components combined with collections can often be a challenge, especially updating the GUI to reflect changes in the domain object. How do we resolve this issue? Well, JavaFX's `ObservableList` to the rescue!

Speaking of rescue, I've created a GUI application to allow users to choose their favorite heroes. This is quite similar to application screens that manage user roles by adding or removing items from list box components. In JavaFX we will be using a `ListView` control to hold String objects. Before we create an instance of a `ListView` we create an `ObservableList` containing our candidates. Here you'll notice the use of a factory class called `FXCollections`, in which you can pass in common collection types to be wrapped and returned to the caller as an `ObservableList`. In the recipe I passed in an array of Strings instead of an `ArrayList`, so hopefully you get the idea about how to use the `FXCollections` class. I trust you will use it wisely: "With great power, there must also come great responsibility". This code line calls the `FXCollections` class to return an observable list (`ObservableList`):

```
ObservableList<String> candidates = FXCollections.observableArrayList(...);
```

After creating an `ObservableList`, a `ListView` class is instantiated using a constructor that receives the observable list. Shown here is code to create and populate a `ListView` object:

```
ListView<String> candidatesListView = new ListView<String>(candidates);
```

In the last item of business, our code will manipulate the `ObservableLists` as if they were `java.util.ArrayLists`. Once manipulated, the `ListView` will be notified and automatically updated to reflect the changes of the `ObservableList`. The following code snippet implements the event handler and action event when the user presses the send right button:

```
// select heroes
Button sendRightButton = new Button(">");
sendRightButton.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent event) {
        String potential = candidatesListView.getSelectionModel().getSelectedItem();
        if (potential != null) {
            candidatesListView.getSelectionModel().clearSelection();
            candidates.remove(potential);
            heroes.add(potential);
        }
    }
});
```

When setting an action we use the generic class `EventHandler` to create an anonymous inner class with the `handle()` method to listen for a button press event. When a button press event arrives, the code will determine which item in the `ListView` was selected. Once the item was determined, we clear the selection, remove the item, and add the item to the hero's `ObserverableList`.

1-12. Generating a Background Process

Problem

You want to create a GUI application that simulates the copying of files using background processing while displaying the progress to the user.

Solution

Create an application typical of a dialog box showing progress indicators while copying files in the background. The following are the main classes used in this recipe:

- `javafx.scene.control.ProgressBar`
- `javafx.scene.control.ProgressIndicator`
- `javafx.concurrent.Task` classes

The following source code is an application that simulates a file copy dialog box displaying progress indicators and performing background processes:

```
package javafx2introbyexample.chapter1.recipe1_12;

import java.util.Random;
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.concurrent.Task;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
```

```
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.ProgressIndicator;
import javafx.scene.control.TextArea;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

/**
 * Background Processes
 * @author cdea
 */
public class BackgroundProcesses extends Application {

    static Task copyWorker;
    final int numFiles = 30;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chapter 1-12 Background Processes");
        Group root = new Group();
        Scene scene = new Scene(root, 330, 120, Color.WHITE);

        BorderPane mainPane = new BorderPane();

mainPane.layoutXProperty().bind(scene.widthProperty().subtract(mainPane.widthProperty()).divid
e(2));
        root.getChildren().add(mainPane);

        final Label label = new Label("Files Transfer:");
        final ProgressBar progressBar = new ProgressBar(0);
        final ProgressIndicator progressIndicator = new ProgressIndicator(0);

        final HBox hb = new HBox();
        hb.setSpacing(5);
        hb.setAlignment(Pos.CENTER);
        hb.getChildren().addAll(label, progressBar, progressIndicator);
        mainPane.setTop(hb);


        final Button startButton = new Button("Start");
```

```
        final Button cancelButton = new Button("Cancel");
        final TextArea textArea = new TextArea();
        textArea.setEditable(false);
        textArea.setPrefSize(200, 70);
        final HBox hb2 = new HBox();
        hb2.setSpacing(5);
        hb2.setAlignment(Pos.CENTER);
        hb2.getChildren().addAll(startButton, cancelButton, textArea);
        mainPane.setBottom(hb2);

        // wire up start button
        startButton.setOnAction(new EventHandler<ActionEvent>() {

            public void handle(ActionEvent event) {
                startButton.setDisable(true);
                progressBar.setProgress(0);
                progressIndicator.setProgress(0);
                textArea.setText("");
                cancelButton.setDisable(false);
                copyWorker = createWorker(numFiles);

                // wire up progress bar
                progressBar.progressProperty().unbind();
                progressBar.progressProperty().bind(copyWorker.progressProperty());
                progressIndicator.progressProperty().unbind();
                progressIndicator.progressProperty().bind(copyWorker.progressProperty());

                // append to text area box
                copyWorker.messageProperty().addListener(new ChangeListener<String>() {

                    public void changed(ObservableValue<? extends String> observable, String
oldValue, String newValue) {
                        textArea.appendText(newValue + "\n");
                    }
                });

                new Thread(copyWorker).start();
            }
        });

        // cancel button will kill worker and reset.
        cancelButton.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent event) {
                startButton.setDisable(false);
                cancelButton.setDisable(true);
                copyWorker.cancel(true);

                // reset
                progressBar.progressProperty().unbind();
                progressBar.setProgress(0);
                progressIndicator.progressProperty().unbind();
                progressIndicator.setProgress(0);
```

```
                textArea.appendText("File transfer was cancelled.");
            }
        });


        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public Task createWorker(final int numFiles) {
        return new Task() {

            @Override
            protected Object call() throws Exception {
                for (int i = 0; i < numFiles; i++) {
                    long elapsedTime = System.currentTimeMillis();
                    copyFile("some file", "some dest file");
                    elapsedTime = System.currentTimeMillis() - elapsedTime;
                    String status = elapsedTime + " milliseconds";

                    // queue up status
                    updateMessage(status);
                    updateProgress(i + 1, numFiles);
                }
                return true;
            }
        };
    }

    public void copyFile(String src, String dest) throws InterruptedException {
        // simulate a long time
        Random rnd = new Random(System.currentTimeMillis());
        long millis = rnd.nextInt(1000);
        Thread.sleep(millis);
    }
}
```

Figure 1-18 shows our Background Processes application simulating a file copy window.

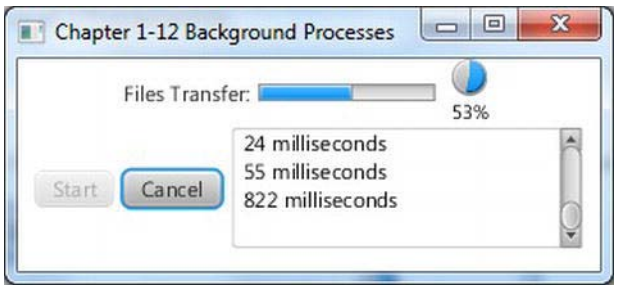

Figure 1-18. *Background processes*

How It Works

One of the main pitfalls of GUI development is knowing when and how to delegate work (`Threads`). We are constantly reminded of thread safety, especially when it comes to blocking the GUI thread.

We begin by creating not one but two progress controls to show off to the user the work being done. One is a progress bar, and the other is a progress indicator. The progress indicator shows a percentage below the indicator icon. The following code snippet shows the initial creation of progress controls:

```
final ProgressBar progressBar = new ProgressBar(0);
final ProgressIndicator progressIndicator = new ProgressIndicator(0);
```

Next, we create a worker thread via our `createWorker()` method. The `createWorker()` convenience method will instantiate and return a `javafx.concurrent.Task` object, which is similar to the Java Swing's `SwingWorker` class. Unlike the `SwingWorker` class, the `Task` object is greatly simplified and easier to use. If you've compared the last recipe you will notice that none of the GUI controls is passed into the `Task`. The clever JavaFX team has created observable properties that allow us to bind to. This fosters a more event-driven approach to handling work (tasks). When creating an instance of a `Task` object you will implement the `call()` method to do work in the background. During the work being done, you may want to queue up intermediate results such as progress or text info, you can call the `updateProgress()` and `updateMessage()` methods. These methods will update information in a threadsafe way so that the observer of the progress properties will be able to update the GUI safely without blocking the GUI thread. The following code snippet demonstrates the ability to queue up messages and progress:

```
// queue up status
updateMessage(status);
updateProgress(i + 1, numFiles);
```

After creating a worker `Task` we unbind any old tasks bound to the progress controls. Once the progress controls are unbound, we then bind the progress controls to our newly created `Task` object `copyWorker`. Shown here is the code used to rebind a new `Task` object to the progress UI controls:

```
// wire up progress bar
progressBar.progressProperty().unbind();
progressBar.progressProperty().bind(copyWorker.progressProperty());
progressIndicator.progressProperty().unbind();
progressIndicator.progressProperty().bind(copyWorker.progressProperty());
```

Next, we implement a `ChangeListener` to append the queued results into the `TextArea` control. Another remarkable thing about JavaFX Properties is that you can attach many listeners similar to Java Swing components. Finally our worker and controls are all wired up to spawn a thread to go off in the background. The following code line shows the launching of a `Task` worker object:

```
new Thread(copyWorker).start();
```

Finally, we discuss the cancel button. The cancel button will simply call the `Task` object's `cancel()` method to kill the process. Once the task is cancelled the progress controls are reset. Once a worker `Task` is cancelled it cannot be reused. That is why the start button re-creates a new `Task`. If you want a more-robust solution, you should look at the `javafx.concurrent.Service` class. The following code line will cancel a `Task` worker object:

```
copyWorker.cancel(true);
```

1-13. Associating Keyboard Sequences to Applications

Problem

You want to create keyboard shortcuts for menu options.

Solution

Create an application that will use JavaFX's key combination APIs. The main classes you will be using are shown here:

- `javafx.scene.input.KeyCode`
- `javafx.scene.input.KeyCodeCombination`
- `javafx.scene.input.KeyCombination`

The following source code listing is an application that displays the available keyboard shortcuts that are bound to menu items. When the user performs a keyboard shortcut the application will display the key combination on the screen:

```
primaryStage.setTitle("Chapter 1-13 Associating Keyboard Sequences");
Group root = new Group();
Scene scene = new Scene(root, 530, 300, Color.WHITE);

final StringProperty statusProperty = new SimpleStringProperty();

InnerShadow iShadow = InnerShadowBuilder.create()
        .offsetX(3.5f)
        .offsetY(3.5f)
        .build();
final Text status = TextBuilder.create()
    .effect(iShadow)
    .x(100)
    .y(50)
    .fill(Color.LIME)
    .font(Font.font(null, FontWeight.BOLD, 35))
    .translateY(50)
    .build();
status.textProperty().bind(statusProperty);
statusProperty.set("Keyboard Shortcuts \nCtrl-N, \nCtrl-S, \nCtrl-X");
root.getChildren().add(status);

MenuBar menuBar = new MenuBar();
menuBar.prefWidthProperty().bind(primaryStage.widthProperty());
root.getChildren().add(menuBar);

Menu menu = new Menu("File");
menuBar.getMenus().add(menu);

MenuItem newItem = MenuItemBuilder.create()
```

```
            .text("New")
            .accelerator(new KeyCodeCombination(KeyCode.N, KeyCombination.CONTROL_DOWN))
            .onAction(new EventHandler<ActionEvent>() {
                    public void handle(ActionEvent event) {
                        statusProperty.set("Ctrl-N");
                    }
                })
            .build();
    menu.getItems().add(newItem);


    MenuItem saveItem = MenuItemBuilder.create()
            .text("Save")
            .accelerator(new KeyCodeCombination(KeyCode.S, KeyCombination.CONTROL_DOWN))
            .onAction(new EventHandler<ActionEvent>() {
                    public void handle(ActionEvent event) {
                        statusProperty.set("Ctrl-S");
                    }
                })
            .build();
    menu.getItems().add(saveItem);

    menu.getItems().add(new SeparatorMenuItem());

    MenuItem exitItem = MenuItemBuilder.create()
            .text("Exit")
            .accelerator(new KeyCodeCombination(KeyCode.X, KeyCombination.CONTROL_DOWN))
            .onAction(new EventHandler<ActionEvent>() {
                    public void handle(ActionEvent event) {
                        statusProperty.set("Ctrl-X");
                    }
                })
            .build();
    menu.getItems().add(exitItem);

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

Figure 1-19 displays an application that demonstrates keyboard sequences or keyboard shortcuts.

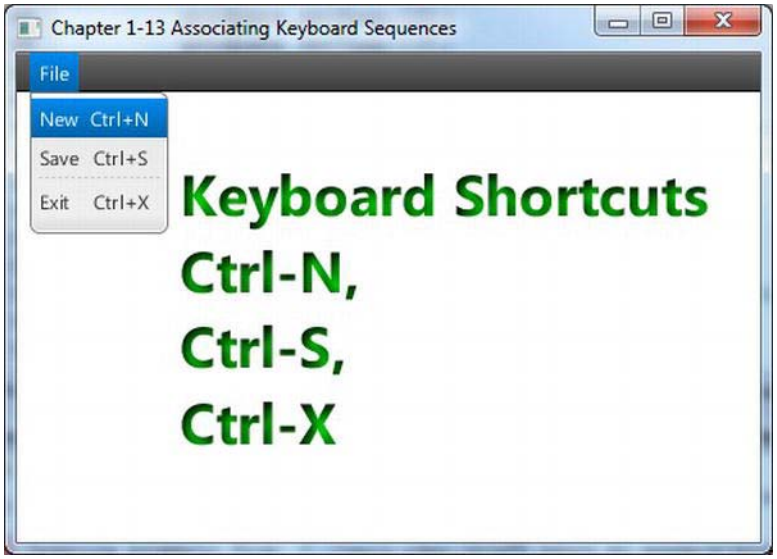

Figure 1-19. *Keyboard sequences*

How It Works

Seeing that the previous recipe was a tad boring, I decided to make things a little more interesting. We will be creating keyboard shortcuts using the new `javafx.scene.input.KeyCodeCombination` and `javafx.scene.input.KeyCombination` classes. This recipe will display `Text` nodes onto the scene graph when the user performs the key combinations. When displaying the `Text` nodes I applied an inner shadow effect. The following code snippet creates a `Text` node with an inner shadow effect:

```
InnerShadow iShadow = InnerShadowBuilder.create()
        .offsetX(3.5f)
        .offsetY(3.5f)
        .build();
final Text status = TextBuilder.create()
    .effect(iShadow)
    .x(100)
    .y(50)
    .fill(Color.LIME)
```

To create a keyboard shortcut you simply call a menu or button control's `setAccelerator()` method. In this recipe we use a `Builder` class and set the key combination using the `accelerator()` method. The following code line specifies the key combinations for a control-N:

```
MenuItem newItem = MenuItemBuilder.create()
    .text("New")
    .accelerator(new KeyCodeCombination(KeyCode.N, KeyCombination.CONTROL_DOWN))
```

1-14. Creating and Working with Tables

Problem

You want to display items in a UI table control similar to Java Swing's JTable component.

Solution

Create an application using JavaFX's `javafx.scene.control.TableView` class. The `TableView` control provides the equivalent functionality similar to Swing's `JTable` component.

To exercise the `TableView` control you will be creating an application that will display bosses and employees. On the left you will implement a `ListView` control containing bosses, and employees (subordinates) will be displayed in a `TableView` control on the right.

Shown here is the source code of a simple domain (`Person`) class to represent a boss or an employee to be displayed in a `ListView` or `TableView` control:

```
package javafx2introbyexample.chapter1.recipe1_14;

import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

/**
 *
 * @author cdea
 */
public class Person {

    private StringProperty aliasName;
    private StringProperty firstName;
    private StringProperty lastName;
    private ObservableList<Person> employees = FXCollections.observableArrayList();


    public final void setAliasName(String value) {
        aliasNameProperty().set(value);
    }

    public final String getAliasName() {
        return aliasNameProperty().get();
    }

    public StringProperty aliasNameProperty() {
        if (aliasName == null) {
            aliasName = new SimpleStringProperty();
        }
        return aliasName;
    }
```

```
    public final void setFirstName(String value) {
        firstNameProperty().set(value);
    }

    public final String getFirstName() {
        return firstNameProperty().get();
    }

    public StringProperty firstNameProperty() {
        if (firstName == null) {
            firstName = new SimpleStringProperty();
        }
        return firstName;
    }

    public final void setLastName(String value) {
        lastNameProperty().set(value);
    }

    public final String getLastName() {
        return lastNameProperty().get();
    }

    public StringProperty lastNameProperty() {
        if (lastName == null) {
            lastName = new SimpleStringProperty();
        }
        return lastName;
    }


    public ObservableList<Person> employeesProperty() {
        return employees;
    }

    public Person(String alias, String firstName, String lastName) {
        setAliasName(alias);
        setFirstName(firstName);
        setLastName(lastName);
    }

}
```

The following is our main application code that displays a list view component on the left containing bosses and a table view control on the right containing employees:

```
        primaryStage.setTitle("Chapter 1-14 Working with Tables");
        Group root = new Group();
        Scene scene = new Scene(root, 500, 250, Color.WHITE);

        // create a grid pane
```

```
GridPane gridpane = new GridPane();
gridpane.setPadding(new Insets(5));
gridpane.setHgap(10);
gridpane.setVgap(10);

// candidates label
Label candidatesLbl = new Label("Boss");
GridPane.setHalignment(candidatesLbl, HPos.CENTER);
gridpane.add(candidatesLbl, 0, 0);

// List of leaders
ObservableList<Person> leaders = getPeople();
final ListView<Person> leaderListView = new ListView<>(leaders);
leaderListView.setPrefWidth(150);
leaderListView.setPrefHeight(150);

// display first and last name with tooltip using alias
leaderListView.setCellFactory(new Callback<ListView<Person>, ListCell<Person>>() {

    public ListCell<Person> call(ListView<Person> param) {
        final Label leadLbl = new Label();
        final Tooltip tooltip = new Tooltip();
            final ListCell<Person> cell = new ListCell<Person>() {
                @Override
                public void updateItem(Person item, boolean empty) {
                        super.updateItem(item, empty);
                        if (item != null) {
                            leadLbl.setText(item.getAliasName());
                            setText(item.getFirstName() + " " + item.getLastName());
                            tooltip.setText(item.getAliasName());
                            setTooltip(tooltip);
                        }
                }
            }; // ListCell
            return cell;

    }
}); // setCellFactory

gridpane.add(leaderListView, 0, 1);

Label emplLbl = new Label("Employees");
gridpane.add(emplLbl, 2, 0);
GridPane.setHalignment(emplLbl, HPos.CENTER);

final TableView<Person> employeeTableView = new TableView<>();
employeeTableView.setPrefWidth(300);
final ObservableList<Person> teamMembers = FXCollections.observableArrayList();
employeeTableView.setItems(teamMembers);

TableColumn<Person, String> aliasNameCol = new TableColumn<>("Alias");
aliasNameCol.setEditable(true);
```

```
        aliasNameCol.setCellValueFactory(new PropertyValueFactory("aliasName"));
        aliasNameCol.setPrefWidth(employeeTableView.getPrefWidth() / 3);

        TableColumn<Person, String> firstNameCol = new TableColumn<>("First Name");
        firstNameCol.setCellValueFactory(new PropertyValueFactory("firstName"));
        firstNameCol.setPrefWidth(employeeTableView.getPrefWidth() / 3);

        TableColumn<Person, String> lastNameCol = new TableColumn<>("Last Name");
        lastNameCol.setCellValueFactory(new PropertyValueFactory("lastName"));
        lastNameCol.setPrefWidth(employeeTableView.getPrefWidth() / 3);

        employeeTableView.getColumns().setAll(aliasNameCol, firstNameCol, lastNameCol);
        gridpane.add(employeeTableView, 2, 1);

        // selection listening
        leaderListView.getSelectionModel().selectedItemProperty().addListener(new
ChangeListener<Person>() {
            public void changed(ObservableValue<? extends Person> observable, Person oldValue,
Person newValue) {
                if (observable != null && observable.getValue() != null) {
                    teamMembers.clear();
                    teamMembers.addAll(observable.getValue().employeesProperty());
                }
            }
        });

        root.getChildren().add(gridpane);

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

The following code is the `getPeople()` method contained in the `WorkingWithTables` main application class. This method helps to populate the UI `TableView` control shown previously:

```
    private ObservableList<Person> getPeople() {
        ObservableList<Person> people = FXCollections.<Person>observableArrayList();
        Person docX = new Person("Professor X", "Charles", "Xavier");
        docX.employeesProperty().add(new Person("Wolverine", "James", "Howlett"));
        docX.employeesProperty().add(new Person("Cyclops", "Scott", "Summers"));
        docX.employeesProperty().add(new Person("Storm", "Ororo", "Munroe"));

        Person magneto = new Person("Magneto", "Max", "Eisenhardt");
        magneto.employeesProperty().add(new Person("Juggernaut", "Cain", "Marko"));
        magneto.employeesProperty().add(new Person("Mystique", "Raven", "Darkhölme"));
        magneto.employeesProperty().add(new Person("Sabretooth", "Victor", "Creed"));

        Person biker = new Person("Mountain Biker", "Jonathan", "Gennick");
        biker.employeesProperty().add(new Person("Josh", "Joshua", "Juneau"));
        biker.employeesProperty().add(new Person("Freddy", "Freddy", "Guime"));
        biker.employeesProperty().add(new Person("Mark", "Mark", "Beaty"));
        biker.employeesProperty().add(new Person("John", "John", "O'Conner"));
```

```
        biker.employeesProperty().add(new Person("D-Man", "Carl", "Dea"));

        people.add(docX);
        people.add(magneto);
        people.add(biker);

        return people;
    }
```

Figure 1-20 displays our application that demonstrates JavaFX's `TableView` control.

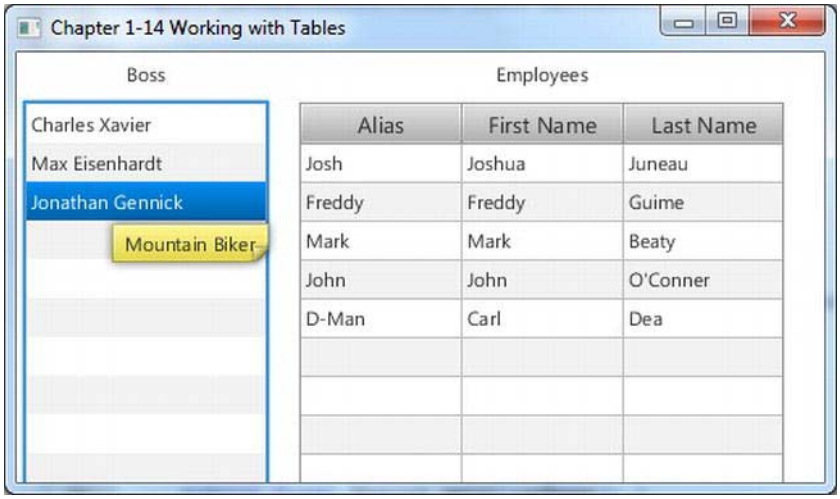

Figure 1-20. *Working with tables*

How It Works

Just for fun I created a simple GUI to display employees and their bosses. You notice in Figure 1-20 on the left is a list of people (Boss). When users click and select a boss, their employees will be shown to in the `TableView` area to the right. You'll also notice the tooltip when you hover over the selected boss.

Before we begin to discuss the `TableView` control I want to explain about the `ListView` that is responsible for updating the `TableView`. In model view fashion we first create an `ObservableList` containing all the bosses for the `ListView` control's constructor. In my code I was politically correct by calling bosses *leaders*. The following code creates a `ListView` control:

```
// List of leaders
ObservableList<Person> leaders = getPeople();
final ListView<Person> leaderListView = new ListView<Person>(leaders);
```

Next, we create a cell factory to properly display the person's name in the `ListView` control. Because each item is a `Person` object, the `ListView` does not know how to render each row in the `ListView` control. We simply create a `javafx.util.Callback` generic type object by specifying the `ListView<Person>` and a `ListCell<Person>` data types. With your trusty NetBeans IDE, it will pregenerate things such as the

implementing method `call()`. Next is the variable cell of type `ListCell<Person>` (within the `call()` method), in which we create an anonymous inner class. The inner class must implement an `updateItem()` method. To implement the `updateItem()` method you will obtain the person information and update the `Label` control (`leadLbl`). Hopefully, you're still with me. The last thing is our tooltip, which is set.

Finally, we get to create a `TableView` control to display the employee base on the selected boss from the `ListView`. When creating a `TableView` we first create the column headers. Use this to create a table column:

```
TableColumn<String> firstNameCol = new TableColumn<String>("First Name");
firstNameCol.setProperty("firstName");
```

Once you have created a column, you'll notice the `setProperty()` method, which is responsible for calling the Person bean's property. So when the list of employees is put into the `TableView`, it will know how to pull the properties to be placed in each cell in the table.

Last is the implementation of the selection listener on the `ListViewer` in JavaFX called a selection item property (`selectionItemProperty()`). We simply create and add a `ChangeListener` to listen to selection events. When a user selects a boss, the `TableView` is cleared and populated with the boss' employees. Actually it is the magic of the `ObservableList` that notifies the `TableView` of changes. To populate the `TableView` via the `teamMembers` (`ObservableList`) variable:

```
teamMembers.clear();
teamMembers.addAll(observable.getValue().employeesProperty());
```

1-15. Organizing UI with Split Views

Problem

You want to split up a GUI screen by using split divider controls.

Solution

Use JavaFX's split pane control. The `javafx.scene.control.SplitPane` class is a UI control that enables you to divide a screen into frame-like regions. The split control allows the user to use the mouse to move the divider between any two split regions.

Shown here is the code to create the GUI application that utilizes the `javafx.scene.control.SplitPane` class to divide the screen into three windowed regions. The three windowed regions are a lefthand column, an upper-right region, and a lower-right region. In addition, you will be adding `Text` nodes into the three regions.

```
// Left and right split pane
SplitPane splitPane = new SplitPane();
splitPane.prefWidthProperty().bind(scene.widthProperty());
splitPane.prefHeightProperty().bind(scene.heightProperty());

VBox leftArea = new VBox(10);

for (int i = 0; i < 5; i++) {
    HBox rowBox = new HBox(20);
    final Text leftText = TextBuilder.create()
```

```
            .text("Left " + i)
            .translateX(20)
            .fill(Color.BLUE)
            .font(Font.font(null, FontWeight.BOLD, 20))
            .build();

        rowBox.getChildren().add(leftText);
        leftArea.getChildren().add(rowBox);
    }
    leftArea.setAlignment(Pos.CENTER);

    // Upper and lower split pane
    SplitPane splitPane2 = new SplitPane();
    splitPane2.setOrientation(Orientation.VERTICAL);
    splitPane2.prefWidthProperty().bind(scene.widthProperty());
    splitPane2.prefHeightProperty().bind(scene.heightProperty());

    HBox centerArea = new HBox();

    InnerShadow iShadow = InnerShadowBuilder.create()
        .offsetX(3.5f)
        .offsetY(3.5f)
        .build();
    final Text upperRight = TextBuilder.create()
        .text("Upper Right")
        .x(100)
        .y(50)
        .effect(iShadow)
        .fill(Color.LIME)
    .font(Font.font(null, FontWeight.BOLD, 35))
    .translateY(50)
        .build();
    centerArea.getChildren().add(upperRight);

    HBox rightArea = new HBox();

    final Text lowerRight = TextBuilder.create()
        .text("Lower Right")
        .x(100)
        .y(50)
        .effect(iShadow)
        .fill(Color.RED)
        .font(Font.font(null, FontWeight.BOLD, 35))
        .translateY(50)
        .build();
    rightArea.getChildren().add(lowerRight);

    splitPane2.getItems().add(centerArea);
    splitPane2.getItems().add(rightArea);

    // add left area
    splitPane.getItems().add(leftArea);
```

```
// add right area
splitPane.getItems().add(splitPane2);

// evenly position divider
ObservableList<SplitPane.Divider> dividers = splitPane.getDividers();
for (int i = 0; i < dividers.size(); i++) {
    dividers.get(i).setPosition((i + 1.0) / 3);
}

HBox hbox = new HBox();
hbox.getChildren().add(splitPane);
root.getChildren().add(hbox);
```

Figure 1-21 depicts the application using split pane controls.

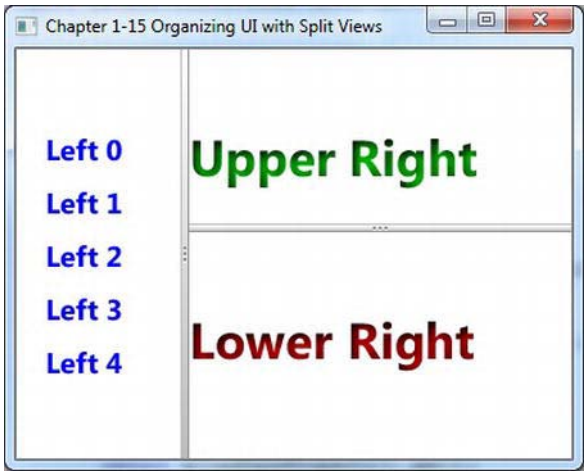

Figure 1-21. *Split views*

How It Works

If you've ever seen a simple RSS reader or the Javadocs, you'll notice that the screen is divided into sections with dividers that allow the user to adjust. In this recipe, three areas are on the left, upper right, and lower right.

I begin by creating a `SplitPane` that divides the left from the right area of the scene. Then I bind its width and height properties to the scene so the areas will take up the available space as the user resizes the Stage. Next I create a VBox layout control representing the left area. In the VBox (leftArea), I loop to generate a bunch of `Text` nodes. Next is creating the right side of the split pane. The following code snippet allows the split pane control (`SplitPane`) to divide horizontally:

```
SplitPane splitPane = new SplitPane();
splitPane.prefWidthProperty().bind(scene.widthProperty());
splitPane.prefHeightProperty().bind(scene.heightProperty());
```

Now we create the `SplitPane` to divide the area vertically forming the upper-right and lower-right region. Shown here is the code used to split a window region vertically:

```
// Upper and lower split pane
SplitPane splitPane2 = new SplitPane();
splitPane2.setOrientation(Orientation.VERTICAL);
```

At last we assemble the split panes and adjust the dividers to be positioned so that the screen real estate is divided evenly. The following code assembles the split panes and iterates through the list of dividers to update their positions:

```
splitPane.getItems().add(splitPane2);

// evenly position divider
ObservableList<SplitPane.Divider> dividers = splitPane.getDividers();
for (int i = 0; i < dividers.size(); i++) {
    dividers.get(i).setPosition((i + 1.0) / 3);
}

HBox hbox = new HBox();
hbox.getChildren().add(splitPane);
root.getChildren().add(hbox);
```

1-16. Adding Tabs to the UI

Problem

You want to create a GUI application with tabs.

Solution

Use JavaFX's tab and tab pane control. The tab (`javafx.scene.control.Tab`) and tab pane control (`javafx.scene.control.TabPane`) classes allow you to place graph nodes in individual tabs.

The following code example creates a simple application having menu options that allow the user to choose a tab orientation. The available tab orientations are top, bottom, left, and right.

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Chapter 1-16 Adding Tabs to a UI");
    Group root = new Group();
    Scene scene = new Scene(root, 400, 250, Color.WHITE);

    TabPane tabPane = new TabPane();

    MenuBar menuBar = new MenuBar();

    EventHandler<ActionEvent> action = changeTabPlacement(tabPane);

    Menu menu = new Menu("Tab Side");
    MenuItem left = new MenuItem("Left");
```

```
        left.setOnAction(action);
        menu.getItems().add(left);

        MenuItem right = new MenuItem("Right");
        right.setOnAction(action);
        menu.getItems().add(right);

        MenuItem top = new MenuItem("Top");
        top.setOnAction(action);
        menu.getItems().add(top);

        MenuItem bottom = new MenuItem("Bottom");
        bottom.setOnAction(action);
        menu.getItems().add(bottom);

        menuBar.getMenus().add(menu);

        BorderPane borderPane = new BorderPane();

        // generate 10 tabs
        for (int i = 0; i < 10; i++) {
            Tab tab = new Tab();
            tab.setText("Tab" + i);
            HBox hbox = new HBox();
            hbox.getChildren().add(new Label("Tab" + i));
            hbox.setAlignment(Pos.CENTER);
            tab.setContent(hbox);
            tabPane.getTabs().add(tab);
        }

        // add tab pane
        borderPane.setCenter(tabPane);

        // bind to take available space
        borderPane.prefHeightProperty().bind(scene.heightProperty());
        borderPane.prefWidthProperty().bind(scene.widthProperty());

        // added menu bar
        borderPane.setTop(menuBar);

        // add border Pane
        root.getChildren().add(borderPane);

        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private EventHandler<ActionEvent> changeTabPlacement(final TabPane tabPane) {
        return new EventHandler<ActionEvent>() {

            public void handle(ActionEvent event) {
                MenuItem mItem = (MenuItem) event.getSource();
```

```
                String side = mItem.getText();
                if ("left".equalsIgnoreCase(side)) {
                    tabPane.setSide(Side.LEFT);
                } else if ("right".equalsIgnoreCase(side)) {
                    tabPane.setSide(Side.RIGHT);
                } else if ("top".equalsIgnoreCase(side)) {
                    tabPane.setSide(Side.TOP);
                } else if ("bottom".equalsIgnoreCase(side)) {
                    tabPane.setSide(Side.BOTTOM);
                }
            }
        };
    }
```

Figure 1-22 displays the tabs application, which allows a user to change the tab orientation.

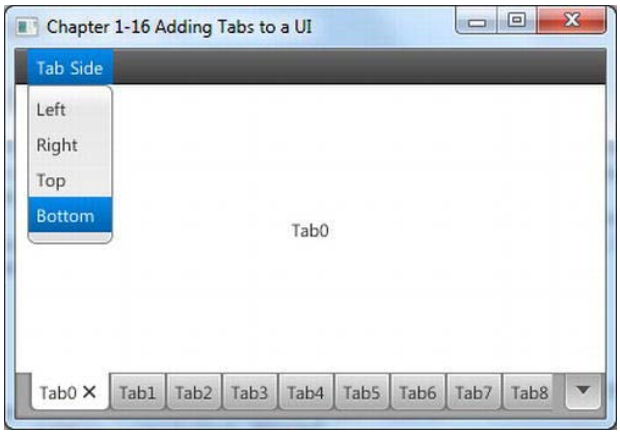

Figure 1-22. TabPane

How It Works

When you use the `TabPane` control, you might already know the orientation in which you want your tabs to appear. This application allows you to set the orientation by the menu options for Left, Right, Top, and Bottom.

To use the `TabPane` you will immediately notice how similar it is to Java Swing's `JTabbedPanel` class. Instead of adding `JPanel` instances, you simply add `javafx.scene.control.Tab` instances. The following code snippet adds `Tab` controls into a tab pane control:

```
TabPane tabPane = new TabPane();
Tab tab = new Tab();
tab.setText("Tab" + i);
tabPane.getTabs().add(tab);
```

When changing the orientation the `TabPane` control, use the `setSide()` method. The following code line sets the orientation of the tab pane control:

```
tabPane.setSide(Side.BOTTOM);
```

1-17. Developing a Dialog Box

Problem

You want to create an application that simulates a change password dialog box.

Solution

Use JavaFX's stage (`javafx.stage.Stage`) and scene (`javafx.scene.Scene`) APIs to create a dialog box.

The following source code listing is an application that simulates a change password dialog box. The application contains menu options to pop up the dialog box. In addition to the menu options, the user will have the ability to set the dialog box's modal state (`modality`).

```
/**
 * Developing A Dialog
 * @author cdea
 */
public class DevelopingADialog extends Application {

    static Stage LOGIN_DIALOG;
    static int dx = 1;
    static int dy = 1;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    private static Stage createLoginDialog(Stage parent, boolean modal) {
        if (LOGIN_DIALOG != null) {
            LOGIN_DIALOG.close();
        }
        return new MyDialog(parent, modal, "Welcome to JavaFX!");
    }

    @Override
    public void start(final Stage primaryStage) {
        primaryStage.setTitle("Chapter 1-17 Developing a Dialog");
        Group root = new Group();
        Scene scene = new Scene(root, 433, 312, Color.WHITE);

        MenuBar menuBar = new MenuBar();
        menuBar.prefWidthProperty().bind(primaryStage.widthProperty());

        Menu menu = new Menu("Home");
```

```
// add change password menu itme
MenuItem newItem = new MenuItem("Change Password", null);
newItem.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent event) {
        if (LOGIN_DIALOG == null) {
            LOGIN_DIALOG = createLoginDialog(primaryStage, true);
        }
        LOGIN_DIALOG.sizeToScene();
        LOGIN_DIALOG.show();
    }
});

menu.getItems().add(newItem);

// add separator
menu.getItems().add(new SeparatorMenuItem());

// add non modal menu item
ToggleGroup modalGroup = new ToggleGroup();
RadioMenuItem nonModalItem = RadioMenuItemBuilder.create()
        .toggleGroup(modalGroup)
        .text("Non Modal")
        .selected(true)
        .build();
nonModalItem.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent event) {
        LOGIN_DIALOG = createLoginDialog(primaryStage, false);
    }
});

menu.getItems().add(nonModalItem);

// add modal selection
RadioMenuItem modalItem = RadioMenuItemBuilder.create()
        .toggleGroup(modalGroup)
        .text("Modal")
        .selected(true)
        .build();
modalItem.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent event) {
        LOGIN_DIALOG = createLoginDialog(primaryStage, true);
    }
});
menu.getItems().add(modalItem);

// add separator
menu.getItems().add(new SeparatorMenuItem());
```

```
        // add exit
        MenuItem exitItem = new MenuItem("Exit", null);
        exitItem.setMnemonicParsing(true);
        exitItem.setAccelerator(new KeyCodeCombination(KeyCode.X,
KeyCombination.CONTROL_DOWN));
        exitItem.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent event) {
                Platform.exit();
            }
        });
        menu.getItems().add(exitItem);

        // add menu
        menuBar.getMenus().add(menu);

        // menu bar to window
        root.getChildren().add(menuBar);

        primaryStage.setScene(scene);
        primaryStage.show();

        addBouncyBall(scene);
    }

    private void addBouncyBall(final Scene scene) {

        final Circle ball = new Circle(100, 100, 20);
        RadialGradient gradient1 = new RadialGradient(0,
                .1,
                100,
                100,
                20,
                false,
                CycleMethod.NO_CYCLE,
                new Stop(0, Color.RED),
                new Stop(1, Color.BLACK));

        ball.setFill(gradient1);

        final Group root = (Group) scene.getRoot();
        root.getChildren().add(ball);

        Timeline tl = new Timeline();
        tl.setCycleCount(Animation.INDEFINITE);
        KeyFrame moveBall = new KeyFrame(Duration.seconds(.0200),
                new EventHandler<ActionEvent>() {

                    public void handle(ActionEvent event) {

                        double xMin = ball.getBoundsInParent().getMinX();
                        double yMin = ball.getBoundsInParent().getMinY();
                        double xMax = ball.getBoundsInParent().getMaxX();
```

```
                        double yMax = ball.getBoundsInParent().getMaxY();

                        // Collision - boundaries
                        if (xMin < 0 || xMax > scene.getWidth()) {
                            dx = dx * -1;
                        }
                        if (yMin < 0 || yMax > scene.getHeight()) {
                            dy = dy * -1;
                        }

                        ball.setTranslateX(ball.getTranslateX() + dx);
                        ball.setTranslateY(ball.getTranslateY() + dy);

                    }
                });

        tl.getKeyFrames().add(moveBall);
        tl.play();
    }
}

class MyDialog extends Stage {

    public MyDialog(Stage owner, boolean modality, String title) {
        super();
        initOwner(owner);
        Modality m = modality ? Modality.APPLICATION_MODAL : Modality.NONE;
        initModality(m);
        setOpacity(.90);
        setTitle(title);
        Group root = new Group();
        Scene scene = new Scene(root, 250, 150, Color.WHITE);
        setScene(scene);

        GridPane gridpane = new GridPane();
        gridpane.setPadding(new Insets(5));
        gridpane.setHgap(5);
        gridpane.setVgap(5);

        Label mainLabel = new Label("Enter User Name & Password");
        gridpane.add(mainLabel, 1, 0, 2, 1);


        Label userNameLbl = new Label("User Name: ");
        gridpane.add(userNameLbl, 0, 1);

        Label passwordLbl = new Label("Password: ");
        gridpane.add(passwordLbl, 0, 2);


        // username text field
        final TextField userNameFld = new TextField("Admin");
```

```
        gridpane.add(userNameFld, 1, 1);

        // password field
        final PasswordField passwordFld = new PasswordField();
        passwordFld.setText("drowssap");
        gridpane.add(passwordFld, 1, 2);


        Button login = new Button("Change");
        login.setOnAction(new EventHandler<ActionEvent>() {

            public void handle(ActionEvent event) {
                close();
            }
        });
        gridpane.add(login, 1, 3);
        GridPane.setHalignment(login, HPos.RIGHT);
        root.getChildren().add(gridpane);
    }
}
```

Figure 1-23 depicts our change password dialog box application with the Non Modal option enabled.

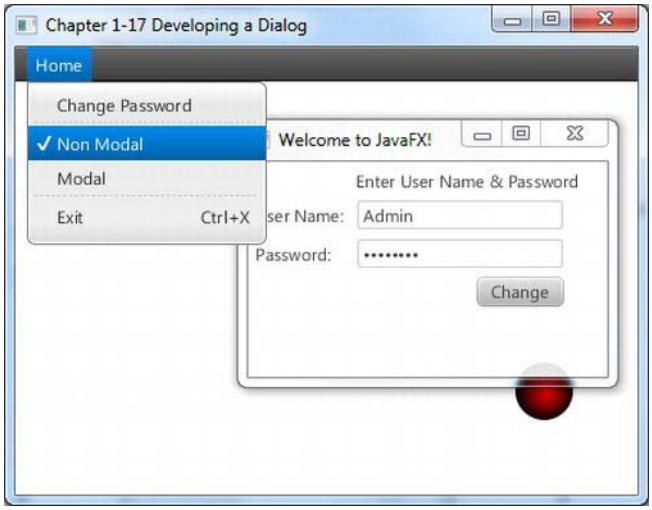

Figure 1-23. *Developing a dialog box*

How It Works

In this recipe we create a login screen using JavaFX. In doing that, we primarily focus our attention on the `javafx.stage.Stage` class. JavaFX uses an instance of a `javafx.stage.Stage` class to be shown to the user. When you extend from that `Stage` class, you have the opportunity (as in `Swing`) to pass in the

owning window in the constructor, which then calls the `initOwner()` method. Next is setting the modal state of the dialog box using the `initModality()` method. Following is a class that extends from the `Stage` class having a constructor initializing the owning stage and modal state:

```
class MyDialog extends Stage {

    public MyDialog(Stage owner, boolean modality, String title) {
        super();
        initOwner(owner);
        Modality m = modality ? Modality.APPLICATION_MODAL : Modality.NONE;
initModality(m);

        ...// The rest of the class
```

The rest of the code creates a scene (`Scene`) similar to the main application's `start()` method. Because login forms are pretty boring, I decided to create an animation of a bouncing ball while the user is busy changing the password in the dialog box. (You will see more about creating animation in recipe 2-2.)

CHAPTER 2

Graphics with JavaFX

Have you ever heard someone say, “When two worlds collide”? This expression is used when a person from a different background or culture is put in a situation where they are at odds and must face very hard decisions. When we build a GUI application needing animations, we are often in a collision course between business and gaming worlds.

In the ever-changing world of RIAs, you probably have noticed an increase of animations such as pulsing buttons, transitions, moving backgrounds, and so on. When GUI applications use animations, they can provide visual cues to the user to let them know what to do next. With JavaFX, you will be able to have the best of both worlds.

Figure 2-1 illustrates a simple drawing coming alive.

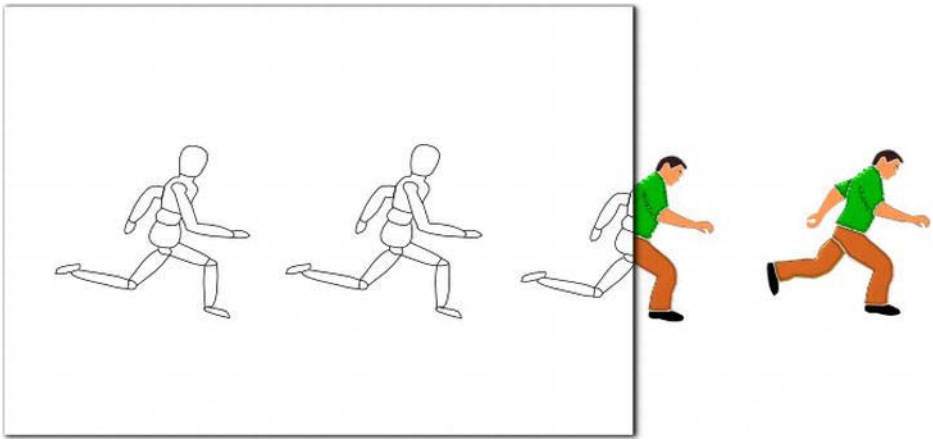

Figure 2-1. *Graphics with JavaFX*

In this chapter you will create images, animations, and Look ‘N’ Feels. Fasten your seatbelts; you’ll discover solutions to integrate cool game-like interfaces into our everyday applications.

■ **Note** Refer to Chapter 1 if you are new to JavaFX. Among other things, it will help you get an environment created in which you can be productive in using JavaFX.

2-1. Creating Images

Problem

There are photos in your file directory that you would like to quickly browse through and showcase.

Solution

Create a simple JavaFX image viewer application. The main Java classes used in this recipe are:

- `javafx.scene.image.Image`
- `javafx.scene.image.ImageView`
- `EventHandler<DragEvent>` classes

The following source code is an implementation of an image viewer application:

```
package javafx2introbyexample.chapter2.recipe2_01;

import java.io.File;
import java.util.ArrayList;
import java.util.List;
import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.DragEvent;
import javafx.scene.input.Dragboard;
import javafx.scene.input.MouseEvent;
import javafx.scene.input.TransferMode;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcBuilder;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.RectangleBuilder;
import javafx.stage.Stage;

/**
 * Creating Images
 * @author cdea
 */
public class CreatingImages extends Application {
    private List<String> imageFiles = new ArrayList<>();
    private int currentIndex = -1;
    public enum ButtonMove {NEXT, PREV};
```

```
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    Application.launch(args);
}

@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Chapter 2-1 Creating a Image");
    Group root = new Group();
    Scene scene = new Scene(root, 551, 400, Color.BLACK);


    // image view
    final ImageView currentImageView = new ImageView();

    // maintain aspect ratio
    currentImageView.setPreserveRatio(true);

    // resize based on the scene
    currentImageView.fitWidthProperty().bind(scene.widthProperty());

    final HBox pictureRegion = new HBox();
    pictureRegion.getChildren().add(currentImageView);
    root.getChildren().add(pictureRegion);

    // Dragging over surface
    scene.setOnDragOver(new EventHandler<DragEvent>() {
        @Override
        public void handle(DragEvent event) {
            Dragboard db = event.getDragboard();
            if (db.hasFiles()) {
                event.acceptTransferModes(TransferMode.COPY);
            } else {
                event.consume();
            }
        }
    });

    // Dropping over surface
    scene.setOnDragDropped(new EventHandler<DragEvent>() {

        @Override
        public void handle(DragEvent event) {
            Dragboard db = event.getDragboard();
            boolean success = false;
            if (db.hasFiles()) {
                success = true;
                String filePath = null;
                for (File file:db.getFiles()) {
                    filePath = file.getAbsolutePath();
```

```
                    currentIndex +=1;
                    imageFiles.add(currentIndex, filePath);

                    // absolute file name
                    System.out.println("file: " + file);
                    // the index in the list of file names
                    System.out.println("currentImageFileIndex = " + currentIndex);
                }

                // set new image as the image to show.
                Image imageimage = new Image(filePath);
                currentImageView.setImage(imageimage);

            }
            event.setDropCompleted(success);
            event.consume();
        }
    });


    // create slide controls
    Group buttonGroup = new Group();

    // rounded rect
    Rectangle buttonArea = RectangleBuilder.create()
            .arcWidth(15)
            .arcHeight(20)
            .fill(new Color(0, 0, 0, .55))
            .x(0)
            .y(0)
            .width(60)
            .height(30)
            .stroke(Color.rgb(255, 255, 255, .70))
            .build();

    buttonGroup.getChildren().add(buttonArea);
    // left control
    Arc leftButton = ArcBuilder.create()
            .type(ArcType.ROUND)
            .centerX(12)
            .centerY(16)
            .radiusX(15)
            .radiusY(15)
            .startAngle(-30)
            .length(60)
            .fill(new Color(1,1,1, .90))
            .build();

    leftButton.addEventHandler(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
        public void handle(MouseEvent me) {
            int indx = gotoImageIndex(ButtonMove.PREV);
            if (indx > -1) {
```

```
                String namePict = imageFiles.get(indx);
                final Image image = new Image(new File(namePict).getAbsolutePath());
                currentImageView.setImage(image);
            }
        }
    });
    buttonGroup.getChildren().add(leftButton);

    // right control
    Arc rightButton = ArcBuilder.create()
            .type(ArcType.ROUND)
            .centerX(12)
            .centerY(16)
            .radiusX(15)
            .radiusY(15)
            .startAngle(180-30)
            .length(60)
            .fill(new Color(1,1,1, .90))
            .translateX(40)
            .build();
    buttonGroup.getChildren().add(rightButton);

    rightButton.addEventHandler(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
        public void handle(MouseEvent me) {
            int indx = gotoImageIndex(ButtonMove.NEXT);
            if (indx > -1) {
                String namePict = imageFiles.get(indx);
                final Image image = new Image(new File(namePict).getAbsolutePath());
                currentImageView.setImage(image);
            }
        }
    });

    // move button group when scene is resized
    buttonGroup.translateXProperty().bind(scene.widthProperty().subtract(buttonArea.getWid
    th() + 6));

    buttonGroup.translateYProperty().bind(scene.heightProperty().subtract(buttonArea.getHe
    ight() + 6));
    root.getChildren().add(buttonGroup);

    primaryStage.setScene(scene);
    primaryStage.show();
}

/**
 * Returns the next index in the list of files to go to next.
 *
 * @param direction PREV and NEXT to move backward or forward in the list of
 * pictures.
 * @return int the index to the previous or next picture to be shown.
 */
```

```
    public int gotoImageIndex(ButtonMove direction) {
        int size = imageFiles.size();
        if (size == 0) {
            currentIndex = -1;
        } else if (direction == ButtonMove.NEXT && size > 1 && currentIndex < size - 1) {
            currentIndex += 1;
        } else if (direction == ButtonMove.PREV && size > 1 && currentIndex > 0) {
            currentIndex -= 1;
        }

        return currentIndex;
    }

}
```

Figure 2-2 depicts the drag-and-drop operation that gives the user visual feedback with a thumbnail-sized image over the surface. In the figure, I'm dragging the image onto the application window.

Figure 2-2. *Drag and drop in progress*

Figure 2-3 shows that the drop operation has succesfully loaded the image.

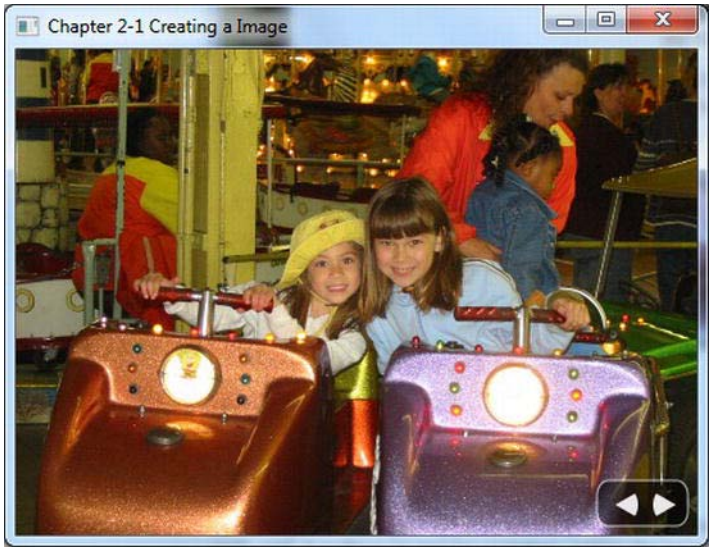

Figure 2-3. *Drop operation completed*

How It Works

This recipe is a simple application that allows you to view images having file formats such as `.jpg`, `.png`, and `.gif`. Loading an image requires using the mouse to drag and drop a file into the window area. The application also allows resizing of the window, which automatically causes scaling of the image's size while maintaining its aspect ratio. After a few images are successfully loaded, you will be able to page through each image conveniently by clicking the left and right button controls, as shown in Figure 2-3.

Before the code walk-through, let's discuss the application's variables. Table 2-1 describes instance variables for our sleek image viewer application.

Table 2-1. The CreatingImages Instance Variables

Variable	Data Type	Example	Description
`imageFiles`	`List<String>`	`/home/cdea/fun.jpg`	A list of `Strings`, each containing an image's absolute file path
`currentIndex`	`int`	0	A zero relative index number into the `imageFiles` list; negative 1 means no images to view
`NEXT`	`enum`	-	User clicks the right arrow button
`PREV`	`enum`	-	User clicks the left arrow button

When dragging an image into the application, the `imageFiles` variable will cache the absolute file path as a String instead of the actual image file in order to save space in memory. If a user drags the same image file into the display area, the list will contain duplicate strings representing the image file. As an image is being displayed, the `currentIndex` variable contains the index into the `imageFiles` list. That `imageFiles` list points to the String representing the current image file. As the user clicks the buttons to display the previous and next image, the `currentIndex` will decrement or increment, respectively. Next, you will walk through the code detailing the steps on how to load and display an image. Later I will discuss the steps on paging through each image with the next and previous buttons.

You will begin by instantiating an instance of a `javafx.scene.image.ImageView` class. The `ImageView` class is a graph node (`Node`) used to display an already loaded `javafx.scene.image.Image` object. Using the `ImageView` node will enable you to create special effects on the image to be displayed without manipulating the physical `Image`. To avoid performance degradation when rendering many effects, you can use numerous `ImageView` objects that reference a single `Image` object. Many types of effects include blurring, fading, and transforming an image.

One of the requirements is preserving the displayed image's aspect ratio as the user resizes the window. Here, you will simply call the `setPreserveRatio()` method with a value of `true` to preserve the image's aspect ratio. Remember that because the user resizes the window, you want to bind the width of the `ImageView` to the `Scene`'s width to allow the scaling of the image to take effect. After setting up the `ImageView`, you will want to pass it to an `HBox` instance (`pictureRegion`) to be put into the scene. The following code creates the `ImageView` instance, preserves the aspect ratio, and scales the image:

```
// image view
final ImageView currentImageView = new ImageView();

// maintain aspect ratio
currentImageView.setPreserveRatio(true);

// resize based on the scene
currentImageView.fitWidthProperty().bind(scene.widthProperty());
```

Next, I want to introduce JavaFX's new native drag-and-drop support, which offers many scenarios a user can perform, such as dragging visual objects from an application to be dropped into another application. In this scenario, the user will be dragging an image file from the host windowing operating system to your image viewer application. When performing this scenario, you must create `EventHandler` objects to listen to `DragEvents`. To fulfill this requirement, you only need to set-up a Scene's drag-over and drag-dropped event handler methods.

To setup the drag-over attribute, you will be calling the Scene's `setOnDragOver()` method with the appropriate generics `EventHandler<DragEvent>` type. Here you will implement the `handle()` method to listen to the drag-over event (`DragEvent`). In the `handle()` method notice the event (`DragEvent`) object's invocation to the `getDragboard()` method. The call to `getDragboard()` will return the drag source (`Dragboard`), better known as the *clipboard*. When you obtain the `Dragboard` object, you can determine and validate what is being dragged over the surface. In this scenario, you are trying to determine whether the `Dragboard` object contains any files. If so, call the event object's `acceptTransferModes()` by passing in the constant `TransferMode.COPY` to provide visual feedback to the user of the application (refer to Figure 2-2). Otherwise it should consume the event by calling the `event.consume()` method. The following code demonstrates setting up a Scene's `OnDragOver` attribute by instantiating an inner class of type `EventHandler` with a formal type parameter `<DragEvent>` and overriding its handle() method:

```
// Dragging over surface
scene.setOnDragOver(new EventHandler<DragEvent>() {
    @Override
```

```
        public void handle(DragEvent event) {
            Dragboard db = event.getDragboard();
            if (db.hasFiles()) {
                event.acceptTransferModes(TransferMode.COPY);
            } else {
                event.consume();
            }
        }
    });
```

Once the drag-over event handler attribute is set, you must create a drag-dropped event handler attribute in order that it may finalize the operation. Listening to a drag-dropped event is similar to listening to a drag-over event in which you will implement the `handle()` method. Once again you obtain the `Dragboard` object from the event to determine whether the clipboard contains any files. If so, you will iterate over the list of files and their names to be added to the `imageFiles` list. This demonstrates setting up a Scene's `OnDragDropped` attribute by instantiating an inner class of type `EventHandler` with a formal type parameter `<DragEvent>` and overriding its `handle()` method:

```
    // Dropping over surface
    scene.setOnDragDropped(new EventHandler<DragEvent>() {

        @Override
        public void handle(DragEvent event) {
            Dragboard db = event.getDragboard();
            boolean success = false;
            if (db.hasFiles()) {
                success = true;
                String filePath = null;
                for (File file:db.getFiles()) {
                    filePath = file.getAbsolutePath();

                    currentIndex +=1;
                    imageFiles.add(currentIndex, filePath);
                }

                // set new image as the image to show.
                Image imageimage = new Image(filePath);
                currentImageView.setImage(imageimage);

            }
            event.setDropCompleted(success);
            event.consume();
        }
    });
```

As the last file is determined, the current image is displayed. The following code demonstrates loading an image to be displayed:

```
    // set new image as the image to show.
    Image imageimage = new Image(filePath);
    currentImageView.setImage(imageimage);
```

For the last requirements relating to the image viewer application, you will be creating simple controls that allow the user to view the next or previous image. I emphasize "simple" controls because JavaFX contains two other methods for creating custom controls. One way (CSS Styling) is discussed later in recipe 2-5. To explore the other alternative, please refer to the Javadoc on the Skin and Skinnable APIs.

To create simple buttons I used Java FX's `javafx.scene.shape.Arc` to build the left and right arrows on top of a small transparent rounded rectangle `javafx.scene.shape.Rectangle`. Next is adding an `EventHandler` that listens to mouse-pressed events that will load and display the appropriate image based on enums `ButtonMove.PREV` and `ButtonMove.NEXT`. You will find the `EventHandler` indispensible and useful in so many ways. When instantiating a generics class with a type variable between the < and > symbols, the same type variable will be defined in the `handle()`'s signature. When implementing the `handle()` method I determine which button was pressed; it then returns the index into the `imageFiles` list of the next image to display. When loading an image using the `Image` class you can load images from the file system or a `URL`, but in this recipe I am using a `File` object. The following code instantiates an `EventHandler<MouseEvent>` with a `handle()` method to display the previous image in the `imageFiles` list:

```
Arc leftButton = //... create an Arc
leftButton.addEventHandler(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
   public void handle(MouseEvent me) {
      int indx = gotoImageIndex(ButtonMove.PREV);
      if (indx > -1) {
         String namePict = imageFiles.get(indx);
         final Image image = new Image(new File(namePict).getAbsolutePath());
         currentImageView.setImage(image);
      }
  }
});
```

The right button's (`rightButton`) event handler is identical, so I trust you get the idea. The only thing different is determining whether the previous button or the next button was pressed via the `ButtonMove` enum. This is passed to the `gotoImageIndex()` method to determine whether an image is available in that direction.

To finish the image viewer application, you have to bind the rectangular buttons control to the Scene's width and height, which repositions the control as the user resizes the window. Here, I bind the `translateXProperty()`to the Scene's width property by subtracting the `buttonArea's` width (Fluent API). I also bind the `translateYProperty()` based on the Scene's height property. Once your buttons control is bound, your user will experience user interface goodness. The following code uses the Fluent API to bind the button control's properties to the Scene's properties:

```
// move button group when scene is resized
buttonGroup.translateXProperty().bind(scene.widthProperty().subtract(buttonArea.getWid
th() + 6));

buttonGroup.translateYProperty().bind(scene.heightProperty().subtract(buttonArea.getHe
ight() + 6));
root.getChildren().add(buttonGroup);
```

2-2. Generating an Animation

Problem

You want to generate an animation. For example, you want to create a news ticker and photo viewer application with the following requirements:

- It will have a news ticker control that scrolls to the left.
- It will fade out the current picture and fade in the next picture as the user clicks the button controls.
- It will fade in and out button controls when the cursor moves in and out of the scene area, respectively.

Solution

Create animated effects by accessing JavaFX's animation APIs (`javafx.animation.*`).To create a news ticker, you need the following classes:

- `javafx.animation.TranslateTransition`
- `javafx.util.Duration`
- `javafx.event.EventHandler<ActionEvent>`
- `javafx.scene.shape.Rectangle`

To fade out the current picture and fade in next picture, you need the following classes:

- `javafx.animation.SequentialTransition`
- `javafx.animation.FadeTransition`
- `javafx.event.EventHandler<ActionEvent>`
- `javafx.scene.image.Image`
- `javafx.scene.image.ImageView`
- `javafx.util.Duration`

To fade in and out button controls when the cursor moves into and out of the scene area, respectively, the following classes are needed:

- `javafx.animation.FadeTransition`
- `javafx.util.Duration`

Shown here is the code used to create a news ticker control:

```
// create ticker area
final Group tickerArea = new Group();
final Rectangle tickerRect = RectangleBuilder.create()
        .arcWidth(15)
```

```
        .arcHeight(20)
        .fill(new Color(0, 0, 0, .55))
        .x(0)
        .y(0)
        .width(scene.getWidth() - 6)
        .height(30)
        .stroke(Color.rgb(255, 255, 255, .70))
        .build();


Rectangle clipRegion = RectangleBuilder.create()
        .arcWidth(15)
        .arcHeight(20)
        .x(0)
        .y(0)
        .width(scene.getWidth() - 6)
        .height(30)
        .stroke(Color.rgb(255, 255, 255, .70))
        .build();

tickerArea.setClip(clipRegion);

// Resize the ticker area when the window is resized
tickerArea.setTranslateX(6);
tickerArea.translateYProperty().bind(scene.heightProperty().subtract(tickerRect.getHeight(
) + 6));
tickerRect.widthProperty().bind(scene.widthProperty().subtract(buttonRect.getWidth() +
16));
clipRegion.widthProperty().bind(scene.widthProperty().subtract(buttonRect.getWidth() +
16));
tickerArea.getChildren().add(tickerRect);


root.getChildren().add(tickerArea);



// add news text
Text news = TextBuilder.create()
        .text("JavaFX 2.0 News! | 85 and sunny | :)")
        .translateY(18)
        .fill(Color.WHITE)
        .build();
tickerArea.getChildren().add(news);

final TranslateTransition ticker = TranslateTransitionBuilder.create()
        .node(news)
        .duration(Duration.millis((scene.getWidth()/300) * 15000))
        .fromX(scene.widthProperty().doubleValue())
        .toX(-scene.widthProperty().doubleValue())
        .fromY(19)
        .interpolator(Interpolator.LINEAR)
```

```
        .cycleCount(1)
        .build();

// when ticker has finished reset and replay ticker animation
ticker.setOnFinished(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae){
        ticker.stop();
        ticker.setFromX(scene.getWidth());
        ticker.setDuration(new Duration((scene.getWidth()/300) * 15000));
        ticker.playFromStart();
    }
});

ticker.play();
```

Here is the code used to fade out the current picture and fade in next picture:

```
// previous button
Arc prevButton = // create arc ...

prevButton.addEventHandler(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {

    public void handle(MouseEvent me) {
        int indx = gotoImageIndex(PREV);
        if (indx > -1) {
            String namePict = imagesFiles.get(indx);
            final Image nextImage = new Image(new File(namePict).getAbsolutePath());
            SequentialTransition seqTransition = transitionByFading(nextImage,
currentImageView);
            seqTransition.play();
        }
    }
});

buttonGroup.getChildren().add(prevButton);

// next button
Arc nextButton = //... create arc

buttonGroup.getChildren().add(nextButton);

nextButton.addEventHandler(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {

    public void handle(MouseEvent me) {
        int indx = gotoImageIndex(NEXT);
        if (indx > -1) {
            String namePict = imagesFiles.get(indx);
            final Image nextImage = new Image(new File(namePict).getAbsolutePath());
            SequentialTransition seqTransition = transitionByFading(nextImage,
currentImageView);
            seqTransition.play();

        }
```

```
            }
        });

    //... the rest of the start(Stage primaryStage) method


    public int gotoImageIndex(int direction) {
        int size = imagesFiles.size();
        if (size == 0) {
            currentIndexImageFile = -1;
        } else if (direction == NEXT && size > 1 && currentIndexImageFile < size - 1) {
            currentIndexImageFile += 1;
        } else if (direction == PREV && size > 1 && currentIndexImageFile > 0) {
            currentIndexImageFile -= 1;
        }

        return currentIndexImageFile;
    }

    public SequentialTransition transitionByFading(final Image nextImage, final ImageView
    imageView) {
        FadeTransition fadeOut = new FadeTransition(Duration.millis(500), imageView);
        fadeOut.setFromValue(1.0);
        fadeOut.setToValue(0.0);
        fadeOut.setOnFinished(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent ae) {
                imageView.setImage(nextImage);
            }
        });
        FadeTransition fadeIn = new FadeTransition(Duration.millis(500), imageView);
        fadeIn.setFromValue(0.0);
        fadeIn.setToValue(1.0);
        SequentialTransition seqTransition = SequentialTransitionBuilder.create()
            .children(fadeOut, fadeIn)
            .build();
        return seqTransition;
    }
```

The following code is used to fade in and out the button controls when the cursor moves into and out of the scene area, respectively:

```
        // Fade in button controls
        scene.setOnMouseEntered(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent me) {
                FadeTransition fadeButtons = new FadeTransition(Duration.millis(500),
        buttonGroup);
                fadeButtons.setFromValue(0.0);
                fadeButtons.setToValue(1.0);
                fadeButtons.play();
            }
        });

        // Fade out button controls
```

```
    scene.setOnMouseExited(new EventHandler<MouseEvent>() {
        public void handle(MouseEvent me) {
            FadeTransition fadeButtons = new FadeTransition(Duration.millis(500),
    buttonGroup);
            fadeButtons.setFromValue(1);
            fadeButtons.setToValue(0);
            fadeButtons.play();
        }
    });
```

Figure 2-4 shows the photo viewer application with a ticker control at the bottom region of the screen.

Figure 2-4. *Photo viewer with a news ticker*

How It Works

In the photo viewer application I decided to incorporate animation effects. The main animation effects I focus on are translating and fading. First, you will create a news ticker control that scrolls `Text` nodes to the left by using a translation transition (`javafx.animation.TranslateTransition`). Next, you will apply another fading effect when the user clicks the previous and next buttons to transition from the current image to the next. To perform this effect, you will use a compound transition (`javafx.animation.SequentialTransition`) consisting of multiple animations. Finally, to create the effect of the button controls fading in and out based on where the mouse is located, you will need a fade transition (`javafx.animation.FadeTransition`).

Before I begin to discuss the steps to fulfill the requirements, I want to mention the basics of JavaFX animation. The JavaFX animation API allows you to assemble timed events that can interpolate over a node's attribute values to produce animated effects. Each timed event is called a keyframe (`KeyFrame`), which is responsible for interpolating over a `Node`'s property over a period of time

(javafx.util.Duration). Knowing that a keyframe's job is to operate on a Node's property value, you will have to create an instance of a KeyValue class that will reference the desired Node property. The idea of interpolation is simply the distributing of values between a start and end value. An example is to move a rectangle by its current x position (zero) to 100 pixels in 1000 milliseconds; in other words, move the rectangle 100 pixels to the right, spanning one second. Shown here is a keyframe and key value to interpolate a rectangle's x property for 1000 milliseconds:

```
final Rectangle rectangle = new Rectangle(0, 0, 50, 50);
KeyValue keyValue = new KeyValue(rectangle.xProperty(), 100);
KeyFrame keyFrame = new KeyFrame(Duration.millis(1000), keyValue);
```

When creating many keyframes that are assembled consecutively, you need to create a TimeLine. Because TimeLine is a subclass of javafx.animation.Animation, there are standard attributes such as its cycle count and auto-reverse that can be set. The *cycle count* is the number of times you want the timeline to play the animation. If you want the cycle count to play the animation indefinitely, use the value Timeline.INDEFINITE. The auto-reverse is the capability for the animation to play the timeline backward. By default, the cycle count is set to 1, and the auto-reverse is set to false. When adding keyframes you will simply add them using the getKeyFrames().add() method on the TimeLine object. The following code snippet demonstrates a timeline playing indefinitely and auto-reverse set to true:

```
Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
timeline.getKeyFrames().add(keyFrame);
timeline.play();
```

With this knowledge of timelines you can animate any graph node in JavaFX. Although you have the ability to create timelines in a low-level way, it can become very cumbersome. You are probably wondering whether there are easier ways to express common animations. Good news! JavaFX has transitions (Transition), which are convenience classes to perform common animated effects. Some of the common animation effects classes are these:

- javafx.animation.FadeTransition
- javafx.animation.PathTransition
- javafx.animation.ScaleTransition
- javafx.animation.TranslateTransition

To see more transitions, see javafx.animation in the Javadoc. Because Transition objects are also subclasses of the javafx.animation.Animation class, you will have the opportunity to set the cycle count and auto-reverse attributes. In this recipe you will be focusing on two transition effects: translate transition (TranslateTransition) and fade transition (FadeTransition).

The first requirement in our problem statement is to create a news ticker. When creating a news ticker control, Text nodes will scroll from right to left inside a rectangular region. When the text scrolls to the left edge of the rectangular region you will want the text to be clipped to create a view port that only shows pixels inside of the rectangle. Here, I first create a Group to hold all the components that comprise a ticker control. Next is the creation of a rectangle using the RectangleBuilder to build a white rounded rectangle filled with 55 percent opacity. After creating the visual region I create a similar rectangle that represents the clipped region using the setClip(someRectangle) method on the Group object. Figure 2-5 shows a rounded rectangular area as a clip region:

Figure 2-5. *Setting the clip region on the Group object*

Once the ticker control is created, you will bind the translate Y based on the Scene's height property minus the ticker control's height. You will also bind the ticker control's width property based on the width of scene minus the button control's width. By binding these properties, the ticker control can change its size and position whenever a user resizes the application window. This makes the ticker control appear to float at the bottom of the window. The following code binds the ticker control's translate Y, width, and clip region's width property:

```
tickerArea.translateYProperty().bind(scene.heightProperty().subtract(tickerRect.getHeight() +
6));
tickerRect.widthProperty().bind(scene.widthProperty().subtract(buttonRect.getWidth() + 16));
clipRegion.widthProperty().bind(scene.widthProperty().subtract(buttonRect.getWidth() + 16));
tickerArea.getChildren().add(tickerRect);
```

Now that you have finished creating a ticker control, you will need to create news to feed into it. You will create a `Text` node with text that represents a news feed. To add a newly created `Text` node to the ticker control, call its `getChildren().add()` method. The following code adds a `Text` node to the ticker control:

```
final Group tickerArea = new Group();
final Rectangle tickerRect = //...
Text news = TextBuilder.create()
    .text("JavaFX 2.0 News! | 85 and sunny | :)")
    // ... more properties defined
    .build();
// add news to ticker control
tickerArea.getChildren().add(news);
```

Next is scrolling the `Text` node from right to left using JavaFX's `TranslateTransition` API. Like many JavaFX classes, in which there are builder classes to easily create objects, you will be using the `TranslateTransition` class' associated builder class called `TranslateTransitionBuilder`. The first step is to set the target node to perform the `TranslateTransition`. Then you will set the duration, which is the total amount of time the `TranslateTransition` will spend when animating. A `TranslateTransition` simplifies the creation of an animation by exposing convenience methods that operate on a `Node`'s translate X and Y properties. The convenience methods are prepended with `from` and `to`. For instance, in the scenario in which you use translate X on a `Text` node, there are methods `fromX()` and `toX()`. The `fromX()` is the starting value and the `toX()` is the end value that will be interpolated. Next, you will set the `TranslateTransition` to a linear transition (`Interpolator.LINEAR`) to interpolate evenly between the start and end values. To see more interpolator types or to see how to create custom interpolators, see the Javadoc on `javafx.animation.Interpolators`. Finally, I set the cycle count to 1, which will animate the ticker once based on the specified duration. The following code snippet details creating a `TranslateTransition` that animates a `Text` node from right to left:

```
final TranslateTransition ticker = TranslateTransitionBuilder.create()
        .node(news)
        .duration(Duration.millis((scene.getWidth()/300) * 15000))
        .fromX(scene.widthProperty().doubleValue())
```

```
            .toX(-scene.widthProperty().doubleValue())
            .fromY(19)
            .interpolator(Interpolator.LINEAR)
            .cycleCount(1)
            .build();
```

When the ticker's news has scrolled completely off of the ticker area to the far left of the Scene, you will want to stop and replay the news feed from the start (the far right). To do this you will create an instance of an `EventHandler<ActionEvent>` object to be set on the ticker (`TranslateTransition`) object using the `setOnFinished()` method. Shown here is how to replay the `TranslateTransition` animation:

```
// when window resizes width wise the ticker will know how far to move
ticker.setOnFinished(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae){
        ticker.stop();
        ticker.setFromX(scene.getWidth());
        ticker.setDuration(new Duration((scene.getWidth()/300) * 15000));
        ticker.playFromStart();
    }
});
```

Once the animation is defined, you simply invoke the `play()` method to get it started. The following code snippet shows how to play a `TranslateTransition`:

```
ticker.play();
```

Now that you have a better understanding of animated transitions, what about a transition that can trigger any number of transitions? JavaFX has two transitions that provide this behavior. The two transitions can invoke individual dependent transitions sequentially or in parallel. In this recipe I use a sequential transition (`SequentialTransition`) to contain two `FadeTransitions` in order to fade out the current image displayed and to fade-in the next image into view. When creating the previous and next button's event handlers, you first determine the next image to be displayed by calling the `gotoImageIndex()` method. Once the next image to be displayed is determined, you will call the `transitionByFading()` method, which returns an instance of a `SequentialTransition`. When calling the `transitionByFading()` method, you'll notice the creation of two `FadeTransitions`. The first transition will change the opacity level from 1.0 to 0.0 to fade out the current image, and the second transition will interpolate the opacity level from 0.0 to 1.0, fading in the next image that then becomes the current image. At last the two `FadeTransitions` are added to the `SequentialTransition` and returned to the caller. The following code creates two `FadeTransitions` and adds them to a `SequentialTransition`:

```
FadeTransition fadeOut = new FadeTransition(Duration.millis(500), imageView);
fadeOut.setFromValue(1.0);
fadeOut.setToValue(0.0);
fadeOut.setOnFinished(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
         imageView.setImage(nextImage);
    }
});

FadeTransition fadeIn = new FadeTransition(Duration.millis(500), imageView);
fadeIn.setFromValue(0.0);
fadeIn.setToValue(1.0);
```

```
SequentialTransition seqTransition = SequentialTransitionBuilder.create()
        .children(fadeOut, fadeIn)
        .build();
```

For the last requirements relating to fading in and out, you use the button controls. You will yet again use the `FadeTransition` to create a ghostly animated effect. For starters, just like any event you are interested in you will be creating an `EventHandler` (more specifically, an `EventHandler<MouseEvent>`). It is easy peasy to add mouse events to the Scene; all you have to do is override the `handle()` method where the inbound parameter is a `MouseEvent` type (the same as its formal type parameter). Inside of the `handle()` method, you will create an instance of a `FadeTransition` object by using the constructor that takes the duration and node as parameters. Next, you'll notice the `setFromValue()` and `setToValue()` methods that are called to interpolate values between 1.0 and 0.0 for the opacity level, causing the fade-in effect to occur. The following code adds an `EventHandler` to create the fade-in effect when the mouse cursor is positioned inside of the Scene:

```
// Fade in button controls
scene.setOnMouseEntered(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        FadeTransition fadeButtons = new FadeTransition(Duration.millis(500),
buttonGroup);
        fadeButtons.setFromValue(0.0);
        fadeButtons.setToValue(1.0);
        fadeButtons.play();
    }
});
```

Last but not least, the fade-out `EventHandler` is basically the same as the fade-in, except that the opacity `From` and `To` values are from 1.0 to 0.0, which make the buttons vanish mysteriously when the mouse pointer moves off the Scene area.

2-3. Animating Shapes Along a Path

Problem

You want to create a way to animate shapes along a path.

Solution

Create an application that allows a user to draw the path for a shape to follow. The main Java classes used in this recipe are these:

- `javafx.animation.PathTransition`
- `javafx.animation.PathTransitionBuilder`
- `javafx.scene.input.MouseEvent`
- `javafx.event.EventHandler`
- `javafx.geometry.Point2D`
- `javafx.scene.shape.LineTo`

- javafx.scene.shape.MoveTo
- javafx.scene.shape.Path

The following code demonstrates drawing a path for a shape to follow:

```
/**
 * Working with the Scene Graph
 * @author cdea
 */
public class WorkingWithTheSceneGraph extends Application {

    Path onePath = new Path();
    Point2D anchorPt;
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chapter 2-3 Working with the Scene Graph");

        final Group root = new Group();

        // add path
        root.getChildren().add(onePath);

        final Scene scene = SceneBuilder.create()
                .root(root)
                .width(300)
                .height(250)
                .fill(Color.WHITE)
                .build();

        RadialGradient gradient1 = new RadialGradient(0,
                .1,
                100,
                100,
                20,
                false,
                CycleMethod.NO_CYCLE,
                new Stop(0, Color.RED),
                new Stop(1, Color.BLACK));

        // create a sphere
        final Circle sphere = CircleBuilder.create()
                .centerX(100)
                .centerY(100)
                .radius(20)
                .fill(gradient1)
```

```
        .build();

// add sphere
root.getChildren().addAll(sphere);

// animate sphere by following the path.
final PathTransition pathTransition = PathTransitionBuilder.create()
    .duration(Duration.millis(4000))
    .cycleCount(1)
    .node(sphere)
    .path(onePath)
    .orientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT)
    .build();

// once finished clear path
pathTransition.onFinishedProperty().set(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event){
        onePath.getElements().clear();
    }
});

// starting initial path
scene.onMousePressedProperty().set(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent event){
        // clear path
        onePath.getElements().clear();
        // start point in path
        anchorPt = new Point2D(event.getX(), event.getY());
        onePath.setStrokeWidth(3);
        onePath.setStroke(Color.BLACK);
        onePath.getElements().add(new MoveTo(anchorPt.getX(), anchorPt.getY()));
    }
});

// dragging creates lineTos added to the path
scene.onMouseDraggedProperty().set(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent event){
        onePath.getElements().add(new LineTo(event.getX(), event.getY()));
    }
});

// end the path when mouse released event
scene.onMouseReleasedProperty().set(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent event){
        onePath.setStrokeWidth(0);
        if (onePath.getElements().size() > 1) {
            pathTransition.stop();
            pathTransition.playFromStart();
        }
    }
});
```

```
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Figure 2-6 shows the drawn path the circle will follow. When the user performs a mouse release, the drawn path will disappear, and the red ball will follow the path drawn earlier.

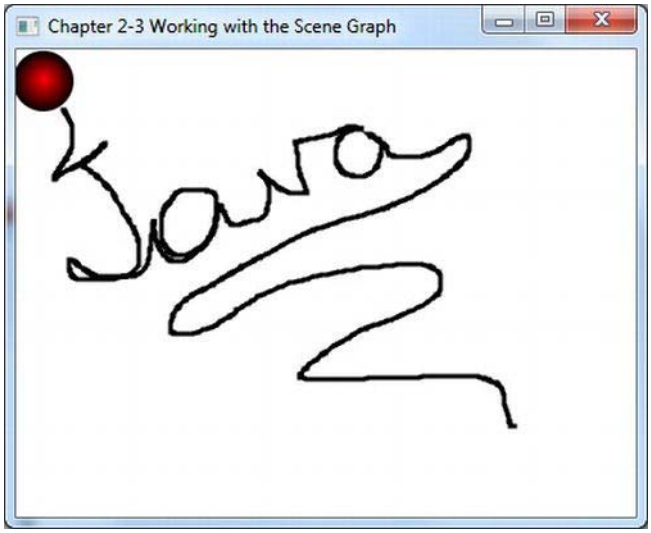

Figure 2-6. *Path transition*

How It Works

In this recipe you'll be creating a simple application enabling you to animate objects by following a drawn path on the Scene graph. To make things simple you will be using one shape (`Circle`) that will perform a path transition (`javafx.animation.PathTransition`). You will allow the user to draw a path on the scene surface by pressing the mouse button like a drawing program. Once you are satisfied with the path drawn, you will release the mouse press that triggers the red ball to follow the path similar to objects moving through pipes inside a building.

You will create two instance variables to maintain the coordinates that make up the path. To hold the path being drawn, you will create an instance of a `javafx.scene.shape.Path` object. You also should know that the path instance should be added to the Scene graph before the start of the application. Shown here is adding the instance variable `onePath` onto the Scene graph:

```
// add path
root.getChildren().add(onePath);
```

Next, you will create an instance variable `anchorPt` (`javafx.geometry.Point2D`) that will hold the path's starting point. Later, you will see how these variables will be updated based on mouse events. Shown here are the instance variables that maintain the currently drawn path:

```
Path onePath = new Path();
Point2D anchorPt;
```

First, let's create a shape that will be animated. In this scenario, you will be creating a cool-looking red ball. To create a spherical-looking ball you will create a gradient color `RadialGradient` that will be used to paint or fill a circle shape. (Refer to recipe 1-6 for how to fill shapes with gradient paint.) Once you have created the red spherical ball you need to create `PathTransition` object to perform the path following animation. By using the convenient `PathTransitionBuilder` class you simply set the duration to four seconds and the cycle count to one. The cycle count is the number of times the animation cycle will occur. Next, you will set the node to reference the red ball (`sphere`). Then, you will set the `path()` method to the instance variable `onePath`, which contains all the coordinates and lines that make up a drawn path. After setting the path for the sphere to animate, you should specify how the shape will follow the path such as perpendicular to a tangent point on the path. The following code creates an instance of a path transition:

```
// animate sphere by following the path.
final PathTransition pathTransition = PathTransitionBuilder.create()
    .duration(Duration.millis(4000))
    .cycleCount(1)
    .node(sphere)
    .path(onePath)
    .orientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT)
    .build();
```

After the creation of your path transition you will want it to clean up when the animation is completed. To reset or clean up the path variable when the animation is finished, you will create and add an event handler to listen to the `onFinished` property event on the path transition object. The following code snippet adds an event handler to clear the current path information:

```
// once finished clear path
pathTransition.onFinishedProperty().set(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent event){
      onePath.getElements().clear();
   }
});
```

With the shape and transition all set up, you will next respond to mouse events that will update the instance variable mentioned earlier. You will be listening to mouse events occurring on the `Scene` object. Here, you will once again rely on creating event handlers to be set on the Scene's `onMouseXXXProperty` methods where the `XXX` denotes the actual mouse event name such as pressed, dragged, and released.

When a user draws a path, he or she will perform a mouse press event to begin the start of the path. To listen to a mouse-pressed event, you will create an event handler with a formal type parameter of `MouseEvent`. Here you will override the `handle()` method. As a mouse-pressed event occurs, you want to clear the instance variable `onePath` of any prior drawn path information. Next, you will simply set the stroke width and color of the path so the user can see the path being drawn. Finally, you will add the starting point to the path using an instance of a `MoveTo` object. Shown here is the handler code to respond when the user performs a mouse press:

```
// starting initial path
scene.onMousePressedProperty().set(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent event){
```

```
                // clear path
                onePath.getElements().clear();
                // start point in path
                anchorPt = new Point2D(event.getX(), event.getY());
                onePath.setStrokeWidth(3);
                onePath.setStroke(Color.BLACK);
                onePath.getElements().add(new MoveTo(anchorPt.getX(), anchorPt.getY()));
            }
        });
```

Once the mouse-pressed event handler is in place, you will be creating another handler for mouse-dragged events. Again, you will look for the Scene's `onMouseXXXProperty()` methods that correspond to the proper mouse event that you care about. In this case, it will be the `onMouseDraggedProperty()` that you want to set. Inside the overridden `handle()` method you will be taking mouse coordinates that will be converted to `LineTo` objects to be added to the path (`Path`). These `LineTo` objects are instances of path element (`javafx.scene.shape.PathElement`) as discussed in recipe 1-5. The following code is an event handler responsible for mouse-dragged events:

```
// dragging creates lineTos added to the path
scene.onMouseDraggedProperty().set(new EventHandler<MouseEvent>() {
        public void handle(MouseEvent event){
                onePath.getElements().add(new LineTo(event.getX(), event.getY()));
        }
});
```

Finally, you will be creating an event handler to listen to a mouse-released event. When a user releases the mouse, the path's stroke is set to zero to appear as if it were removed. Then you will reset the path transition by stopping it and playing it from the start. The following code is an event handler responsible for mouse-released event:

```
    // end the path when mouse released event
    scene.onMouseReleasedProperty().set(new EventHandler<MouseEvent>() {
        public void handle(MouseEvent event){
            onePath.setStrokeWidth(0);
            if (onePath.getElements().size() > 1) {
                pathTransition.stop();
                pathTransition.playFromStart();
            }
        }
    });
```

2-4. Manipulating Layout via Grids

Problem

You want to create a nice-looking form type user interface using grid type layout.

Solution

Create a simple form designer application to manipulate the user interface dynamically using the JavaFX's `javafx.scene.layout.GridPane`. The form designer application will have the following features:

- It will toggle the display of the Grid layout's grid lines for debugging.
- It will adjust the top padding of the `GridPane`.
- It will adjust the left padding of the `GridPane`.
- It will adjust the horizontal gap between cells in the `GridPane`.
- It will adjust the vertical gap between cells in the `GridPane`.
- It will align controls within cells horizontally.
- It will align controls within cells vertically.

The following code is the main launching point for the form designer application:

```
/**
 * Manipulating Layout Via Grids
 * @author cdea
 */
public class ManipulatingLayoutViaGrids extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chapter 2-4 Manipulating Layout via Grids ");
        Group root = new Group();
        Scene scene = new Scene(root, 640, 480, Color.WHITE);

        // Left and right split pane
        SplitPane splitPane = new SplitPane();
        splitPane.prefWidthProperty().bind(scene.widthProperty());
        splitPane.prefHeightProperty().bind(scene.heightProperty());

        // Form on the right
        GridPane rightGridPane = new MyForm();

        GridPane leftGridPane = new GridPaneControlPanel(rightGridPane);

        VBox leftArea = new VBox(10);
        leftArea.getChildren().add(leftGridPane);
        HBox hbox = new HBox();
        hbox.getChildren().add(splitPane);
        root.getChildren().add(hbox);
        splitPane.getItems().addAll(leftArea, rightGridPane);
```

```
        primaryStage.setScene(scene);

        primaryStage.show();
    }

}
```

When the form designer application is launched, the target form to be manipulated will be shown to the right side of the window's split pane. Shown following is the code of a simple grid-like form class that extends from `GridPane` that will be manipulated by the form designer application:

```
/**
 * MyForm is a form to be manipulated by the user.
 * @author cdea
 */
public class MyForm extends GridPane{
    public MyForm() {

        setPadding(new Insets(5));
        setHgap(5);
        setVgap(5);

        Label fNameLbl = new Label("First Name");
        TextField fNameFld = new TextField();
        Label lNameLbl = new Label("Last Name");
        TextField lNameFld = new TextField();
        Label ageLbl = new Label("Age");
        TextField ageFld = new TextField();

        Button saveButt = new Button("Save");

        // First name label
        GridPane.setHalignment(fNameLbl, HPos.RIGHT);
        add(fNameLbl, 0, 0);

        // Last name label
        GridPane.setHalignment(lNameLbl, HPos.RIGHT);
        add(lNameLbl, 0, 1);

        // Age label
        GridPane.setHalignment(ageLbl, HPos.RIGHT);
        add(ageLbl, 0, 2);

        // First name field
        GridPane.setHalignment(fNameFld, HPos.LEFT);
        add(fNameFld, 1, 0);
```

```
        // Last name field
        GridPane.setHalignment(lNameFld, HPos.LEFT);
        add(lNameFld, 1, 1);

        // Age Field
        GridPane.setHalignment(ageFld, HPos.RIGHT);
        add(ageFld, 1, 2);

        // Save button
        GridPane.setHalignment(saveButt, HPos.RIGHT);
        add(saveButt, 1, 3);

    }
}
```

When the form designer application is launched, the grid property control panel will be shown to the left side of the window's split pane. The property control panel will allow a user to manipulate the target form's grid pane attributes dynamically. The following code represents the grid property control panel that will manipulate a target grid pane's properties:

```
/** GridPaneControlPanel represents the left area of the split pane
 * allowing the user to manipulate the GridPane on the right.
 *
 * Manipulating Layout Via Grids
 * @author cdea
 */
public class GridPaneControlPanel extends GridPane{
    public GridPaneControlPanel(final GridPane targetGridPane) {
        super();


        setPadding(new Insets(5));
        setHgap(5);
        setVgap(5);

        // Setting Grid lines
        Label gridLinesLbl = new Label("Grid Lines");
        final ToggleButton gridLinesToggle = new ToggleButton("Off");
        gridLinesToggle.selectedProperty().addListener(new ChangeListener<Boolean>(){
            public void changed(ObservableValue<? extends Boolean> ov, Boolean oldValue,
Boolean newVal) {
                targetGridPane.setGridLinesVisible(newVal);
                gridLinesToggle.setText(newVal ? "On" : "Off");
            }
        });

        // toggle grid lines label
        GridPane.setHalignment(gridLinesLbl, HPos.RIGHT);
        add(gridLinesLbl, 0, 0);
```

```
        // toggle grid lines
        GridPane.setHalignment(gridLinesToggle, HPos.LEFT);
        add(gridLinesToggle, 1, 0);

        // Setting padding [top]
        Label gridPaddingLbl = new Label("Top Padding");

        final Slider gridPaddingSlider = SliderBuilder.create()
                .min(0)
                .max(100)
                .value(5)
                .showTickLabels(true)
                .showTickMarks(true)
                .minorTickCount(1)
                .blockIncrement(5)
                .build();
        gridPaddingSlider.valueProperty().addListener(new ChangeListener<Number>() {
            public void changed(ObservableValue<? extends Number> ov, Number oldVal, Number
newVal) {
                double top = targetGridPane.getInsets().getTop();
                double right = targetGridPane.getInsets().getRight();
                double bottom = targetGridPane.getInsets().getBottom();
                double left = targetGridPane.getInsets().getLeft();

                Insets newInsets = new Insets((double) newVal, right, bottom, left);

                targetGridPane.setPadding(newInsets);
            }
        });


        // padding adjustment label
        GridPane.setHalignment(gridPaddingLbl, HPos.RIGHT);
        add(gridPaddingLbl, 0, 1);

       // padding adjustment slider
       GridPane.setHalignment(gridPaddingSlider, HPos.LEFT);
       add(gridPaddingSlider, 1, 1);

        // Setting padding [top]
        Label gridPaddingLeftLbl = new Label("Left Padding");

        final Slider gridPaddingLeftSlider = SliderBuilder.create()
                .min(0)
                .max(100)
                .value(5)
                .showTickLabels(true)
                .showTickMarks(true)
                .minorTickCount(1)
                .blockIncrement(5)
                .build();
        gridPaddingLeftSlider.valueProperty().addListener(new ChangeListener<Number>() {
```

```
            public void changed(ObservableValue<? extends Number> ov, Number oldVal, Number
newVal) {
                double top = targetGridPane.getInsets().getTop();
                double right = targetGridPane.getInsets().getRight();
                double bottom = targetGridPane.getInsets().getBottom();
                double left = targetGridPane.getInsets().getLeft();

                Insets newInsets = new Insets(top, right, bottom, (double) newVal);

                targetGridPane.setPadding(newInsets);
            }
        });


        // padding adjustment label
        GridPane.setHalignment(gridPaddingLeftLbl, HPos.RIGHT);
        add(gridPaddingLeftLbl, 0, 2);

        // padding adjustment slider
        GridPane.setHalignment(gridPaddingLeftSlider, HPos.LEFT);
        add(gridPaddingLeftSlider, 1, 2);

        // Horizontal gap
        Label gridHGapLbl = new Label("Horizontal Gap");

        final Slider gridHGapSlider = SliderBuilder.create()
                .min(0)
                .max(100)
                .value(5)
                .showTickLabels(true)
                .showTickMarks(true)
                .minorTickCount(1)
                .blockIncrement(5)
                .build();
        gridHGapSlider.valueProperty().addListener(new ChangeListener<Number>() {
            public void changed(ObservableValue<? extends Number> ov, Number oldVal, Number
newVal) {
                targetGridPane.setHgap((double) newVal);
            }
        });


        // hgap label
        GridPane.setHalignment(gridHGapLbl, HPos.RIGHT);
        add(gridHGapLbl, 0, 3);

       // hgap slider
       GridPane.setHalignment(gridHGapSlider, HPos.LEFT);
       add(gridHGapSlider, 1, 3);

        // Vertical gap
        Label gridVGapLbl = new Label("Vertical Gap");
```

```
        final Slider gridVGapSlider = SliderBuilder.create()
                .min(0)
                .max(100)
                .value(5)
                .showTickLabels(true)
                .showTickMarks(true)
                .minorTickCount(1)
                .blockIncrement(5)
                .build();
        gridVGapSlider.valueProperty().addListener(new ChangeListener<Number>() {
            public void changed(ObservableValue<? extends Number> ov, Number oldVal, Number
newVal) {
                targetGridPane.setVgap((double) newVal);
            }
        });


        // vgap label
        GridPane.setHalignment(gridVGapLbl, HPos.RIGHT);
        add(gridVGapLbl, 0, 4);

        // vgap slider
        GridPane.setHalignment(gridVGapSlider, HPos.LEFT);
        add(gridVGapSlider, 1, 4);

        // Cell Column
        Label cellCol = new Label("Cell Column");
        final TextField cellColFld = new TextField("0");

        // cell Column label
        GridPane.setHalignment(cellCol, HPos.RIGHT);
        add(cellCol, 0, 5);

        // cell Column field
        GridPane.setHalignment(cellColFld, HPos.LEFT);
        add(cellColFld, 1, 5);

        // Cell Row
        Label cellRowLbl = new Label("Cell Row");
        final TextField cellRowFld = new TextField("0");

        // cell Row label
        GridPane.setHalignment(cellRowLbl, HPos.RIGHT);
        add(cellRowLbl, 0, 6);

        // cell Row field
        GridPane.setHalignment(cellRowFld, HPos.LEFT);
        add(cellRowFld, 1, 6);

        // Horizontal Alignment
        Label hAlignLbl = new Label("Horiz. Align");
        final ChoiceBox hAlignFld = new ChoiceBox(FXCollections.observableArrayList(
```

```
            "CENTER", "LEFT", "RIGHT")
        );
        hAlignFld.getSelectionModel().select("LEFT");

        // cell Row label
        GridPane.setHalignment(hAlignLbl, HPos.RIGHT);
        add(hAlignLbl, 0, 7);

        // cell Row field
        GridPane.setHalignment(hAlignFld, HPos.LEFT);
        add(hAlignFld, 1, 7);

        // Vertical Alignment
        Label vAlignLbl = new Label("Vert. Align");
        final ChoiceBox vAlignFld = new ChoiceBox(FXCollections.observableArrayList(
            "BASELINE", "BOTTOM", "CENTER", "TOP")
        );
        vAlignFld.getSelectionModel().select("TOP");
        // cell Row label
        GridPane.setHalignment(vAlignLbl, HPos.RIGHT);
        add(vAlignLbl, 0, 8);

        // cell Row field
        GridPane.setHalignment(vAlignFld, HPos.LEFT);
        add(vAlignFld, 1, 8);

        // Vertical Alignment
        Label cellApplyLbl = new Label("Cell Constraint");
        final Button cellApplyButton = new Button("Apply");
        cellApplyButton.setOnAction(new EventHandler<ActionEvent>() {

            public void handle(ActionEvent event) {

                for (Node child:targetGridPane.getChildren()) {

                    int targetColIndx = 0;
                    int targetRowIndx = 0;
                    try {
                        targetColIndx = Integer.parseInt(cellColFld.getText());
                        targetRowIndx = Integer.parseInt(cellRowFld.getText());
                    } catch (Exception e) {

                    }
                    System.out.println("child = " + child.getClass().getSimpleName());
                    int col = GridPane.getColumnIndex(child);
                    int row = GridPane.getRowIndex(child);
                    if (col == targetColIndx && row == targetRowIndx) {
                        GridPane.setHalignment(child,
HPos.valueOf(hAlignFld.getSelectionModel().getSelectedItem().toString()));
                        GridPane.setValignment(child,
VPos.valueOf(vAlignFld.getSelectionModel().getSelectedItem().toString()));
                    }
```

```
                }
            }
        });

        // cell Row label
        GridPane.setHalignment(cellApplyLbl, HPos.RIGHT);
        add(cellApplyLbl, 0, 9);

        // cell Row field
        GridPane.setHalignment(cellApplyButton, HPos.LEFT);
        add(cellApplyButton, 1, 9);

    }
}
```

Figure 2-7 shows a form designer application with the `GridPane` property control panel on the left and the target form on the right.

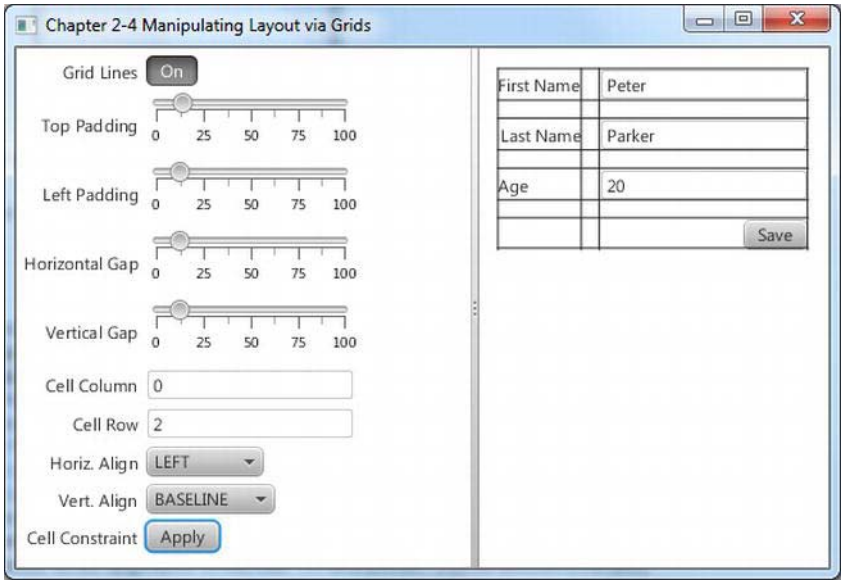

Figure 2-7. *Manipulating layout via grids*

How It Works

The form designer application will allow the user to adjust properties dynamically using the `GridPane` property control panel to the left. While adjusting properties from the left control panel the target form on the right side will be manipulated dynamically. When creating a simple form designer application you will be binding controls to various properties onto the target form (`GridPane`). This designer application is basically broken out into three classes: `ManipulatingLayoutViaGrids`, `MyForm`, and `GridPaneControlPanel`. First the `ManipulatingLayoutViaGrids` class is the main application to be

launched. Second, MyForm is the target form that will be manipulated. Last, GridPaneControlPanel is the grid property control panel that has UI controls bound to the targets form's grid pane properties.

You begin by creating the main launching point for the application (ManipulatingLayoutViaGrids). This class is responsible for creating a split pane (SplitPane) that sets up the target form to the right and instantiates a GridPaneControlPanel to be displayed to the left. To instantiate a GridPaneControlPanel you must pass in the target form you want to manipulate into the constructor. I will discuss this further, but suffice it to say that the GridPaneControlPanel constructor will wire up its controls to properties on the target form.

Next, you will simply create a dummy form that I've called MyForm. This form will be your target form that the property control panel will be manipulating. Here, you will notice the MyForm extends GridPane. In the MyForm's constructor you will create and add controls to be put into the form (GridPane). To learn more about the GridPane refer to recipe 1-8. The following code is a target form to be manipulated by the form designer application:

```
/**
 * MyForm is a form to be manipulated by the user.
 * @author cdea
 */
public class MyForm extends GridPane{
    public MyForm() {

        setPadding(new Insets(5));
        setHgap(5);
        setVgap(5);

        Label fNameLbl = new Label("First Name");
        TextField fNameFld = new TextField();
        Label lNameLbl = new Label("Last Name");
        TextField lNameFld = new TextField();
        Label ageLbl = new Label("Age");
        TextField ageFld = new TextField();

        Button saveButt = new Button("Save");

        // First name label
        GridPane.setHalignment(fNameLbl, HPos.RIGHT);
        add(fNameLbl, 0, 0);
    //… The rest of the form code
```

To manipulate the target form you will need to create a grid property control panel (GridPaneControlPanel). This class is responsible for binding the target form's grid pane properties to UI controls that allow the user to adjust values using the keyboard and mouse. As you learned earlier in recipe 1-10, you can bind values with JavaFX Properties. But instead of binding values directly, you can also be notified when a property has changed.

Another feature that you can apply to properties is that you can add change listeners. JavaFXjavafx.beans.value.ChangeListeners is similar to Java Swing's property change support (java.beans.PropertyChangeListener). Similarly, when a bean's property value has changed you will want to be notified of that change. Change listeners are designed to intercept the change by making the old and new value available to the developer. You will start by creating a JavaFXchange listener for the toggle button to turn gridlines on or off. When a user interacts with the toggle button, the change listener will simply update the target's grid pane's gridlinesVisible property. Because a toggle button's (ToggleButton) selected property is a Boolean value, you will instantiate a ChangeListener class with its

formal type parameter as Boolean. You'll also notice the overridden method changed() where its inbound parameters will match the generics formal type parameter specified when instantiating a ChangeListener<Boolean>. When a property change event occurs, the change listener will invoke setGridLinesVisible() on the target grid pane with the new value and update the toggle button's text. The following code snippet shows a ChangeListener<Boolean> added to a ToggleButton:

```
gridLinesToggle.selectedProperty().addListener(
new ChangeListener<Boolean>(){
        public void changed(ObservableValue<? extends Boolean> ov, Boolean oldValue, Boolean
newVal) {
targetGridPane.setGridLinesVisible(newVal);
gridLinesToggle.setText(newVal ? "On" : "Off");
        }
});
```

Next, you will be applying a change listener to a slider control that allows the user to adjust the target grid pane's top padding. To create a change listener for a slider you will be instantiating a ChangeListener<Number>. Again, you will be overriding the changed() method with a signature the same as its formal type parameter Number. When a change occurs, the slider's value will be used to create an Insets object that becomes the new padding for the target grid pane. Shown here is the change listener for the top padding and slider control:

```
    gridPaddingSlider.valueProperty().addListener(new ChangeListener<Number>() {
        public void changed(ObservableValue<? extends Number> ov, Number oldVal, Number
    newVal) {
            double top = targetGridPane.getInsets().getTop();
            double right = targetGridPane.getInsets().getRight();
            double bottom = targetGridPane.getInsets().getBottom();
            double left = targetGridPane.getInsets().getLeft();

            Insets newInsets = new Insets((double) newVal, right, bottom, left);

            targetGridPane.setPadding(newInsets);
        }
    });
```

Because the implementation of the other slider controls that handle left padding, horizontal gap, and vertical gap are virtually identical to the top padding slider control mentioned previously, you can fast forward to cell constraints controls.

The last bits of grid control panel properties that you want to manipulate are the target grid pane's cell constraints. For brevity I only allow the user to set a component's alignment inside of a cell of a GridPane. To see more properties to modify, refer to the Javadoc on javafx.scene.layout.GridPane. Figure 2-8 depicts the cell constraint settings for individual cells. An example is to left-justify the label Age on the target grid pane. Because cells are zero relative, you will enter **0** in the Cell Column field and two into the Cell Row field. Next, you will select the drop-down box Horiz. Align to LEFT. Once satisfied with the settings, click Apply. Figure 2-9 shows the Age label control left-aligned horizontally. To implement this, you will create an EventHandler<ActionEvent> for the apply button's onAction attribute by calling its setOnAction() method. Again when creating EventHandlers you will be overriding the handle() method. Inside of the handle() method you will basically iterate over all node children owned by the target grid pane to determine whether it is the specified cell. Once the specified cell and child node is determined the alignment will be applied. The following code is an EventHandler to apply cell constraint when the apply button is pressed:

```
final Button cellApplyButton = new Button("Apply");
cellApplyButton.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent event) {

        for (Node child:targetGridPane.getChildren()) {

            int targetColIndx = 0;
            int targetRowIndx = 0;
            try {
                targetColIndx = Integer.parseInt(cellColFld.getText());
                targetRowIndx = Integer.parseInt(cellRowFld.getText());
            } catch (Exception e) {

            }
            System.out.println("child = " + child.getClass().getSimpleName());
            int col = GridPane.getColumnIndex(child);
            int row = GridPane.getRowIndex(child);
            if (col == targetColIndx && row == targetRowIndx) {
                GridPane.setHalignment(child,
HPos.valueOf(hAlignFld.getSelectionModel().getSelectedItem().toString()));
                GridPane.setValignment(child,
VPos.valueOf(vAlignFld.getSelectionModel().getSelectedItem().toString()));
            }
        }

    }
});
```

Figure 2-8 depicts the cell constraint grid control panel section that left-aligns the control at cell column zero and cell row 2.

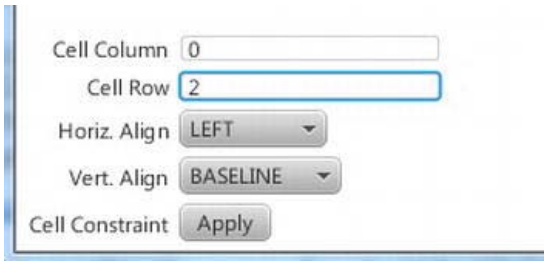

Figure 2-8. *Cell constraints*

Figure 2-9 depicts the target grid pane with the grid lines turned on along with the Age label left-aligned horizontally at cell column 0 and cell row 2.

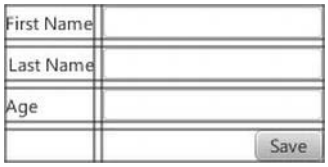

Figure 2-9. *Target grid pane*

2-5. Enhancing with CSS

Problem

You want to change the Look 'N' Feel of the GUI interface.

Solution

Use JavaFX's CSS styling to be applied on graph nodes. The following code demonstrates using CSS styling on graph nodes. The code creates four themes: Caspian, Control Style 1, Control Style 2, and Sky. Each theme is defined using CSS and affects the Look 'N' Feel of a dialog box. Following the code, you can see the two different renditions of the dialog box:

```
package javafx2introbyexample.chapter2.recipe2_05;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.control.SplitPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

/**
 * Enhancing with CSS
 * @author cdea
 */
public class EnhancingWithCss extends Application {
```

```
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {

            primaryStage.setTitle("Chapter 2-5 Enhancing with CSS ");
            Group root = new Group();
            final Scene scene = new Scene(root, 640, 480, Color.BLACK);

            MenuBar menuBar = new MenuBar();
            Menu menu = new Menu("Look 'N' Feel");

            // default caspian look n feel
            ObservableList<String> caspian = FXCollections.observableArrayList();
            caspian.addAll(scene.getStylesheets());
            MenuItem caspianLnf = new MenuItem("Caspian");
            caspianLnf.setOnAction(skinForm(caspian, scene));
            menu.getItems().add(caspianLnf);

            menu.getItems().add(createMenuItem("Control Style 1", "controlStyle1.css",
scene));
            menu.getItems().add(createMenuItem("Control Style 2", "controlStyle2.css",
scene));
            menu.getItems().add(createMenuItem("Sky", "sky.css", scene));

menuBar.getMenus().add(menu);
            // stretch menu
menuBar.prefWidthProperty().bind(primaryStage.widthProperty());

            // Left and right split pane
            SplitPane splitPane = new SplitPane();
            splitPane.prefWidthProperty().bind(scene.widthProperty());
            splitPane.prefHeightProperty().bind(scene.heightProperty());

            // Form on the right
            GridPane rightGridPane = new MyForm();

            GridPane leftGridPane = new GridPaneControlPanel(rightGridPane);
            VBox leftArea = new VBox(10);
            leftArea.getChildren().add(leftGridPane);


            HBox hbox = new HBox();
            hbox.getChildren().add(splitPane);
            VBox vbox = new VBox();
            vbox.getChildren().add(menuBar);
            vbox.getChildren().add(hbox);
```

```
            root.getChildren().add(vbox);
            splitPane.getItems().addAll(leftArea, rightGridPane);

            primaryStage.setScene(scene);

            primaryStage.show();

    }

    protected final MenuItem createMenuItem(String label, String css, final Scene scene){
        MenuItem menuItem = new MenuItem(label);
        ObservableList<String> cssStyle = loadSkin(css);
        menuItem.setOnAction(skinForm(cssStyle, scene));
        return menuItem;
    }

    protected final ObservableList<String> loadSkin(String cssFileName) {
        ObservableList<String> cssStyle = FXCollections.observableArrayList();
        cssStyle.addAll(getClass().getResource(cssFileName).toExternalForm());
        return cssStyle;
    }

    protected final EventHandler<ActionEvent> skinForm(final ObservableList<String> cssStyle,
final Scene scene) {
        return new EventHandler<ActionEvent>(){
            public void handle(ActionEvent event) {
                scene.getStylesheets().clear();
                scene.getStylesheets().addAll(cssStyle);
            }
        };
    }

}
```

Figure 2-10 depicts the standard JavaFX Caspian Look ‘n’ Feel (theme).

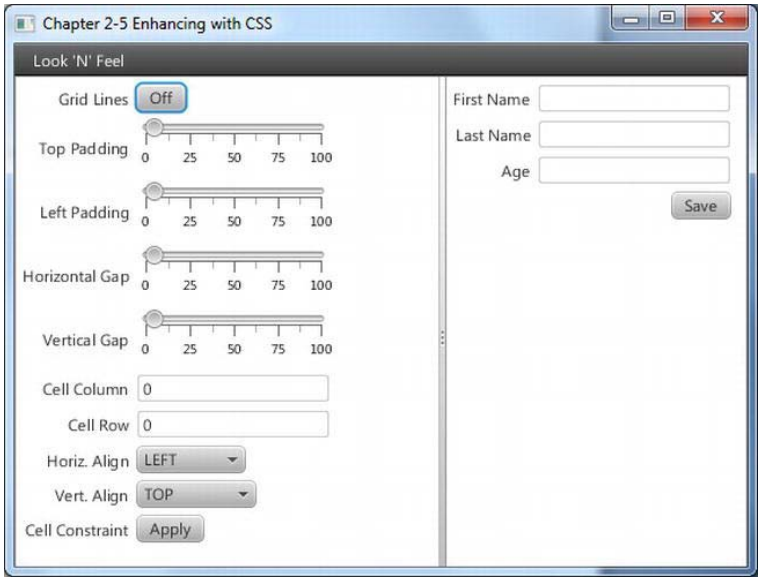

Figure 2-10. *Caspian Look 'N' Feel*

Figure 2-11 depicts the Sky Look 'N' Feel (theme).

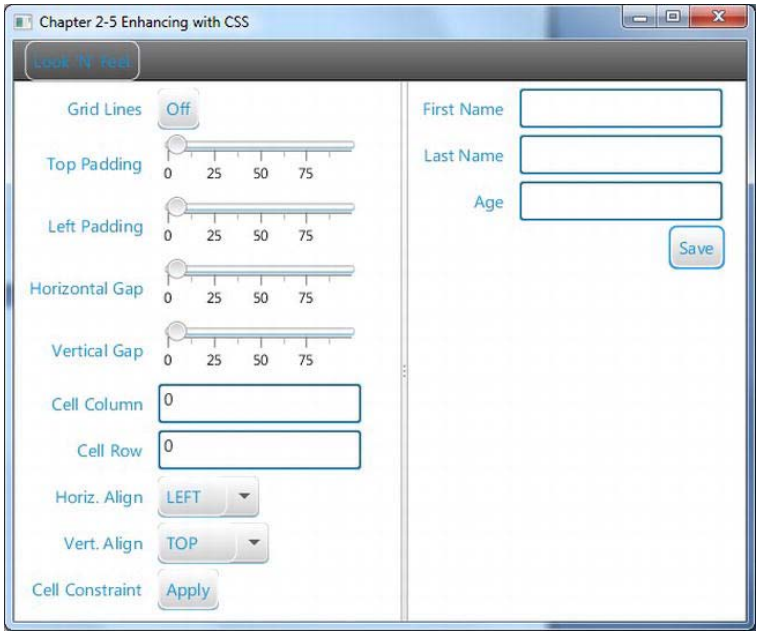

Figure 2-11. *Sky Look 'N' Feel*

How It Works

JavaFX has the capability to apply CSS styles onto the Scene graph and its nodes very much like browsers applying CSS styles onto elements in an HTML document object model (DOM). In this recipe you will be skinning a user interface using JavaFX styling attributes. I basically used the recipe's UI to apply the various Look 'n' Feels. To showcase the available skins, a menu selection allows the user to choose the Look 'N' Feel to apply to the UI.

Before discussing CSS styling properties, I want to show you how to load the CSS styles to be applied to a JavaFX application. You will first need to create menu items to allow the user to choose the preferred Look 'N' Feel. When creating a menu item you will create a convenience method to build a menu item that would load the specified CSS and an `EventHandler` action to apply the chosen CSS style onto the current UI. To add the Caspian theme as a menu item you will notice that no resources are needed to be loaded because it is JavaFX's current Look 'n' Feel. Shown here is adding a menu item containing the Caspian Look 'N' Feel CSS style that can be applied to the current UI:

```
MenuItem caspianLnf = new MenuItem("Caspian");
caspianLnf.setOnAction(skinForm(caspian, scene));
```

Shown here is adding a menu item containing the sky Look 'N' Feel CSS style ready to be applied to the current UI:

```
MenuBar menuBar = new MenuBar();
Menu menu = new Menu("Look 'N' Feel");
menu.getItems().add(createMenuItem("Sky", "sky.css", scene));
```

Calling the `createMenuItem()` method will also call another convenience method to load the CSS file called `loadSkin()`. It will also set the menu items `onAction` attribute with an appropriate `EventHandler` by calling the `skinForm()` method. To recap, the `loadSkin` is responsible for loading the CSS file, and the `skinForm()` method's job is to apply the skin onto the UI application. Shown here are the convenience methods to build menu items that apply CSS styles onto a UI application:

```
protected final MenuItem createMenuItem(String label, String css, final Scene scene){
    MenuItem menuItem = new MenuItem(label);
    ObservableList<String> cssStyle = loadSkin(css);
    menuItem.setOnAction(skinForm(cssStyle, scene));
    return menuItem;
}

protected final ObservableList<String> loadSkin(String cssFileName) {
    ObservableList<String> cssStyle = FXCollections.observableArrayList();
    cssStyle.addAll(getClass().getResource(cssFileName).toExternalForm());
    return cssStyle;
}

    protected final EventHandler<ActionEvent> skinForm(final ObservableList<String> cssStyle,
final Scene scene) {
        return new EventHandler<ActionEvent>(){
            public void handle(ActionEvent event) {
```

```
                scene.getStylesheets().clear();
                scene.getStylesheets().addAll(cssStyle);
            }
        };
    }
```

■ **Note** To run this recipe example, make sure the CSS files are located in the compiled classes area. Resource files can be loaded easily when placed in the same directory (package) as the compiled class file that is loading them. The CSS files are co-located with this code example file. In NetBeans, you can select Clean and build project or you can copy files to your classes build area.

Now, that you know how to load CSS styles, let's talk about the JavaFX CSS selectors and styling properties. Like CSS style sheets, there are selectors or style classes associated with `Node` objects in the Scene graph. All Scene graph nodes have a method called `setStyle()` to apply styling properties that could potentially change the node's background color, border, stroke, and so on. Because all graph nodes extend from the `Node` class, derived classes will be able to inherit the same styling properties. Knowing the inheritance hierarchy of node types is very important because the type of node will determine the types of styling properties you can affect. For instance a Rectangle extends from Shape, which extends from Node. The inheritance does not include `-fx-border-style`, which is the part of nodes that extends from `Region`. Based on the type of node there are limitations to what styles you are able to set. To see a full listing of all style selectors refer to the JavaFX CSS Reference Guide: `http://download.oracle.com/docs/cd/E17802_01/javafx/javafx/1.3/docs/api/javafx.scene/doc-files/cssref.html`.

All JavaFX styling properties will be prefixed with `-fx-`. For example, all `Nodes` have the styling property to affect its opacity the attribute used is `-fx-opacity`. Following are selectors to style JavaFX `javafx.scene.control.Labels` and `javafx.scene.control.Buttons`:

```
.label {
    -fx-text-fill: rgba(17, 145, 213);
    -fx-border-color: rgba(255, 255, 255, .80);
    -fx-border-radius: 8;
    -fx-padding: 6 6 6 6;
    -fx-font: bold italic 20pt "LucidaBrightDemiBold";

}
.button{
    -fx-text-fill: rgba(17, 145, 213);
    -fx-border-color: rgba(255, 255, 255, .80);
    -fx-border-radius: 8;
    -fx-padding: 6 6 6 6;
    -fx-font: bold italic 20pt "LucidaBrightDemiBold";

}
```

CHAPTER 3

Media with JavaFX

JavaFX provides a media-rich API capable of playing audio and video. The Media API allows developers to incorporate audio and video into their RIAs. One of the main benefits of the Media API is its cross-platform abilities when distributing media content via the Web. With a range of devices (tablet, music player, TV, and so on) that need to play multimedia content, the need for a cross-platform API is essential.

Imagine a not-so-distant future where your TV or wall is capable of interacting with you in ways that you've never dreamed possible. For instance, while viewing a movie you could select items or clothing used in the movie to be immediately purchased, all from the comfort of your home. With this future in mind, developers seek to enhance the interactive qualities of their media-based applications.

In this chapter you will learn how to play audio and video in an interactive way. Find your seats for Act III of JavaFX as audio and video take center stage (as depicted in Figure 3-1).

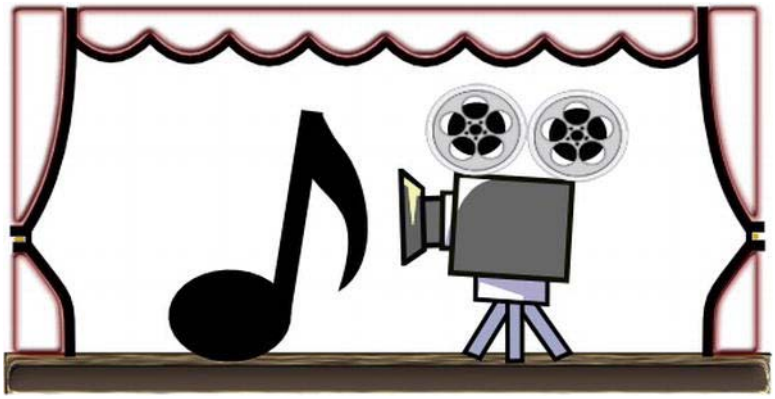

Figure 3-1. *Audio and video*

3-1. Playing Audio

Problem

You want to listen to music and become entertained with a graphical visualization.

Solution

Create an MP3 player by utilizing the following classes:

- `javafx.scene.media.Media`
- `javafx.scene.media.MediaPlayer`
- `javafx.scene.media.AudioSpectrumListener`

The following source code is an implementation a of simple MP3 player:

```
package javafx2introbyexample.chapter3.recipe3_01;

import java.io.File;
import java.util.Random;
import javafx.application.*;
import javafx.event.EventHandler;
import javafx.geometry.Point2D;
import javafx.scene.*;
import javafx.scene.input.*;
import javafx.scene.media.*;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;
import javafx.scene.text.Text;
import javafx.stage.*;

/**
 * Playing Audio
 * @author cdea
 */
public class PlayingAudio extends Application {
    private MediaPlayer mediaPlayer;
    private Point2D anchorPt;
    private Point2D previousLocation;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(final Stage primaryStage) {
        primaryStage.setTitle("Chapter 3-1 Playing Audio");
        primaryStage.centerOnScreen();
        primaryStage.initStyle(StageStyle.TRANSPARENT);

        Group root = new Group();
        Scene scene = new Scene(root, 551, 270, Color.rgb(0, 0, 0, 0));

        // application area
```

```
        Rectangle applicationArea = RectangleBuilder.create()
                .arcWidth(20)
                .arcHeight(20)
                .fill(Color.rgb(0, 0, 0, .80))
                .x(0)
                .y(0)
                .strokeWidth(2)
                .stroke(Color.rgb(255, 255, 255, .70))
                .build();
        root.getChildren().add(applicationArea);
        applicationArea.widthProperty().bind(scene.widthProperty());
        applicationArea.heightProperty().bind(scene.heightProperty());

        final Group phaseNodes = new Group();
        root.getChildren().add(phaseNodes);


        // starting initial anchor point
        scene.setOnMousePressed(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent event){
                anchorPt = new Point2D(event.getScreenX(), event.getScreenY());
            }
        });

        // dragging the entire stage
        scene.setOnMouseDragged(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent event){
                if (anchorPt != null && previousLocation != null) {
                    primaryStage.setX(previousLocation.getX() + event.getScreenX() -
anchorPt.getX());
                    primaryStage.setY(previousLocation.getY() + event.getScreenY() -
anchorPt.getY());
                }
            }
        });

        // set the current location
        scene.setOnMouseReleased(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent event){
                previousLocation = new Point2D(primaryStage.getX(), primaryStage.getY());
            }
        });



        // Dragging over surface
        scene.setOnDragOver(new EventHandler<DragEvent>() {
            @Override
            public void handle(DragEvent event) {
                Dragboard db = event.getDragboard();
                if (db.hasFiles()) {
                    event.acceptTransferModes(TransferMode.COPY);
```

```
                } else {
                    event.consume();
                }
            }
        });


        // Dropping over surface
        scene.setOnDragDropped(new EventHandler<DragEvent>() {

            @Override
            public void handle(DragEvent event) {
                Dragboard db = event.getDragboard();
                boolean success = false;
                if (db.hasFiles()) {
                    success = true;
                    String filePath = null;
                    for (File file:db.getFiles()) {
                        filePath = file.getAbsolutePath();
                        System.out.println(filePath);
                    }
                    // play file
                    Media media = new Media(new File(filePath).toURI().toString());

                    if (mediaPlayer != null) {
                        mediaPlayer.stop();
                    }

                    mediaPlayer = MediaPlayerBuilder.create()
                            .media(media)
                            .audioSpectrumListener(new AudioSpectrumListener() {
                        @Override
                        public void spectrumDataUpdate(double timestamp, double duration,
    float[] magnitudes, float[] phases) {
                            phaseNodes.getChildren().clear();
                            int i = 0;
                            int x = 10;
                            int y = 150;
                            final Random rand = new Random(System.currentTimeMillis());
                            for(float phase:phases) {
                                int red = rand.nextInt(255);
                                int green = rand.nextInt(255);
                                int blue = rand.nextInt(255);

                                Circle circle = new Circle(10);
                                circle.setCenterX(x + i);
                                circle.setCenterY(y + (phase * 100));
                                circle.setFill(Color.rgb(red, green, blue, .70));
                                phaseNodes.getChildren().add(circle);
                                i+=5;
                            }
                        }
```

```
            })
            .build();

            mediaPlayer.setOnReady(new Runnable() {
                @Override
                public void run() {
                    mediaPlayer.play();
                }
            });
        }

        event.setDropCompleted(success);
        event.consume();
    }
}); // end of setOnDragDropped


// create slide controls
final Group buttonGroup = new Group();

// rounded rect
Rectangle buttonArea = RectangleBuilder.create()
        .arcWidth(15)
        .arcHeight(20)
        .fill(new Color(0, 0, 0, .55))
        .x(0)
        .y(0)
        .width(60)
        .height(30)
        .stroke(Color.rgb(255, 255, 255, .70))
        .build();

buttonGroup.getChildren().add(buttonArea);
// stop audio control
Node stopButton = RectangleBuilder.create()
        .arcWidth(5)
        .arcHeight(5)
        .fill(Color.rgb(255, 255, 255, .80))
        .x(0)
        .y(0)
        .width(10)
        .height(10)
        .translateX(15)
        .translateY(10)
        .stroke(Color.rgb(255, 255, 255, .70))
        .build();

stopButton.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
       if (mediaPlayer!= null) {
        mediaPlayer.stop();
       }
```

```
    }
});
buttonGroup.getChildren().add(stopButton);

// play control
final Node playButton = ArcBuilder.create()
        .type(ArcType.ROUND)
        .centerX(12)
        .centerY(16)
        .radiusX(15)
        .radiusY(15)
        .startAngle(180-30)
        .length(60)
        .fill(new Color(1,1,1, .90))
        .translateX(40)
        .build();
playButton.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        mediaPlayer.play();
    }
});


// pause control
final Group pause = new Group();
final Node pauseButton = CircleBuilder.create()
        .centerX(12)
        .centerY(16)
        .radius(10)
        .stroke(new Color(1,1,1, .90))
        .translateX(30)
        .build();
final Node firstLine = LineBuilder.create()
        .startX(6)
        .startY(16 - 10)
        .endX(6)
        .endY(16 - 2)
        .strokeWidth(3)
        .translateX(34)
        .translateY(6)
        .stroke(new Color(1,1,1, .90))
        .build();

final Node secondLine = LineBuilder.create()
        .startX(6)
        .startY(16 - 10)
        .endX(6)
        .endY(16 - 2)
        .strokeWidth(3)
        .translateX(38)
        .translateY(6)
        .stroke(new Color(1,1,1, .90))
```

```
                .build();
        pause.getChildren().addAll(pauseButton, firstLine, secondLine);

        pause.setOnMousePressed(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent me) {
                if (mediaPlayer!=null) {
                    buttonGroup.getChildren().remove(pause);
                    buttonGroup.getChildren().add(playButton);
                    mediaPlayer.pause();
                }
            }
        });

        playButton.setOnMousePressed(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent me) {
                if (mediaPlayer != null) {
                    buttonGroup.getChildren().remove(playButton);
                    buttonGroup.getChildren().add(pause);
                    mediaPlayer.play();
                }
            }
        });

        buttonGroup.getChildren().add(pause);
        // move button group when scene is resized

buttonGroup.translateXProperty().bind(scene.widthProperty().subtract(buttonArea.getWidth() +
6));

buttonGroup.translateYProperty().bind(scene.heightProperty().subtract(buttonArea.getHeight() +
6));
        root.getChildren().add(buttonGroup);


        // close button
        final Group closeApp = new Group();
        Node closeButton = CircleBuilder.create()
                .centerX(5)
                .centerY(0)
                .radius(7)
                .fill(Color.rgb(255, 255, 255, .80))
                .build();
        Node closeXmark = new Text(2, 4, "X");
        closeApp.translateXProperty().bind(scene.widthProperty().subtract(15));
        closeApp.setTranslateY(10);
        closeApp.getChildren().addAll(closeButton, closeXmark);
        closeApp.setOnMouseClicked(new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                Platform.exit();
            }
        });
```

```
            root.getChildren().add(closeApp);

            primaryStage.setScene(scene);
            primaryStage.show();
            previousLocation = new Point2D(primaryStage.getX(), primaryStage.getY());

        }
}
```

Figure 3-2 shows a JavaFX MP3 player with visualizations.

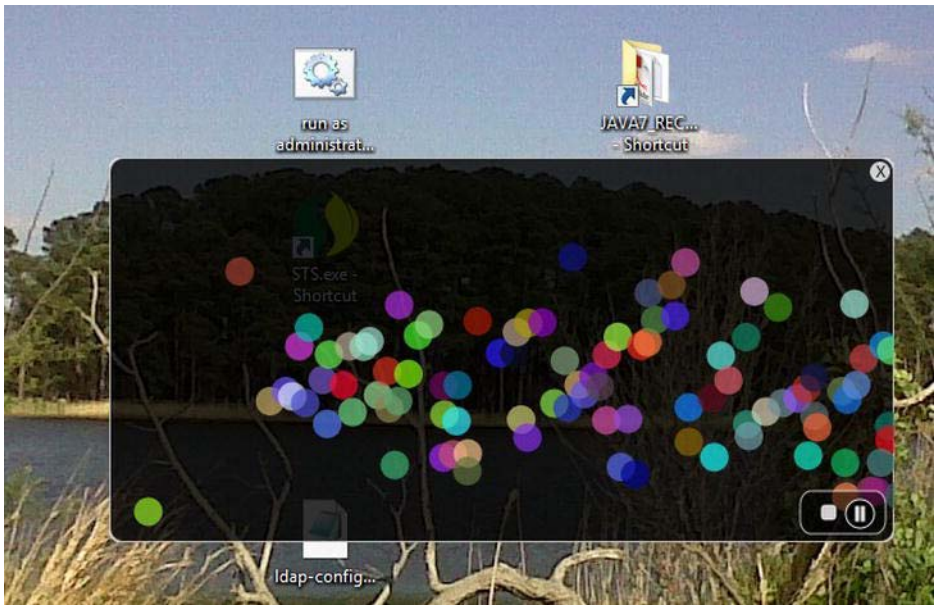

Figure 3-2. *JavaFX MP3 player*

How It Works

Before we get started, let's discuss the instructions on how to operate our MP3 player. A user will be able to drag and drop an audio file into the application area to be subsequently played. Located on the lower right of the application are buttons to stop, pause, and resume play of audio media. (The button controls are shown in Figure 3-2.) As the music is playing, the user will also notice randomly colored balls bouncing around to the music. Once the user is done with listening to music, he/she can quit the application by clicking the white rounded close button located in the upper-right corner.

It is similar to recipe 2-1, in which you learned how to use the drag-and-drop desktop metaphor to load files into a JavaFX application. Instead of image files, however, the user will be using audio files. To load audio files JavaFX currently supports the following file formats: `.mp3`, `.wav`, and `.aiff`.

Following the same look and feel, you will use the same style as recipe 12-1. In this recipe, I modified the button controls to resemble buttons, similar to many media player applications. When the pause button is pressed, it will pause the audio media from playing and toggle to the play button control,

thus allowing the user to resume. As an added bonus, the MP3 player will appear as an irregular shaped, semitransparent window without borders that can also be dragged around the desktop using the mouse. Now that you know how to operate the music player, let's walk through the code.

First, you will create instance variables that will maintain state information for the lifetime of the application. Table 3-1 describes all instance variables used in our music player application. The first variable is a reference to a media player (`MediaPlayer`) object that will be created in conjunction with a `Media` object containing an audio file. Next, you will create an `anchorPt` variable used to save the starting coordinate of a mouse press when the user begins to drag the window across the screen. When calculating the upper-left bounds of the application window during a mouse-dragged operation, the `previousLocation` variable will contain the previous window's screen X and Y coordinates.

Table 3-1 lists the MP3 player application's instance variables:

Table 3-1. *MP3 Player Application Instance Variables*

Variable	Data Type	Example	Description
`mediaPlayer`	`MediaPlayer`	n/a	A media player control that plays audio and video
`anchorPt`	`Point2D`	100,100	A coordinate where the user begins to drag the window
`previousLocation`	`Point2D`	0,0	The upper-left corner of the stage's previous coordinate; assists in dragging the window

In previous chapters relating to GUIs, you saw that GUI applications normally contain a title bar and windowed borders surrounding the `Scene`. Here, I wanted to raise the bar a little by showing you how to create irregularly shaped semitransparent windows, thus making things look more hip or modern. As you begin to create the media player, you'll notice in the `start()` method that we prepare the `Stage` object by initializing the style using `StageStyle.TRANSPARENT`. After we initialize the style to `StageStyle.TRANSPARENT`, the window will be undecorated, with the entire window area opaque set to zero (invisible). The following code shows how to create a transparent window without a title bar or windowed borders:

```
primaryStage.initStyle(StageStyle.TRANSPARENT);
```

With the invisible stage you will create a rounded rectangular region that will be the applications surface, or main content area. Next, you will notice the width and height of the rectangle bound to the scene object in case the window is resized. Because the window isn't going to be resized, the bind isn't necessary (it will be needed, however in recipe 3-2, when you get a chance to enlarge your video screen to take on a full screen mode).

After creating a black, semitransparent, rounded, rectangular area (`applicationArea`), you'll be creating a simple `Group` object to hold all the randomly colored `Circle` nodes that will show off graphical visualizations while the audio is being played. Later, you will see how the `phaseNodes` (`Group`) variable is updated based on sound information using an `AudioSpectrumListener`.

Next, you will be adding `EventHandler<MouseEvent>` instances to the `Scene` object to monitor mouse events as the user drags the window around the screen. The first event in this scenario is a mouse press, which will save the cursor's current (X, Y) coordinates to the variable `anchorPt`. The following code is adding an `EventHandler` to the mouse pressed property of the `Scene`:

```
// starting initial anchor point
```

```
        scene.setOnMousePressed(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent event){
                anchorPt = new Point2D(event.getScreenX(), event.getScreenY());
            }
        });
```

After implementing the mouse press event handler, you can create an `EventHandler` to the `Scene`'s mouse-dragged property. The mouse–dragged event handler will update and position the application window (`Stage`) dynamically, based on the previous window's location (upper-left corner) along with the `anchorPt` variable. Shown here is an event handler responsible for the mouse-dragged event on the `Scene` object:

```
        // dragging the entire stage
        scene.setOnMouseDragged(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent event){
                if (anchorPt != null && previousLocation != null) {
                    primaryStage.setX(previousLocation.getX() + event.getScreenX() -
anchorPt.getX());
                    primaryStage.setY(previousLocation.getY() + event.getScreenY() -
anchorPt.getY());
                }
            }
        });
```

You will want to handle the mouse-released event. Once the mouse is released, the event handler will update the `previousLocation` variable for subsequent mouse-dragged events to move the application window about the screen. The following code snippet updates the `previousLocation` variable:

```
        // set the current location
        scene.setOnMouseReleased(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent event){
                previousLocation = new Point2D(primaryStage.getX(), primaryStage.getY());
            }
        });
```

Next, you will be implementing the drag-and-drop scenario to load the audio file from the file system (File manager). When handling a drag-and-drop scenario, it is similar to recipe 2-1, in which you created an `EventHandler` to handle `DragEvents`. Instead of loading image files we will be loading audio files from the host file system. For brevity, I will simply mention the code lines of the drag-and-dropped event handler. Once the audio file is available, you will create a `Media` object by passing in the file as a `URI`. The following code snippet is how to create a `Media` object:

```
Media media = new Media(new File(filePath).toURI().toString());
```

Once you have created a `Media` object you will have to create an instance of a `MediaPlayer` in order to play the sound file. Both the `Media` and `MediaPlayer` objects are immutable, which is why new instances of each will be created every time the user drags a file into the application. Next, you will check the instance variable `mediaPlayer` for a previous instance to make sure it is stopped before creating a new `MediaPlayer` instance. The following code checks for a prior media player to be stopped:

```
if (mediaPlayer != null) {
```

```
    mediaPlayer.stop();
  }
```

So, here is where we create a `MediaPlayer` instance. For ease of coding you will be turning to the `MediaPlayer`'s builder class `MediaPlayerBuilder`. A `MediaPlayer` object is responsible for controlling the playing of media objects. Notice that a `MediaPlayer` will treat sound or video media the same in terms of playing, pausing, and stopping media. When creating a media player using the `MediaPlayerBuilder` class, you will be specifying the `media` and `audioSpectrumListener` attribute methods. Setting the autoPlay attribute to true will play the audio media immediately after it has been loaded. The last thing to specify on the `MediaPlayer` instance is an `AudioSpectrumListener`. So, what exactly is this type of listener, you say? Well, according to the Javadocs, it states that it is an observer receiving periodic updates of the audio spectrum. In layman's terms, it is the audio media's sound data such as volume and tempo, and so on. To create an instance of an `AudioSpectrumListener` you will create an inner class overriding the method `spectrumDataUpdate()`. Table 3-2 lists all inbound parameters for the audio spectrum listener's method. For more details refer to the Javadocs at `http://download.oracle.com/javafx/2.0/api/javafx/scene/media/AudioSpectrumListener.html`.

Table 3-2. *The AudioSpectrumListener's Method spectrumDataUpdate() Inbound Parameters*

Variable	Data Type	Example	Description
`timestamp`	`double`	`2.4261`	When the event occurred, in seconds
`duration`	`Double`	0.1	The duration of time (in seconds) the spectrum was computed
`magnitudes`	`float[]`	-50.474335	An array of float values representing each band's spectrum magnitude in decibels (nonpositive float value)
`phases`	`float[]`	1.2217305	An array of float values representing each band's phase

Here, you will be creating randomly colored circle nodes to be positioned and placed on the scene based on the variable phases (array of floats). To draw each colored circle, you will be incrementing the circle's center X by 5 pixels and adding the circle's center Y with each phase value multiplied by 100. Shown here is the code snippet that plots each randomly colored circle:

```
        circle.setCenterX(x + i);
        circle.setCenterY(y + (phase * 100));
        ... // setting the circle
        i+=5;
```

Here is an inner class implementation of an `AudioSpectrumListener`:

```
        new AudioSpectrumListener() {
            @Override
            public void spectrumDataUpdate(double timestamp, double duration, float[]
magnitudes, float[] phases) {

                phaseNodes.getChildren().clear();
                int i = 0;
```

```
            int x = 10;
            int y = 150;
            final Random rand = new Random(System.currentTimeMillis());
            for(float phase:phases) {
              int red = rand.nextInt(255);
              int green = rand.nextInt(255);
              int blue = rand.nextInt(255);

              Circle circle = new Circle(10);
              circle.setCenterX(x + i);
              circle.setCenterY(y + (phase * 100));
              circle.setFill(Color.rgb(red, green, blue, .70));
              phaseNodes.getChildren().add(circle);
              i+=5;
            }

          }
        };
```

Once the media player is created, you will create a `java.lang.Runnable` to be set into the `onReady` attribute to be invoked when the media is in a ready state. Once the ready event is realized the `run()` method will call the media player object's `play()` method to begin the audio. With the dragged-drop sequence completed, we appropriately notify the drag-and-drop system by invoking the event's `setDropCompleted()` method with a value of true. The following code snippet implements a `Runnable` to begin the media player as soon as the media player is in a ready state:

```
mediaPlayer.setOnReady(new Runnable() {
   @Override
   public void run() {
      mediaPlayer.play();
   }
});
```

Finally you will be creating buttons with JavaFX shapes to represent the stop, play, pause, and close buttons. When creating shapes or custom nodes, you can add event handlers to nodes in order to respond to mouse clicks. Although there are advanced ways to build custom controls in JavaFX, I chose to build my own button icons from simple rectangles, arcs, circles and lines. To see more-advanced ways to create custom controls, refer to the Javadocs on the `Skinnable` API or recipe 2-5. To attach event handlers for a mouse press, simply call the `setOnMousePress()` method by passing in an `EventHandler<MouseEvent>` instance. The following code demonstrates adding an `EventHandler` to respond to mouse press on the `stopButton` node:

```
stopButton.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
       if (mediaPlayer!= null) {
         mediaPlayer.stop();
       }
    }
});
```

Because all the buttons use the same preceding code snippet, I will only list the method calls that each button will perform on the media player. The last button, Close, isn't related to the media player, but it is how to exit the MP3 player application. The following actions are responsible for stopping, pausing, playing, and exiting the MP3 player application:

```
Stop - mediaPlayer.stop();
Pause - mediaPlayer.pause();
Play - mediaPlayer.play();
Close - Platform.exit();
```

3-2. Playing Video

Problem

You want to view a video file complete with controls to play, pause, stop, and seek.

Solution

Create a video media player by utilizing the following classes:

- `javafx.scene.media.Media`
- `javafx.scene.media.MediaPlayer`
- `javafx.scene.media.MediaView`

The following code is an implementation of a JavaFX basic video player:

```
public void start(final Stage primaryStage) {
    primaryStage.setTitle("Chapter 3-2 Playing Video");
    ... setting up the stage

    // rounded rectangle with slightly transparent
    Node applicationArea = createBackground(scene);
    root.getChildren().add(applicationArea);

    // allow the user to drag window on the desktop
    attachMouseEvents(scene, primaryStage);

    // allows the user to see the progress of the video playing
    progressSlider = createSlider(scene);
    root.getChildren().add(progressSlider);

    // Dragging over surface
    scene.setOnDragOver(… Drag Over code );

    // update slider as video is progressing (later removal)
    progressListener = new ChangeListener<Duration>() {
        public void changed(ObservableValue<? extends Duration> observable, Duration
oldValue, Duration newValue) {
            progressSlider.setValue(newValue.toSeconds());
        }
```

```
        };


        // Dropping over surface
        scene.setOnDragDropped(new EventHandler<DragEvent>() {

            @Override
            public void handle(DragEvent event) {
                Dragboard db = event.getDragboard();
                boolean success = false;
                URI resourceUrlOrFile = null;

                … // detect and obtain media file

                // load media
                Media media = new Media(resourceUrlOrFile.toString());

                // stop previous media player and clean up
                if (mediaPlayer != null) {
                    mediaPlayer.stop();
                    mediaPlayer.currentTimeProperty().removeListener(progressListener);
                    mediaPlayer.setOnPaused(null);
                    mediaPlayer.setOnPlaying(null);
                    mediaPlayer.setOnReady(null);
                }

                // create a new media player
                mediaPlayer = MediaPlayerBuilder.create()
                        .media(media)
                        .build();

                // as the media is playing move the slider for progress
                mediaPlayer.currentTimeProperty().addListener(progressListener);


                // play video when ready status
                mediaPlayer.setOnReady(new Runnable() {
                    @Override
                    public void run() {
                       progressSlider.setValue(1);

progressSlider.setMax(mediaPlayer.getMedia().getDuration().toMillis()/1000);
                       mediaPlayer.play();
                    }
                });

                // Lazy init media viewer
                if (mediaView == null) {
                    mediaView = MediaViewBuilder.create()
                            .mediaPlayer(mediaPlayer)
                            .x(4)
                            .y(4)
```

```
                        .preserveRatio(true)
                        .opacity(.85)
                        .smooth(true)
                        .build();


mediaView.fitWidthProperty().bind(scene.widthProperty().subtract(220));

mediaView.fitHeightProperty().bind(scene.heightProperty().subtract(30));

                // make media view as the second node on the scene.
                root.getChildren().add(1, mediaView);
            }

            // sometimes loading errors occur
            mediaView.setOnError(new EventHandler<MediaErrorEvent>() {
                public void handle(MediaErrorEvent event) {
                    event.getMediaError().printStackTrace();
                }
            });

            mediaView.setMediaPlayer(mediaPlayer);

            event.setDropCompleted(success);
            event.consume();
        }
    });

    // rectangular area holding buttons
    final Group buttonArea = createButtonArea(scene);

    // stop button will stop and rewind the media
    Node stopButton = createStopControl();

    // play button can resume or start a media
    final Node playButton = createPlayControl();

    // pauses media play
    final Node pauseButton = createPauseControl();

    stopButton.setOnMousePressed(new EventHandler<MouseEvent>() {
        public void handle(MouseEvent me) {
            if (mediaPlayer!= null) {
                buttonArea.getChildren().removeAll(pauseButton, playButton);
                buttonArea.getChildren().add(playButton);
                mediaPlayer.stop();
            }
        }
    });
    // pause media and swap button with play button
    pauseButton.setOnMousePressed(new EventHandler<MouseEvent>() {
        public void handle(MouseEvent me) {
```

```
            if (mediaPlayer!=null) {
                buttonArea.getChildren().removeAll(pauseButton, playButton);
                buttonArea.getChildren().add(playButton);
                mediaPlayer.pause();
                paused = true;
            }
        }
    });

    // play media and swap button with pause button
    playButton.setOnMousePressed(new EventHandler<MouseEvent>() {
        public void handle(MouseEvent me) {
            if (mediaPlayer != null) {
                buttonArea.getChildren().removeAll(pauseButton, playButton);
                buttonArea.getChildren().add(pauseButton);
                paused = false;
                mediaPlayer.play();
            }
        }
    });

    // add stop button to button area
    buttonArea.getChildren().add(stopButton);

    // set pause button as default
    buttonArea.getChildren().add(pauseButton);

    // add buttons
    root.getChildren().add(buttonArea);

    // create a close button
    Node closeButton= createCloseButton(scene);
    root.getChildren().add(closeButton);

    primaryStage.setOnShown(new EventHandler<WindowEvent>() {
        public void handle(WindowEvent we) {
            previousLocation = new Point2D(primaryStage.getX(), primaryStage.getY());
        }
    });

    primaryStage.setScene(scene);
    primaryStage.show();

}
```

Following is our `attachMouseEvents()` method that adds an `EventHandler` to the `Scene` to provide the ability to make the video player go into full screen mode.

```
private void attachMouseEvents(Scene scene, final Stage primaryStage) {
```

```
        // Full screen toggle
        scene.setOnMouseClicked(new EventHandler<MouseEvent>() {
            public void handle(MouseEvent event){
                if (event.getClickCount() == 2) {
                    primaryStage.setFullScreen(!primaryStage.isFullScreen());
                }
            }
        });
        ... // the rest of the EventHandlers
    }
```

The following code is a method that creates a slider control with a `ChangeListener` to enable the user to seek backward and forward through the video:

```
    private Slider createSlider(Scene scene) {
        Slider slider = SliderBuilder.create()
                .min(0)
                .max(100)
                .value(1)
                .showTickLabels(true)
                .showTickMarks(true)
                .build();

        slider.valueProperty().addListener(new ChangeListener<Number>() {
            public void changed(ObservableValue<? extends Number> observable, Number oldValue,
Number newValue) {
                if (paused) {
                    long dur = newValue.intValue() * 1000;
                    mediaPlayer.seek(new Duration(dur));
                }
            }
        });
        slider.translateYProperty().bind(scene.heightProperty().subtract(30));
        return slider;
    }
```

Figure 3-3 depicts a JavaFX basic video player with a slider control.

Figure 3-3. *JavaFX basic video player*

How It Works

To create a video player you will model the application similar to recipe 3-1 by reusing the same application features such as drag-and-drop files, media button controls, and so on. For the sake of clarity, I took the previous recipe and moved much of the UI code into convenience functions so you will be able to focus on the Media APIs without getting lost in the UI code. The rest of the recipes in this chapter consist of adding simple features to the JavaFX basic media player created in this recipe. This being said, the code snippets in the following recipes will be brief, consisting of the necessary code needed for each new desired feature.

Before we begin, I want to talk about media formats. As of the writing of this book. JavaFX 2.0 supports a cross-platform video format called VP6 with a file extension of `.flv` (which stands for the popular Adobe Flash Video format). The actual encoder and decoder (Codec) to create VP6 and `.flv` files are licensed through a company called On2. In 2009, On2 was acquired by Google to build VP7 and VP8 to be open and free to advance HTML5. I don't want to confuse you with the drama, but it is difficult to see how things will unfold as media formats become favored or considered obsolete. Because JavaFX's goal is to be cross-platform, it would seem logical to use the most popular codec on the Net, but you will be forced to obtain a license to encode your videos into the VP6 `.flv` file format. So the bottom line is that JavaFX currently can only play video files that are encoded in VP6. (I try to keep in mind that this is the state of media formats today, so don't channel any frustrations toward the JavaFX SDK.) Please refer to the Javadoc API for more details on the formats to be used. A word to the wise: beware of web sites claiming to be able to convert videos for free. As of this writing, the only encoders capable of encoding video to VP6 legally are the commercial converters from Adobe and Wildform (`http://www.wildform.com`).

Now, that you know what is the acceptable file format you are probably wondering how to obtain such a file of this type if you don't have encoding software. If you don't have an `.flv` file lying around, you can obtain one from one of my favorite sites called the Media College (`http://www.mediacollege.com`). From photography to movies, Media College provides forums, tutorials, and resources that help guide you into the world of media. There you will obtain a particular media file to be used in the remaining recipes in this chapter. To obtain the `.flv` file you will navigate to the following URL: `http://www.mediacollege.com/adobe/flash/video/tutorial/example-flv.html`.

Next, you will locate the link entitled `Windy 50s Mobility Scooter Race` that points to our `.flv` media file (`20051210-w50s.flv`). In order to download a link consisting of a file, right-click to select "Save target as" or "Save link as". Once you have saved the file locally on your file system, you can drag the file into the media player application to begin the demo.

■ **Note** As of the writing of this book, the JavaFX media player API currently supports the video format VP6 using an `.flv` container.

Just like the audio player created in the last recipe, our JavaFX basic video player has the same basic media controls, including stop, pause, and play. In addition to these simple controls we have added new capabilities such as seeking and full screen mode.

When playing a video you'll need a view area (`javafx.scene.media.MediaView`) to show the video. You will also be creating a slider control to monitor the progress of the video, which is located at the lower left of the application shown in Figure 3-3. The slider control allows the user to seek backward and forward through the video. The ability to seek will work only if the video is paused. One last bonus feature is making the video become full screen by double-clicking the application window. To restore the window, repeat the double click or press Escape.

To quickly get started, let's jump into the code. After setting up the stage in the `start()` method, you will create a black semitransparent background by calling the `createBackground()` method (`applicationArea`). Next, you will be invoking the `attachMouseEvents()` method to wire up all the `EventHandlers` into the scene that will enable the user to drag the application window about the desktop. Another `EventHandler` to be attached to the `Scene` will allow the user to switch to full screen mode. To make a window turn into full screen mode, you will create a conditional to check for the double click of the application window. Once the double-click is performed you will call the `Stage's` method `setFullScreen()` with a Boolean value opposite of the currently set value. Shown here is how to make a window go to full screen mode:

```
// Full screen toggle
scene.setOnMouseClicked(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent event){
        if (event.getClickCount() == 2) {
            primaryStage.setFullScreen(!primaryStage.isFullScreen());
        }
    }
});
```

As we continue our steps inside the `start()` method, you will create a slider control by calling the convenience method `createSlider()`. The `createSlider()` method will instantiate a `Slider` control and add a `ChangeListener` to move the slider as the video is playing. The `ChangeListener`'s `changed()` method is invoked any time the slider's value changes. Once the `changed()` method is invoked you will have an opportunity to see the old value and the new value. The following code creates a `ChangeListener` to update the slider as the video is being played:

```
// update slider as video is progressing (later removal)
progressListener = new ChangeListener<Duration>() {
    public void changed(ObservableValue<? extends Duration> observable, Duration
oldValue, Duration newValue) {
        progressSlider.setValue(newValue.toSeconds());
```

```
            }
        };
```

After creating the progress listener (`progressListener`), you will be creating the dragged-dropped `EventHandler` on the `Scene`.

The goal is to determine whether the pause button was pressed before the user can move the slider. Once a paused flag is determined, you will obtain the new value to be converted to milliseconds. The `dur` variable is used to move the `mediaPlayer` to seek the position into the video as the user slides the control left or right. The `ChangeListener`'s `changed()` method is invoked any time the slider's value changes. The following code is responsible for moving the seek position into the video based on the user moving the slider.

```
slider.valueProperty().addListener(new ChangeListener<Number>() {
   public void changed(ObservableValue<? extends Number> observable, Number oldValue, Number
newValue) {
                if (paused) {
                    long dur = newValue.intValue() * 1000;
                    mediaPlayer.seek(new Duration(dur));
                }
            }
        });
```

Moving right along, you will be implementing a drag-dropped `EventHandler` to handle the .flv media file being dropped into the application window area. Here you'll first check to see whether there was a previous `mediaPlayer`. If so, you will stop the previous `mediaPlayer` object and do some cleanup:

```
        // stop previous media player and clean up
        if (mediaPlayer != null) {
           mediaPlayer.stop();
           mediaPlayer.currentTimeProperty().removeListener(progressListener);
           mediaPlayer.setOnPaused(null);
           mediaPlayer.setOnPlaying(null);
           mediaPlayer.setOnReady(null);
        }



        // play video when ready status
        mediaPlayer.setOnReady(new Runnable() {
            @Override
            public void run() {
               progressSlider.setValue(1);

        progressSlider.setMax(mediaPlayer.getMedia().getDuration().toMillis()/1000);
           mediaPlayer.play();
        }
    }); // setOnReady()
```

As with the audio player, we create a `Runnable` instance to be run when the media player is in a ready state. You'll notice also that the `progressSlider` control being set up to use values in seconds.

Once the media player object is in a ready state you will be creating a `MediaView` instance to display the media. Shown following is the creation of a `MediaView` object to be put into the scene graph to display video content:

```
        // Lazy init media viewer
        if (mediaView == null) {
            mediaView = MediaViewBuilder.create()
                    .mediaPlayer(mediaPlayer)
                    .x(4)
                    .y(4)
                    .preserveRatio(true)
                    .opacity(.85)
                    .build();


            mediaView.fitWidthProperty().bind(scene.widthProperty().subtract(220));

            mediaView.fitHeightProperty().bind(scene.heightProperty().subtract(30));

                    // make media view as the second node on the scene.
                    root.getChildren().add(1, mediaView);
                }

                // sometimes loading errors occur
                mediaView.setOnError(new EventHandler<MediaErrorEvent>() {
                    public void handle(MediaErrorEvent event) {
                        event.getMediaError().printStackTrace();
                    }
                });

                mediaView.setMediaPlayer(mediaPlayer);
                event.setDropCompleted(success);
                event.consume();
            }
        });
```

Whew! We are finally finished with our drag-dropped `EventHandler` for our `Scene`. Up next is pretty much the rest of the media button controls similar to the end of recipe 3-1. The only thing different is a single instance variable named `paused` of type `boolean` that denotes whether the video was paused. This `paused` flag when set to true will allow the slider control to seek forward or backward through the video; otherwise false. Following is the `pauseButton` and `playButton` controlling the `mediaPlayer` object and setting the `paused` flag accordingly:

```
// pause media and swap button with play button
pauseButton.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        if (mediaPlayer!=null) {
            buttonArea.getChildren().removeAll(pauseButton, playButton);
            buttonArea.getChildren().add(playButton);
            mediaPlayer.pause();
            paused = true;
        }
```

```
        }
    });


    // play media and swap button with pause button
    playButton.setOnMousePressed(new EventHandler<MouseEvent>() {
        public void handle(MouseEvent me) {
            if (mediaPlayer != null) {
                buttonArea.getChildren().removeAll(pauseButton, playButton);
                buttonArea.getChildren().add(pauseButton);
                paused = false;
                mediaPlayer.play();
            }
        }
    });
```

So that is how to create a video media player. In the next recipe, you will be able to listen to media events and invoke actions.

3-3. Controlling Media Actions and Events

Problem

You want the media player to provide feedback in response to certain events. An example is displaying the text "Paused" on the screen when the media player's `paused` event is triggered.

Solution

You can use many media event handler methods. Shown in Table 3-3 are all the possible media events that are raised to allow the developer to attach `EventHandlers` or `Runnables`.

Table 3-3. *Media Events*

Class	Set On Method	On Method Property Method	Description
`Media`	`setOnError()`	`onErrorProperty()`	When an error occurs
`MediaPlayer`	`setOnEndOfMedia()`	`onEndOfMediaProperty()`	Reached the end of the media play
`MediaPlayer`	`setOnError()`	`onErrorProperty()`	Error occurred
`MediaPlayer`	`setOnHalted()`	`onHaltedProperty()`	Media status changes to HALTED
`MediaPlayer`	`setOnMarker()`	`onMarkerProperty()`	Marker event triggered

Class	Set On Method	On Method Property Method	Description
MediaPlayer	setOnPaused()	onPausedProperty()	Paused event occurred
MediaPlayer	setOnPlaying()	onPlayingProperty()	The media is currently playing
MediaPlayer	setOnReady()	onReadyProperty()	Media player is in Ready state
MediaPlayer	setOnRepeat()	onRepeatProperty()	Repeat property is set
MediaPlayer	setOnStalled()	onStalledProperty()	Media player is stalled
MediaPlayer	setOnStopped()	onStoppedProperty()	Media player has stopped
MediaView	setOnError()	onErrorProperty()	Error occurred in Media View

The following code will present to the user a text “Paused” with “Duration” with a decimal of milliseconds which is overlaid on top of the video when the user clicks the pause button (see Figure 3-4):

```
   // when paused event display pause message
   mediaPlayer.setOnPaused(new Runnable() {
     @Override
     public void run() {
        pauseMessage.setText("Paused \nDuration: " +
mediaPlayer.currentTimeProperty().getValue().toMillis());
        pauseMessage.setOpacity(.90);

    }
  });
```

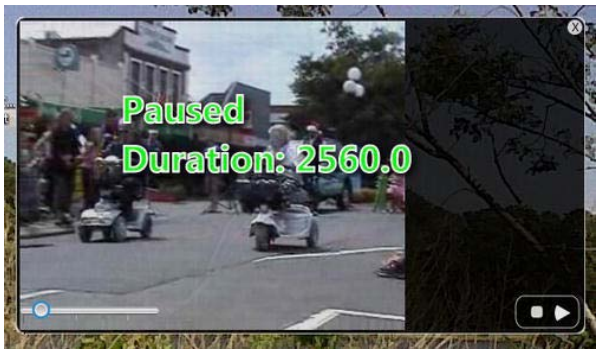

Figure 3-4. *Paused event*

How It Works

An event driven architecture (EDA) is a prominent architectural pattern used to model loosely coupled components and services that pass messages asynchronously. The JavaFX team has designed the Media API to be event driven. This recipe will demonstrate how to implement in response to media events.

With event-based programming in mind, you will discover nonblocking or callback behaviors when invoking functions. In this recipe you will implement the display of text in response to an `onPaused` event instead of placing your code into the pause button. Instead of tying code directly to a button via an `EventHandler`, you will be implementing code that will respond to the media player's `onPaused` event being triggered. When responding to media events, you will be implementing `java.lang.Runnables`.

You'll be happy to know that you've been using event properties and implementing `Runnables` all along. Hopefully you noticed this in all the recipes in this chapter. When the media player is in a ready state, the `Runnable` code will be invoked. Why is this correct? Well, when the media player is finished loading the media, the `onReady` property will be notified. That way you can be sure you can invoke the `MediaPlayer`'s `play()` method. I trust that you will get used to event style programming. The following code snippet demonstrates the setting of a `Runnable` instance into a media player object's `OnReady` property:

```
        mediaPlayer.setOnReady(new Runnable() {
            @Override
            public void run() {
                mediaPlayer.play();
            }
        });
```

You will be taking steps similar to the `onReady` property. Once a `Paused` event has been triggered, the `run()` method will be invoked to present to the user a message containing a `Text` node with the word `Paused` and a duration showing the time in milliseconds into the video. Once displayed, you might want to write down the duration as markers (as you'll learn recipe 3-4). The following code snippet shows an attached `Runnable` instance, which is responsible for displaying a paused message and duration in milliseconds at the point in which it was paused in the video:

```
  // when paused event display pause message
  mediaPlayer.setOnPaused(new Runnable() {
     @Override
     public void run() {
        pauseMessage.setText("Paused \nDuration: " +
mediaPlayer.currentTimeProperty().getValue().toMillis());
        pauseMessage.setOpacity(.90);

     }
  });
```

3-4. Marking a Position in a Video

Problem

You want to provide closed caption text while playing a video in the media player.

Solution

Begin by applying recipe 3-3. By obtaining the marked durations (in milliseconds) from the previous recipe you will create media marker events at points into the video. With each media marker you will associate text that will be displayed as closed captions. When a marker comes to pass, a text will be shown to the upper-right side.

The following code snippet demonstrates media marker events being handled in the `onDragDropped` event property of the `Scene` object:

```
... // inside the start() method

final VBox messageArea = createClosedCaptionArea(scene);
root.getChildren().add(messageArea);

// Dropping over surface
scene.setOnDragDropped(new EventHandler<DragEvent>() {

    @Override
    public void handle(DragEvent event) {
        Dragboard db = event.getDragboard();

        ... // drag dropped code goes here


        // load media
        Media media = new Media(resourceUrlOrFile.toString());

        ... // clean up media player

        // create a new media player
        mediaPlayer = MediaPlayerBuilder.create()
                .media(media)
                .build();

        ...// Set media 'onXXX' event properties

        mediaView.setMediaPlayer(mediaPlayer);


        media.getMarkers().put("Starting race", Duration.millis(1959.183673));
        media.getMarkers().put("He is begining \nto get ahead", Duration.millis(3395.918367));
        media.getMarkers().put("They are turning \nthe corner", Duration.millis(6060.408163));
        media.getMarkers().put("The crowds cheer", Duration.millis(9064.489795));
        media.getMarkers().put("He makes the \nfinish line", Duration.millis(11546.122448));

        // display closed captions
        mediaPlayer.setOnMarker(new EventHandler<MediaMarkerEvent> (){
            public void handle(MediaMarkerEvent event){
                closedCaption.setText(event.getMarker().getKey());
            }
        });
```

```
            event.setDropCompleted(success);
            event.consume();
        }
    }); // end of setOnDragDropped()
```

Shown following is a factory method that returns an area that will contain the closed caption to be displayed to the right of the video:

```
    private VBox createClosedCaptionArea(final Scene scene) {
        // create message area
        final VBox messageArea = new VBox(3);
        messageArea.setTranslateY(30);
        messageArea.translateXProperty().bind(scene.widthProperty().subtract(152) );
        messageArea.setTranslateY(20);
        closedCaption = TextBuilder.create()
            .stroke(Color.WHITE)
            .fill(Color.YELLOW)
            .font(new Font(15))
            .build();
        messageArea.getChildren().add(closedCaption);
        return messageArea;
    }
```

Figure 3-5 depicts the video media player displaying closed caption text.

Figure 3-5. *Closed caption text*

How It Works

The Media API has many event properties that the developer can attach `EventHanders` or `Runnables` instances so they can respond when the events are triggered. Here you focus on the `OnMarker` event property. The `Marker` property is responsible for receiving marker events (`MediaMarkerEvent`).

Let's begin by adding markers into our `Media` object. It contains a method `getMarkers()` that returns an `javafx.collections.ObservableMap<String, Duration>`. With an observable map, you can add key value pairs that represent each marker. Adding keys should be a unique identifier, and the value is an instance of `Duration`. For simplicity I used the closed caption text as the key for each media marker. The marker durations are those written down as you press the pause button at points in the video from

recipe 3-3. Please be advised that I don't recommend doing this in production code. You may want to use a parallel `Map`.

After adding markers you will be setting an `EventHandler` into the `MediaPlayer` object's `OnMarker` property using the `setOnMarker()` method. Next, you will create the `EventHandler` instance to handle `MediaMarkerEvents` that are raised. Once an event has been received, obtain the key representing the text to be used in the closed caption. The instance variable `closedCaption` (`javafx.scene.text.Text` node) will simply be shown by calling the `setText()` method with the key or string associated to a marker.

That's it for media markers. That goes to show you how you can coordinate special effects, animations, and so on during a video quite easily.

3-5. Synchronizing Animation and Media

Problem

You want to incorporate animated effects in your media display. For example, you want to scroll "The End" after a video is finished playing.

Solution

Use recipe 3-3 together with recipe 2-2. Receipe 3-3 shows how to respond to media events. Recipe 2-2 demonstrates how to use `javafx.animation.TranslateTransition` to animate text.

The following code demonstrates an attached action when an end of a media event is triggered:

```
mediaPlayer.setOnEndOfMedia(new Runnable() {
   @Override
   public void run() {
      closedCaption.setText("");
      animateTheEnd.getNode().setOpacity(.90);
      animateTheEnd.playFromStart();
   }
});
```

Shown here is a method that creates a `translateTransition` of a `Text` node containing the string "The End" that animates after an end of media event is triggered:

```
public TranslateTransition createTheEnd(Scene scene) {
    Text theEnd = TextBuilder.create()
       .text("The End")
       .font(new Font(40))
       .strokeWidth(3)
       .fill(Color.WHITE)
       .stroke(Color.WHITE)
       .x(75)
       .build();

    TranslateTransition scrollUp = TranslateTransitionBuilder.create()
        .node(theEnd)
        .duration(Duration.seconds(1))
        .interpolator(Interpolator.EASE_IN)
```

```
            .fromY(scene.getHeight() + 40)
            .toY(scene.getHeight()/2)
            .build();
        return scrollUp;
    }
```

Figure 3-6 depicts the text node "The End" scrolling up after the `OnEndOfMedia` event is triggered.

Figure 3-6. *Animate The End*

How It Works

In this recipe you will be able to synchronize events to animated effects. In other words, when the video reaches the end, an `OnEndOfMedia` property event will initiate a `Runnable` instance. Once initiated, a `TranslateTransition` animation is performed by scrolling a `Text` node upward with the string "The End".

So, let me describe the `setOnEndOfMedia()` method associated with the `MediaPlayer` object. Just like recipe 3-3, we simply call the `setOnEndOfMedia()` method by passing in a `Runnable` that contains our code that will invoke an animation. If you don't know how animation works, please refer to recipe 2-2. Once the event occurs, you will see the text scroll upward. The following code snippet is from inside the `scene.setOnDragDropped()` method:

```
    mediaPlayer.setOnEndOfMedia(new Runnable() {
       @Override
       public void run() {
          closedCaption.setText("");
          animateTheEnd.getNode().setOpacity(.90);
          animateTheEnd.playFromStart();
       }
    });
```

For the sake of space, I trust you know where the code block would reside. If not, you may refer to recipe 3-3, in which you will notice other `OnXXX` properties methods. To see the entire code listing, visit the book's web site to download the source code.

To animate the text "The End" you will create a convenience `createTheEnd()` method to create an instance of a `Text` node and return a `TranslateTransition` object to the caller. The `TranslateTransition` returned will do the following: wait a second before playing video. Next is the interpolator in which I used the `Interpolator.EASE_IN` to move the `Text` node by easing in before a full stop. Last is setting up the Y property of the node to move from the bottom to the center of the Media view area.

The following code is an animation to scroll a node in an upward motion:

```
TranslateTransition scrollUp = TranslateTransitionBuilder.create()
            .node(theEnd)
            .duration(Duration.seconds(1))
            .interpolator(Interpolator.EASE_IN)
            .fromY(scene.getHeight() + 40)
            .toY(scene.getHeight()/2)
            .build();
```

CHAPTER 4

JavaFX on the Web

JavaFX provides new capabilities to interoperate with HTML5. The underlying web page–rendering engine in JavaFX is the popular open-source API called Webkit. Webkit is also used in Google's Chrome and Apple's Safari browsers. HTML5 is the new standard markup language for rendering content in web browsers. HTML5 content consists of JavaScript, CSS, Scalable Vector Graphics (SVG), and new HTML element tags.

The relationship between JavaFX and HTML5 is important because they complement one another by drawing from each of their individual strengths. For instance, JavaFX's rich client APIs coupled with HTML5's rich web content create a user experience resembling a web application with the characteristics of desktop software. This new breed of applications is called RIAs.

In this chapter, we will cover the following:

- Embedding JavaFX applications in an HTML web page
- Displaying HTML 5 content
- Manipulating HTML5 content with Java code
- Responding to HTML events
- Displaying content from the database

4-1. Embedding JavaFX Applications in a Web Page

Problem

You hope to get promoted out of your cubicle into an office with windows by impressing your boss by creating a proof of concept using JavaFX with your existing web development skills.

Solution

Create a Hello World program using the NetBeans IDE 7.1 or later by using its new project wizard to create an application to run in a browser. Shown following are steps to follow to create a Hello World JavaFX application that is embedded in an HTML web page:

■ **Note** For in-depth JavaFX deployment strategies refer to Oracle's deploying JavaFX Applications: `http://download.oracle.com/javafx/2.0/deployment/deployment_toolkit.htm`.

Here are the steps to follow in running the new project wizard:

1. Select New Project in the File menu of the NetBeans IDE version 7.1 or later. Figure 4-1 highlights the menu option in the NetBeans File menu.

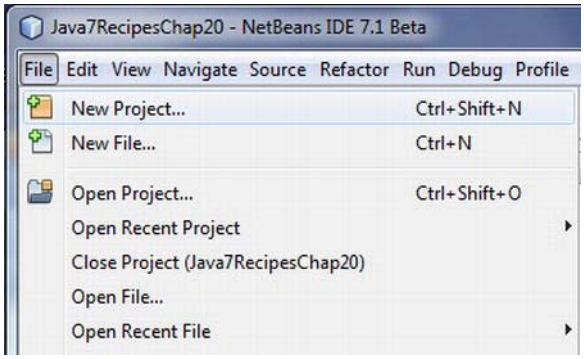

Figure 4-1. *Creating a new JavaFX project*

2. Select JavaFX in the Categories section under Choose Project, as shown in Figure 4-2. Next, select JavaFX Application under Projects. Then click Next to proceed.

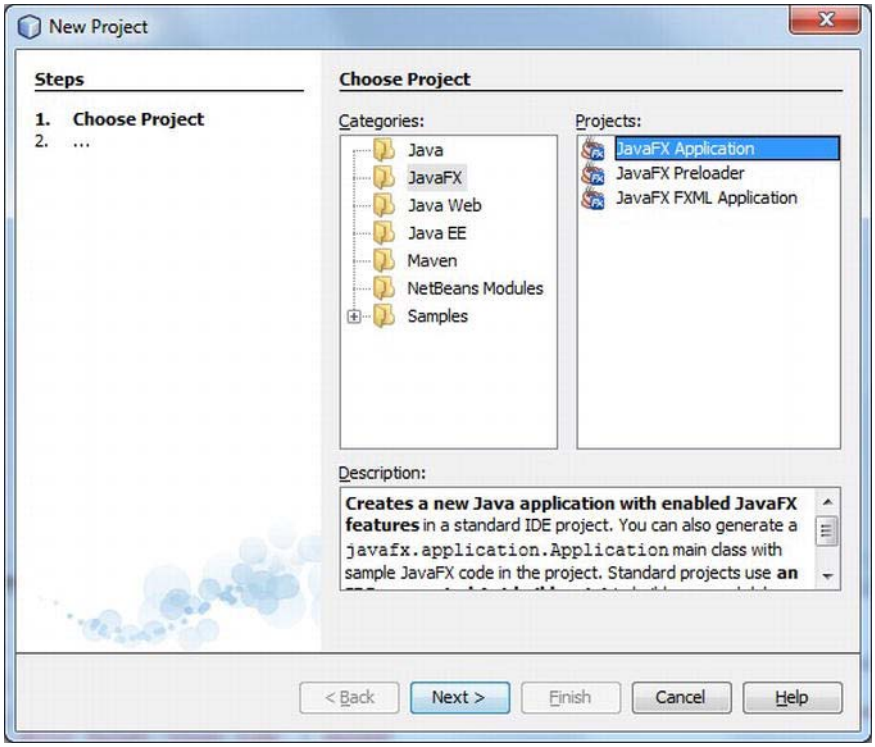

Figure 4-2. *New Project dialog box*

3. Create a project by specifying a name and selecting the check box to allow the wizard to generate a main class called `MyJavaFXApp.java`. Figure 4-3 shows a New JavaFX application wizard that specifies the project name and location. When you finish, click the Finish button.

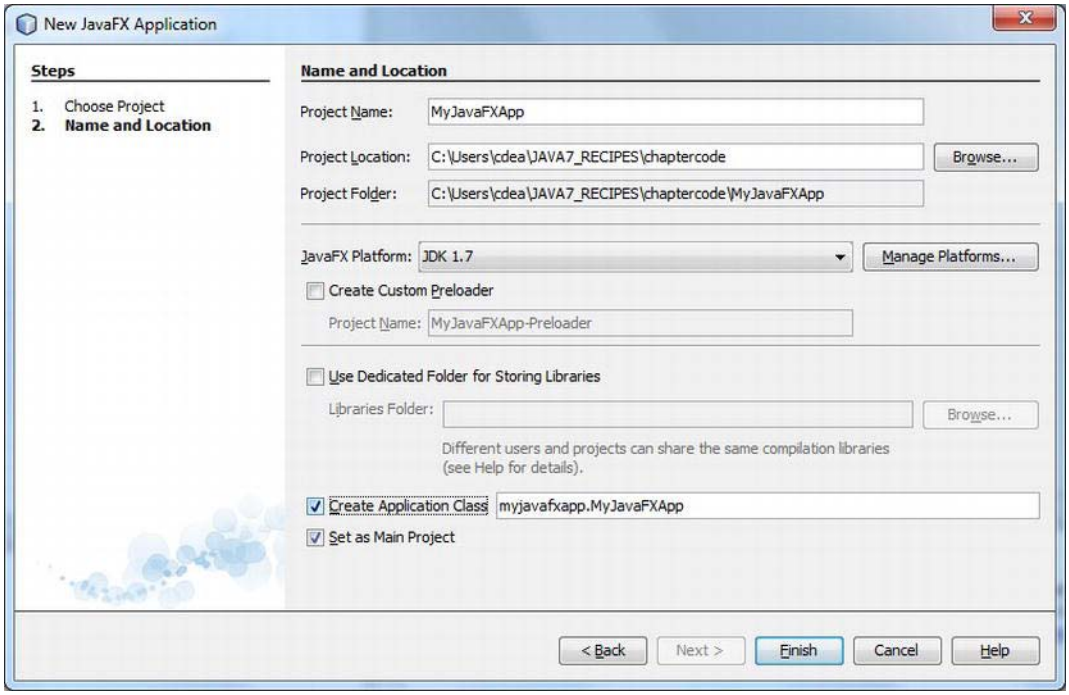

Figure 4-3. *New JavaFX Application dialog box, in which you specify Project Name and Project Location*

4. Once your new project has been created, you modify project properties. To modify the properties, right-click the project and select Properties via the popup menu. Figure 4-4 shows the project created with a main JavaFX file named `MyJavaFXApp.java`.

Figure 4-4. *MyJavaFXApp.java project*

5. Go into the project's properties, as shown in Figure 4-5. Select Sources in the categories option area. Next, check the Source/Binary Format option to point to JDK 7.

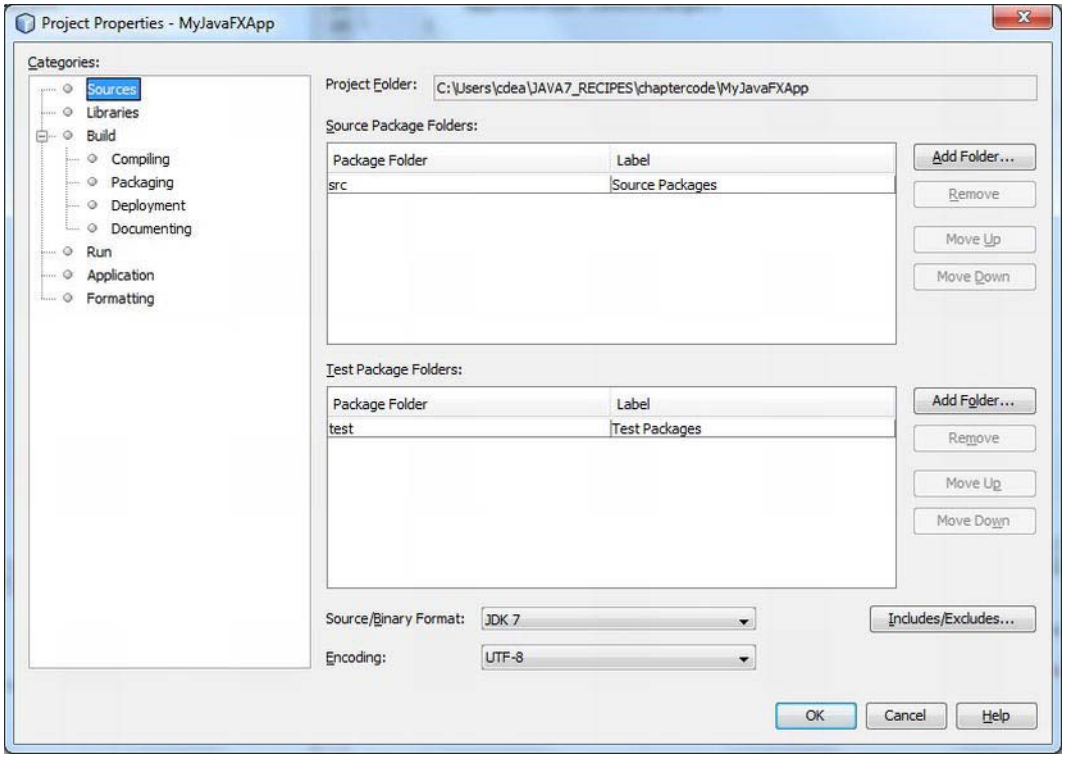

Figure 4-5. *Project Properties MyJavaFXApp dialog window*

6. Select the Run option in the Categories list shown in Figure 4-6. Select the in Browser radio button option. Then click the OK button.

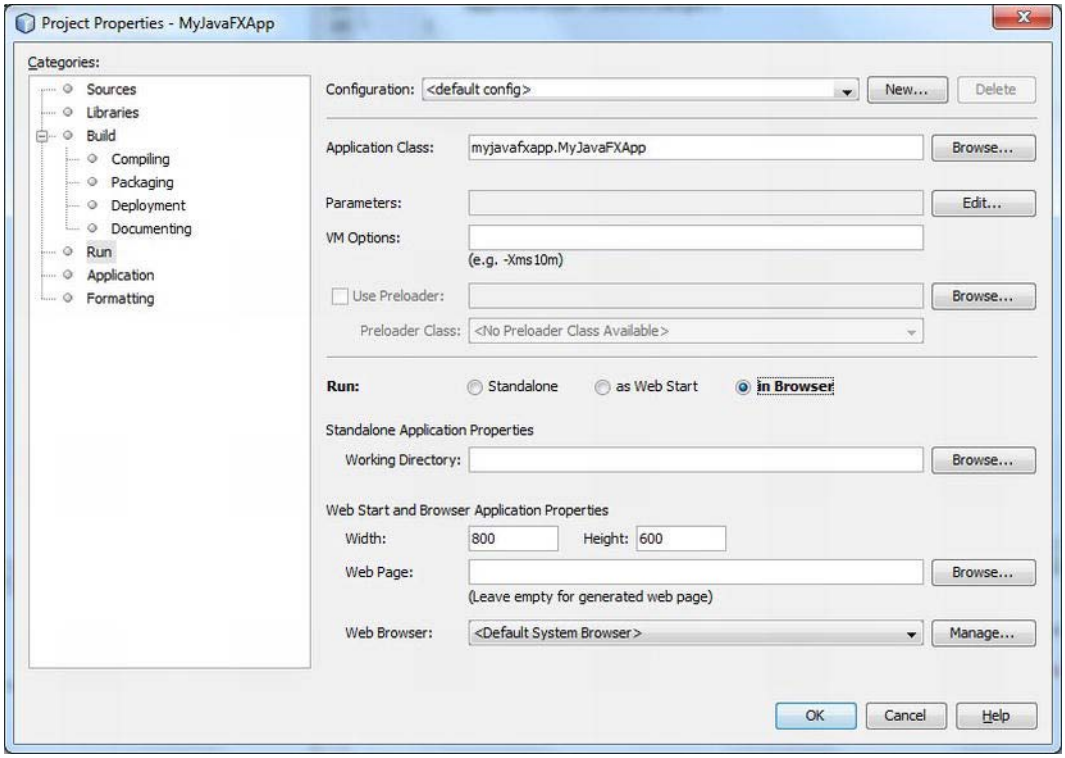

Figure 4-6. *Setting up the Run option in Browser*

7. Run and test the project by clicking the Run button on the toolbar or the F6 key. Figure 4-7 depicts the resulting Hello World application running in a browser.

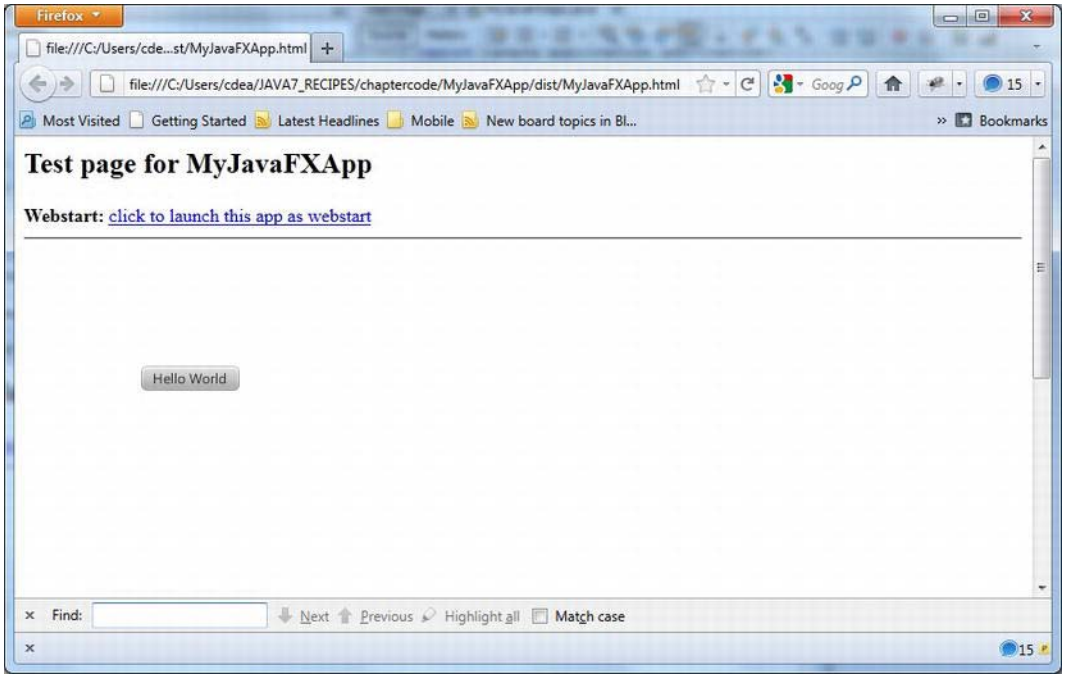

Figure 4-7. *The MyJavaFXApp Hello World application running inside a browser*

How It Works

To create an embedded JavaFX application inside an HTML page, you use the NetBeans IDE. Although there are different deployment strategies, such as Webstart and Standalone modes, here you use the NetBeans new project wizard to automatically deploy as a local web page containing your JavaFX application in your browser. For in-depth JavaFX deployment strategies, refer to Oracle's Deploying JavaFX Applications: `http://download.oracle.com/javafx/2.0/deployment/deployment_toolkit.htm`.

Following is the code generated by this solution. You will notice the JavaFX classes being used; for example, `Stage`, `Group`, and `Scene` classes.

■ **Note** You can drag the imports and body of code from another code file for this recipe into the body of your new main project class, changing the name on the class definition line, as appropriate.

Following is the source code when the NetBeans' wizard generates a new project to create a JavaFX application embedded in a HTML web page:

```
package myjavafxapp;

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

/**
 *
 * @author cdea
*/
public class MyJavaFXApp extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");
        Group root = new Group();
        Scene scene = new Scene(root, 300, 250);
        Button btn = new Button();
        btn.setLayoutX(100);
        btn.setLayoutY(80);
        btn.setText("Hello World");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            public void handle(ActionEvent event) {
                System.out.println("Hello World");
            }
        });
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

In Step 1, you initiate a new project (shown in Figure 4-7). In Step 2, you select the standard JavaFX application to be created. After selecting the project type, you will be specifying the name of the project. Make sure you click the Create Application Class check box to allow the wizard to generate the MyJavaFXApp Java file. Once you have clicked Finish, your newly created application will appear in the projects tab. Next, you will take Step 5 in changing project properties.

In Step 5 you will be changing two categories: Sources and Run. In the Sources category, make sure the Source/Binary Format is set to JDK 1.6 or later. After updating the Sources category, you will be changing how the project will run (Step 6) through the Run category. In Step 6, after selecting the in

Browser radio button option, you will notice the Width and Height below the working directory field. To use your own custom web page, you click the browse button to select an existing HTML file, but in this recipe you can leave it blank to allow the wizard to generate a generic HTML page. Assuming that you are done with your settings, click OK to close the Project Properties dialog window.

Last, you will run your embedded JavaFX web application (Step 7). To run your application you will want to make sure this project is set as the main project by selecting in the menu `Run -> Set Main Project ->MyJavaFXApp`. Once you are initiating a run, your browser will launch, containing a generic web page with your JavaFX application. You'll also notice that a convenient link allows you to launch the application as a Webstart application (not embedded).

4-2. Displaying HTML5 Content

Problem

You are so engrossed with a project for work that you often miss your kid's soccer games. What you need is a clock application to keep track of the time.

Solution

Create a JavaFX based-application containing an analog clock that was created as HTML5 content. Use JavaFX's `WebView` API to render HTML5 content in your application.

The following source code is a JavaFX application displaying an animated analog clock. The application will load an SVG file named `clock3.svg` and display the contents onto the JavaFX Scene graph:

```
package javafx2introbyexample.chapter4.recipe4_02;

import java.net.URL;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

/**
 *
 * @author cdea
*/
public class DisplayHtml5Content extends Application {
    private Scene scene;
    @Override public void start(Stage stage) {
        // create the scene
        stage.setTitle("Chapter 4-2 Display Html5 Content");
        final WebView browser = new WebView();
        URL url = getClass().getResource("clock3.svg");
        browser.getEngine().load(url.toExternalForm());
        scene = new Scene(browser,590,400, Color.rgb(0, 0, 0, .80));
        stage.setScene(scene);
        stage.show();
    }
```

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

This JavaFX code will load and render HTML5 content. Assuming that you have a designer who has provided content such as HTML5, it will be your job to render assets in JavaFX. The following code represents an SVG file named `clock3.svg` that is predominantly generated by the powerful tool Inkscape, which is an illustrator tool capable of generating SVG. In the following code, notice hand-coded JavaScript code (inside the `CDATA` tag) that will position the second, minute, and hour hands of the clock based on the current time of day. Because all the logic (from setting the time to animating the hands) is inside this file, things are self contained, which means any HTML5 capable viewer can display the file's contents. So when debugging, you can easily render content in any HTML5-compliant browser. Later in this chapter, we will demonstrate JavaFX code that can interact with HTML5 content.
Shown here is a pared-down version of the SVG analog clock. (To obtain the file's source code, download the code from the book's web site.) This is an SVG analog clock created in Inkscape (`clock3.svg`):

```
<svg
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:cc="http://creativecommons.org/ns#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:svg="http://www.w3.org/2000/svg"
   xmlns="http://www.w3.org/2000/svg"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd"
   xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"
   width="300"
   height="250"
   id="svg4171"
   version="1.1"
   inkscape:version="0.48.1 "
   sodipodi:docname="clock3.svg" onload="updateTime()">

<script>

<![CDATA[
var xmlns="http://www.w3.org/2000/svg"

function updateTime()
{
  var date = new Date()

  var hr = parseInt(date.getHours())
  if (hr > 12) {
    hr = hr - 12;
  }
  var min = parseInt(date.getMinutes())
  var sec = parseInt(date.getSeconds())
  var pi=180
```

```
  var secondAngle = sec * 6 + pi
  var minuteAngle = ( min + sec / 60 ) * 6 + pi
  var hourAngle   = (hr + min / 60 + sec /3600) * 30 + pi

  moveHands(secondAngle, minuteAngle, hourAngle)
}

function moveHands(secondAngle, minuteAngle, hourAngle) {

  var secondHand = document.getElementById("secondHand")
  var minuteHand = document.getElementById("minuteHand")
  var hourHand = document.getElementById("hourHand")

  secondHand.setAttribute("transform","rotate("+ secondAngle + ")")
  minuteHand.setAttribute("transform","rotate("+ minuteAngle +")")
  hourHand.setAttribute("transform","rotate("+ hourAngle + ")")

}

]]>

</script>
<defs id="defs4173">
... // beginning of SVG code
... // Main clock code

<g id="hands" transform="translate(108,100)">
<g id="minuteHand">
<line stroke-width="3.59497285" y2="50" stroke-linecap="round" stroke="#00fff6" opacity=".9"
/>
<animateTransform attributeName="transform" type="rotate" repeatCount="indefinite" dur="60min"
by="360" />
</g>

<g id="hourHand">
<line stroke-width="5" y2="30" stroke-linecap="round" stroke="#ffcb00" opacity=".9" />
<animateTransform attributeName="transform" type="rotate" repeatCount="indefinite" dur="12h"
by="360" />
</g>
<g id="secondHand">
<line stroke-width="2" y1="-20" y2="70" stroke-linecap="round" stroke="red"/>
<animateTransform attributeName="transform" type="rotate" repeatCount="indefinite" dur="60s"
by="360" />
</g>
</g>

    ... // The rest of the Clock code: shiney glare, black button cover (center) on top of
arms


</svg>
```

Figure 4-8 depicts a JavaFX application, rendering the SVG file `clock3.svg` displaying an analog clock.

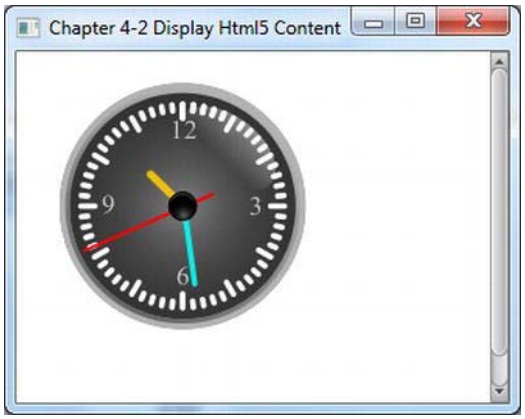

Figure 4-8. *Analog clock*

How It Works

In this recipe, you will be creating an analog clock application that will take existing HTML5 content to be rendered onto the JavaFX Scene graph. HTML5 allows the use of SVG content to be shown in browsers. SVG is similar to JavaFX's Scene graph, in which nodes can be scaled at different sizes while preserving details. To manipulate SVG or any HTML5 elements, you will be using the JavaScript language. Depicted in Figure 4-8 is a JavaFX application displaying an animated analog clock. To learn more about SVG, visit `http://www.w3schools.com/svg/default.asp`. Before running this example, make sure the `clock3.svg` file is located in the build path. In NetBeans you may need to perform a clean and build before running the application that will copy the resource (`clock3.svg`) to the build path. You may also want to manually copy the `clock3.svg` file to reside in the build path co-located where the `DisplayHtml5Content.class` file is located if you are running application on the command line.

In software development you will undoubtedly experience working with a designer where he/she will use popular tools to generate web content that will be wired up to an application's functions. To create an analog clock, I enlisted my daughter, who is quite proficient with the open-source tool Inkscape. Although Inkscape was used to generate the content for this recipe, I will not go into details regarding the tool because it is beyond the scope of this book. To learn more about Inkscape, please visit `http://www.inkscape.org` for tutorials and demos. To model the Designer and Developer Workflow, she created a cool looking clock and I added JavaScript/SVG code to move the clock's hour, minute, and second hands. Inkscape allows you to create shapes, text, and effects to generate amazing illustrations. Because SVG files are considered as HTML5 content, you will be able to display SVG drawings inside of an HTML5-capable browser. In this scenario, you will be displaying the analog clock in JavaFX's `WebView` node. You can think of a `WebView` node as a mini browser capable of loading URLs to be displayed. When loading a URL you will notice the call to `getEngine().load()` where the `getEngine()` method will return an instance of `javafx.scene.web.WebEngine` object. So, the `WebView` object is implicitly creating one `javafx.scene.web.WebEngine` object instance per `WebView` object. Shown here is the JavaFX's `WebEngine` object loading a file `clock3.svg`:

```
final WebView browser = new WebView();
URL url = getClass().getResource("clock3.svg");
```

```
browser.getEngine().load(url.toExternalForm());
```

You are probably wondering why the JavaFX source code is so small. The code is small because its job is to instantiate an instance of a `javafx.scene.web.WebView` that instantiates a `javafx.scene.web.WebEngine` class and passes a URL. After that, the `WebEngine` object does all the work by rendering HTML5 content just like any browser. When rendering the content, notice that the clock's arms move or animate; for example, the second hand rotates clockwise. Before animating the clock, you have to set the clock's initial position by calling the JavaScript `updateTime()` function via the `onload` attribute on the entire SVG document (located on the root `svg` element). Once the clock's arms are set, you will add SVG code to draw and animate by using the line and animate transform elements, respectively. Shown here is a SVG code snippet to animate the second hand indefinitely:

```
<g id="secondHand">
<line stroke-width="2" y1="-20" y2="70" stroke-linecap="round" stroke="red"/>
<animateTransform attributeName="transform" type="rotate" repeatCount="indefinite"
dur="60s" by="360" />
</g>
```

On a final note, if you want to create a clock like the one depicted in this recipe, visit `http://screencasters.heathenx.org/blog` to learn about all things Inkscape. Another impressive and beautiful display of custom controls that focuses on gauges and dials is the Steel Series by Gerrit Grunwald. To be totally amazed, visit his blog at `http://harmoniccode.blogspot.com`.

4-3. Manipulating HTML5 Content with Java Code

Problem

You are an underpaid developer, and your boss refuses to let you relocate to the cube next to the window. You must find a way to determine the weather without leaving your workspace.

Solution

Create a weather application that fetches data from Yahoo's weather service. The following code implements a weather application that retrieves Yahoo's weather information to be rendered as HTML in a JavaFX application:

```
package javafx2introbyexample.chapter4.recipe4_03;

import javafx.animation.*;
import javafx.application.Application;
import javafx.beans.property.*;
import javafx.beans.value.*;
import javafx.concurrent.Worker.State;
import javafx.scene.*;
import javafx.scene.web.*;
import javafx.stage.Stage;
import javafx.util.Duration;
import org.w3c.dom.*;
```

```
/**
 * Shows a preview of the weather and 3 day forecast
 * @author cdea
 */
public class ManipulatingHtmlContent extends Application {
    String url = "http://weather.yahooapis.com/forecastrss?p=USMD0033&u=f";
    int refreshCountdown = 60;

    @Override public void start(Stage stage) {
        // create the scene
        stage.setTitle("Chapter 4-3 Manipulating HTML content");
        Group root = new Group();
        Scene scene = new Scene(root, 460, 340);

        final WebEngine webEngine = new WebEngine(url);

        StringBuilder template = new StringBuilder();
        template.append("<head>\n");
        template.append("<style type=\"text/css\">body {background-
color:#b4c8ee;}</style>\n");
        template.append("</head>\n");
        template.append("<body id='weather_background'>");

        final String fullHtml = template.toString();

        final WebView webView = new WebView();

        IntegerProperty countDown = new SimpleIntegerProperty(refreshCountdown);
        countDown.addListener(new ChangeListener<Number>() {
            @Override
            public void changed(ObservableValue<? extends Number> observable, Number oldValue,
Number newValue){
               // when change occurs on countDown call JavaScript to update text in
HTMLwebView.getEngine().executeScript("document.getElementById('countdown').innerHTML =
'Seconds till refresh: " + newValue + "'");
                if (newValue.intValue() == 0) {
                    webEngine.reload();
                }
            }
        });
        final Timeline timeToRefresh = new Timeline();
        timeToRefresh.getKeyFrames().addAll(
                new KeyFrame(Duration.ZERO, new KeyValue(countDown, refreshCountdown)),
                new KeyFrame(Duration.seconds(refreshCountdown), new KeyValue(countDown, 0))
        );
```

```
        webEngine.getLoadWorker().stateProperty().addListener(new ChangeListener<State>() {
            @Override
            public void changed(ObservableValue<? extends State> observable, State oldValue,
State newValue){
                System.out.println("done!" + newValue.toString());
                if (newValue != State.SUCCEEDED) {
                    return;
                }
                // request 200 OK
                Weather weather = parse(webEngine.getDocument());

StringBuilder locationText = new StringBuilder();
                locationText.append("<b>")
                        .append(weather.city)
                        .append(", ")
                        .append(weather.region)
                        .append(" ")
                        .append(weather.country)
                        .append("</b><br />\n");

                String timeOfWeatherTextDiv = "<b id=\"timeOfWeatherText\">" +
weather.dateTimeStr + "</b><br />\n";
                String countdownText = "<b id=\"countdown\"></b><br />\n";
                webView.getEngine().loadContent(fullHtml + locationText.toString() +
                        timeOfWeatherTextDiv +
                        countdownText +
                        weather.htmlDescription);
                System.out.println(fullHtml + locationText.toString() +
                        timeOfWeatherTextDiv +
                        countdownText +
                        weather.htmlDescription);
                timeToRefresh.playFromStart();
            }
        });

        root.getChildren().addAll(webView);

        stage.setScene(scene);
        stage.show();
    }

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

    private static String obtainAttribute(NodeList nodeList, String attribute) {
        String attr = nodeList
                .item(0)
                .getAttributes()
                .getNamedItem(attribute)
                .getNodeValue()
                .toString();
```

```
        return attr;

    }

    private static Weather parse(Document doc) {

        NodeList currWeatherLocation =
doc.getElementsByTagNameNS("http://xml.weather.yahoo.com/ns/rss/1.0", "location");

        Weather weather = new Weather();
        weather.city = obtainAttribute(currWeatherLocation, "city");
        weather.region = obtainAttribute(currWeatherLocation, "region");
        weather.country = obtainAttribute(currWeatherLocation, "country");

        NodeList currWeatherCondition =
doc.getElementsByTagNameNS("http://xml.weather.yahoo.com/ns/rss/1.0", "condition");
        weather.dateTimeStr = obtainAttribute(currWeatherCondition, "date");
        weather.currentWeatherText = obtainAttribute(currWeatherCondition, "text");
        weather.temperature = obtainAttribute(currWeatherCondition, "temp");

        String forcast = doc.getElementsByTagName("description")
                        .item(1)
                        .getTextContent();
        weather.htmlDescription = forcast;

        return weather;
    }

}
class Weather {
    String dateTimeStr;
    String city;
    String region;
    String country;
    String currentWeatherText;
    String temperature;
    String htmlDescription;

}
```

Figure 4-9 depicts the weather application that fetches data from the Yahoo Weather service. In the third line of displayed text, you'll notice that Seconds till refresh: 31 is a countdown in seconds until the next retrieval of weather information. The actual manipulation of HTML content occurs here.

Figure 4-9. *Weather application*

The following is output to the console of the HTML that is rendered onto the `WebView` node:

```
<head>
<style type="text/css">body {background-color:#b4c8ee;}
</style>
</head>
<body id='weather_background'><b>Berlin, MD US</b><br />
<b id="timeOfWeatherText">Thu, 06 Oct 2011 8:51 pm EDT</b><br />
<b id="countdown"></b><br />

<img src="http://l.yimg.com/a/i/us/we/52/33.gif"/><br />
<b>Current Conditions:</b><br />
Fair, 49 F<BR />
<BR /><b>Forecast:</b><BR />
Thu - Clear. High: 66 Low: 48<br />
Fri - Sunny. High: 71 Low: 52<br />
<br />
<a
href="http://us.rd.yahoo.com/dailynews/rss/weather/Berlin__MD/*http://weather.yahoo.com/foreca
st/USMD0033_f.html">Full Forecast at Yahoo! Weather</a><BR/><BR/>
(provided by <a href="http://www.weather.com" >The Weather Channel</a>)<br/>
```

How It Works

In this recipe you will be creating a JavaFX application able to retrieve XML information from Yahoo's weather service. Once the XML is parsed, HTML content is assembled and rendered onto JavaFX's `WebView` node. The `WebView` object instance is a graph node capable of rendering and retrieving XML or any HTML5 content. The application will also display a countdown of the number of seconds until the next retrieval from the weather service.

When accessing weather information for your area through Yahoo's weather service, you will need to obtain a location ID or the URL to the RSS feed associated with your city. Before I explain the code line by line, I will list the steps to obtain the URL for the RSS feed of your local weather forecasts.

1. Open browser to `http://weather.yahoo.com/`.
2. Enter city or ZIP code and press Go button.
3. Click the small orange colored RSS button near the right side of the web page (under "Add weather to your website").
4. Copy and paste the URL address line in your browser to be used in the code for your weather application. For example, I used the following RSS URL web address: `http://weather.yahooapis.com/forecastrss?p=USMD0033&u=f`.

Now that you have obtained a valid RSS URL web address, let's use it in our recipe example. When creating the `ManipulatingHtmlContent` class, you will need two instance variables: `url` and `refreshCountdown`. The `url` variable will be assigned to the RSS URL web address from Step 4. The `refreshCountdown` variable of type `int` is assigned 60 to denote the time in seconds until a refresh or another retrieval of the weather information takes place.

Like all our JavaFX examples inside of the `start()` method, we begin by creating the `Scene` object for the initial main content region. Next, we create a `javafx.scene.web.WebEngine` instance by passing in the `url` into the constructor. The `WebEngine` object will asynchronously load the web content from Yahoo's weather service. Later we will discuss the callback method responsible for handling the content when the web content is done loading. The following code line will create and load a URL web address using a `WebEngine` object:

```
final WebEngine webEngine = new WebEngine(url);
```

After you create a `WebEngine` object, you will be creating an HTML document that will form as a template for later assembling when the web content is successfully loaded. Although the code contains HTML markup tags in Java code, which totally violates the principles of the separation of concerns, I inlined HTML by concatenating string values for brevity. To have a proper MVC-style separation, you may want to create a separate file containing your HTML content with substitution sections for data that will change over time. The code snippet that follows is the start of the creation of a template used to display weather information:

```
StringBuilder template = new StringBuilder();
template.append("<head>\n")
   .append("<style type=\"text/css\">body {background-color:#b4c8ee;}</style>\n")
   .append("</head>\n")
   .append("<body id='weather_background'>");
```

Once you have created your web page by concatenating strings, you will create a `WebView` object instance, which is a displayable graph node that will be responsible for rendering the web page. Remember from recipe 4-2, in which we discussed that a `WebView` will have its own instance of a `WebEngine`. Knowing this fact, we only use the `WebView` node to render the assembled HTML web page, not to retrieve the XML weather information via a URL. In other words, the `WebEngine` object is responsible for retrieving the XML from Yahoo's Weather service to be parsed and then fed into the `WebView` object to be displayed as HTML. The following code snippet instantiates a `WebView` graph node that is responsible for rendering HTML5 content:

```
final WebView webView = new WebView();
```

Next, you will create a countdown timer to refresh the weather information being displayed in the application window. First, you will instantiate an `IntegerProperty` variable, `countdown`, to hold the

number of seconds until the next refresh time. Second, you will add a change listener (`ChangeListener`) to update the HTML content dynamically using JavaFX's capability to execute JavaScript. The change listener also will determine whether the countdown has reached zero. If so, it will invoke the `webEngine`'s (`WebEngine`) `reload()` method to refresh or retrieve the weather information again. The following is the code that creates an `IntegerProperty` value to update the countdown text within the HTML using the `executeScript()` method:

```
IntegerProperty countDown = new SimpleIntegerProperty(refreshCountdown);
countDown.addListener(new ChangeListener<Number>() {
     @Override
     public void changed(ObservableValue<? extends Number> observable, Number oldValue,
Number newValue){

webView.getEngine().executeScript("document.getElementById('countdown').innerHTML =
'Seconds till refresh: " + newValue + "'");
                if (newValue.intValue() == 0) {
                    webEngine.reload();
                }
            }
}); // addListener()
```

After implementing your `ChangeListener`, you can create a `TimeLine` object to cause change on the `countdown` variable, thus triggering the `ChangeListener` to update the HTML text depicting the seconds until refresh. The follow code implements a `TimeLine` to update the `countDown` variable:

```
final Timeline timeToRefresh = new Timeline();
timeToRefresh.getKeyFrames().addAll(
    new KeyFrame(Duration.ZERO, new KeyValue(countDown, refreshCountdown)),
    new KeyFrame(Duration.seconds(refreshCountdown), new KeyValue(countDown, 0))
);
```

In summary, the rest of the code creates a `ChangeListener` that responds to a `State.SUCCEEDED`. Once the `webEngine` (`WebEngine`) has finished retrieving the XML, the change listener (`ChangeListener`) is responsible for parsing and rendering the assembled web page into the `webView` node. The following code parses and displays the weather data by calling the `loadContent()` method on the `WebView`'s `WebEngine` instance:

```
            if (newValue != State.SUCCEEDED) {
                return;
            }
            Weather weather = parse(webEngine.getDocument());

            ...// the rest of the inlined HTML

            String countdownText = "<b id=\"countdown\"></b><br />\n";
            webView.getEngine().loadContent(fullHtml + location.toString() +
                    timeOfWeatherTextDiv +
                    countdownText +
                    weather.htmlDescription);
```

To parse the XML returned by the `webEngine`'s `getDocument()` method, you will interrogate the `org.w3c.dom.Document` object. For convenience, I created a `parse()` method to walk the DOM to obtain

weather data and return as a `Weather` object. See Javadocs and Yahoo's RSS XML Schema for more information on data elements returned from weather service.

4-4. Responding to HTML Events

Problem

You begin to feel sorry for your other cube mates who are also oblivious to the outside world. A storm is approaching and you want to let them know to take their umbrella before leaving the building.

Solution

Add a Panic Button to your weather application that will simulate an e-mail notification. A Calm Down button is also added to retract the warning message.

The following code implements the weather application with additional buttons to warn and disregard a warning of impending stormy weather:

```
    @Override public void start(Stage stage) {


...  // template building
```

This code will add HTML buttons with the `onclick` attributes set to invoke the JavaScript `alert` function:

```
        template.append("<body id='weather_background'>");
        template.append("<form>\n");
        template.append("  <input type=\"button\" onclick=\"alert('warning')\" value=\"Panic
Button\" />\n");
        template.append("  <input type=\"button\" onclick=\"alert('unwarning')\" value=\"Calm
down\" />\n");
        template.append("</form>\n");
```

The following code is added to the `start()` method to create the warning message with opacity set as zero to be invisible:

```
        // calls the createMessage() method to build warning message
        final Text warningMessage = createMessage(Color.RED, "warning: ");
        warningMessage.setOpacity(0);

       ... // Countdown code
```

Continuing inside of the `start()` method, this code section is added to update the warning message after weather information was retrieved successfully:

```
        webEngine.getLoadWorker().stateProperty().addListener(new ChangeListener<State>() {
            public void changed(ObservableValue<? extends State> observable, State oldValue,
State newValue){
                System.out.println("done!" + newValue.toString());
                if (newValue != State.SUCCEEDED) {
                    return;
                }
```

```
                Weather weather = parse(webEngine.getDocument());
                warningMessage.setText("Warning: " + weather.currentWeatherText + "\nTemp: " +
weather.temperature + "\n E-mailed others");

                ... // the rest of changed() method
        }); // end of addListener method
```

This code sets the `OnAlert` property, which is an event handler to respond when a the Panic or Calm Down button is pressed:

```
        webView.getEngine().setOnAlert(new EventHandler<WebEvent<String>>(){
            public void handle(WebEvent<String> evt) {
                warningMessage.setOpacity("warning".equalsIgnoreCase(evt.getData()) ? 1d :
0d);
            }
        }); // end of setOnAlert() method.

        root.getChildren().addAll(webView, warningMessage);

        stage.setScene(scene);
        stage.show();

    } // end of start() method
```

The following method is code that you will add as a private method that is responsible for creating a text node (`javafx.scene.text.Text`) to be used as the warning message when the user presses the Panic Button:

```
    private Text createMessage(Color color, String message) {
        DropShadow dShadow = DropShadowBuilder.create()
                             .offsetX(3.5f)
                             .offsetY(3.5f)
                             .build();
        Text textMessage = TextBuilder.create()
                    .text(message)
                    .x(100)
                    .y(50)
                    .strokeWidth(2)
                    .stroke(Color.WHITE)
                    .effect(dShadow)
                    .fill(color)
                    .font(Font.font(null, FontWeight.BOLD, 35))
                    .translateY(50)
                    .build();
        return textMessage;
    }
} // end of the RespondingToHtmlEvents class
```

Figure 4-10 shows our weather application displaying a warning message after the Panic Button has been pressed. To remove the warning message, you can press the Calm Down button.

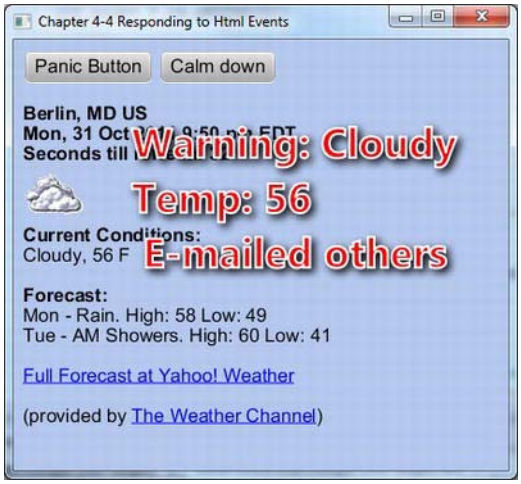

Figure 4-10. *Weather application displaying warning message*

How It Works

In this recipe you will add additional features to the weather application (from recipe 4-3) that responds to HTML events. The application you will be creating is similar to the previous recipe, except you will be adding HTML buttons on the web page to be rendered onto the `WebView` node. The first button added is the Panic Button that, when pressed, displays a warning message stating the current weather condition and a simulated e-mail notification to your cube mates. To retract the warning message you will also add a Calm Down button.

■ **Note** Because the code is so similar to the previous recipe, I will point out the additions to the source code without going into great detail.

To add the buttons, you will use the HTML tag `<input type="button"…>` with an `onclick` attribute set to use JavaScript's `alert()` function to notify JavaFX of an alert event. Shown here are the two buttons added to the web page:

```
        StringBuilder template = new StringBuilder();
        ...// Header part of HTML Web page
        template.append("<form>\n");
        template.append("  <input type=\"button\" onclick=\"alert('warning')\" value=\"Panic
Button\" />\n");
        template.append("  <input type=\"button\" onclick=\"alert('unwarning')\" value=\"Calm
down\" />\n");
        template.append("</form>\n");
```

When the web page renders allowing you to press the buttons, the `onclick` attribute will call JavaScript's `alert()` function that contains a string message. When the `alert()` function is invoked, the web page's owning parent (the `webView`'s`WebEngine` instance) will be notified of the alert via the `WebEngine`'s `OnAlert` attribute. To respond to JavaScript's alerts, you will add an event handler (`EventHandler`) to respond to `WebEvent` objects. In the `handle()` method, you will simply show and hide the warning message by toggling the opacity of the `warningMessage` node (`javafx.scene.text.Text`).

The following code snippet toggles the opacity of the warning message based on comparing the event's data (`evt.getData()`) that contains the string passed in from the JavaScript's `alert()` function. So, if the message is "warning," the `warningMessage` opacity is set to 1; otherwise, set to 0 (both of type `double`).

```
webView.getEngine().setOnAlert(new EventHandler<WebEvent<String>>(){
    public void handle(WebEvent<String> evt) {
       warningMessage.setOpacity("warning".equalsIgnoreCase(evt.getData()) ? 1d : 0d);
    }
});
```

Please see the Javadocs for additional HTML web events (`WebEvent`).

4-5. Displaying Content from the Database

Problem

You want to keep up on the latest news monitoring the local legislature and science regarding the detrimental effects of the lack of light in small cubical work areas.

Solution

Create a JavaFX RSS reader. The RSS feed location URLs will be stored in a database to be later retrieved. Listed here are the main classes used in this recipe:

- `javafx.scene.control.Hyperlink`
- `javafx.scene.web.WebEngine`
- `javafx.scene.web.WebView`
- `org.w3c.dom.Document`
- `org.w3c.dom.Node`
- `org.w3c.dom.NodeList`

This recipe will be using an embedded database called Derby from the Apache group at `http://www.apache.org`. As a requirement, you will need to download the Derby software. To download the software, visit `http://db.apache.org/derby/derby_downloads.html` to download the latest version containing the libraries. Once downloaded, you can unzip or untar into a directory. To compile and run this recipe, you will need to update the classpath in your IDE or environment variable to point to Derby libraries (`derby.jar` and `derbytools.jar`). You can type a valid RSS URL into the text field when running the example code, and then hit the enter key to load your new RSS headlines. After loading is complete,

the headline news is listed to the upper right frame region. Next, you will have an opportunity to choose a headline news article to read fully by clicking on a view button beneath it.

The following code implements an RSS reader in JavaFX:

```
package javafx2introbyexample.chapter4.recipe4_05;

import java.util.*;
import javafx.application.Application;
import javafx.beans.value.*;
import javafx.collections.ObservableList;
import javafx.concurrent.Worker.State;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.input.*;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.web.*;
import javafx.stage.Stage;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/**
 * Display Contents From Database
 * @author cdea
 */
public class DisplayContentsFromDatabase extends Application {

    @Override public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 640, 480, Color.WHITE);
        final Map<String, Hyperlink> hyperLinksMap = new TreeMap<>();

        final WebView newsBrief = new WebView(); // upper right
        final WebEngine webEngine = new WebEngine();
        final WebView websiteView = new WebView(); // lower right

        webEngine.getLoadWorker().stateProperty().addListener(new ChangeListener<State>() {
            public void changed(ObservableValue<? extends State> observable, State oldValue,
State newValue){
                if (newValue != State.SUCCEEDED) {
                    return;
                }

                RssFeed rssFeed = parse(webEngine.getDocument(), webEngine.getLocation());

                hyperLinksMap.get(webEngine.getLocation()).setText(rssFeed.channelTitle);

                // print feed info:
                StringBuilder rssSource = new StringBuilder();
```

```
                rssSource.append("<head>\n")
                        .append("</head>\n")
                        .append("<body>\n");
                rssSource.append("<b>")
                         .append(rssFeed.channelTitle)
                         .append(" (")
                         .append(rssFeed.news.size())
                         .append(")")
                         .append("</b><br />\n");
                 StringBuilder htmlArticleSb = new StringBuilder();
                for (NewsArticle article:rssFeed.news) {

                    htmlArticleSb.append("<hr />\n")
                         .append("<b>\n")
                         .append(article.title)
                         .append("</b><br />")
                         .append(article.pubDate)
                         .append("<br />")
                         .append(article.description)
                         .append("<br />\n")
                         .append("<input type=\"button\" onclick=\"alert('")
                            .append(article.link)
                            .append("')\" value=\"View\" />\n");
                }

                String content = rssSource.toString() + "<form>\n" + htmlArticleSb.toString()
+ "</form></body>\n";
                System.out.println(content);
                newsBrief.getEngine().loadContent(content);
                // write to disk if not already.
                DBUtils.saveRssFeed(rssFeed);
            }
        }); // end of webEngine addListener()

        newsBrief.getEngine().setOnAlert(new EventHandler<WebEvent<String>>(){
            public void handle(WebEvent<String> evt) {
                websiteView.getEngine().load(evt.getData());
            }
        }); // end of newsBrief setOnAlert()

        // Left and right split pane
        SplitPane splitPane = new SplitPane();
        splitPane.prefWidthProperty().bind(scene.widthProperty());
        splitPane.prefHeightProperty().bind(scene.heightProperty());

        final VBox leftArea = new VBox(10);
        final TextField urlField = new TextField();
        urlField.setOnAction(new EventHandler<ActionEvent>(){
            public void handle(ActionEvent ae){
                String url = urlField.getText();
                final Hyperlink jfxHyperLink = createHyperLink(url, webEngine);
                hyperLinksMap.put(url, jfxHyperLink);
```

```
            HBox rowBox = new HBox(20);
            rowBox.getChildren().add(jfxHyperLink);
            leftArea.getChildren().add(rowBox);
            webEngine.load(url);
            urlField.setText("");
        }
    }); // end of urlField setOnAction()

    leftArea.getChildren().add(urlField);

    List<RssFeed> rssFeeds = DBUtils.loadFeeds();
    for (RssFeed feed:rssFeeds) {
        HBox rowBox = new HBox(20);
        final Hyperlink jfxHyperLink = new Hyperlink(feed.channelTitle);
        jfxHyperLink.setUserData(feed);
        final String location = feed.link;
        hyperLinksMap.put(feed.link, jfxHyperLink);
        jfxHyperLink.setOnAction(new EventHandler<ActionEvent>() {
                public void handle(ActionEvent evt) {
                    webEngine.load(location);
                }
            }
        );
        rowBox.getChildren().add(jfxHyperLink);
        leftArea.getChildren().add(rowBox);

    } // end of for loop


    // Dragging over surface
    scene.setOnDragOver(new EventHandler<DragEvent>() {
        @Override
        public void handle(DragEvent event) {
            Dragboard db = event.getDragboard();
            if (db.hasUrl()) {
                event.acceptTransferModes(TransferMode.COPY);
            } else {
                event.consume();
            }
        }
    });  // end of scene.setOnDragOver()

    // Dropping over surface
    scene.setOnDragDropped(new EventHandler<DragEvent>() {

        @Override
        public void handle(DragEvent event) {
            Dragboard db = event.getDragboard();
            boolean success = false;
            HBox rowBox = new HBox(20);
            if (db.hasUrl()) {
                if (!hyperLinksMap.containsKey(db.getUrl())) {
```

```
                        final Hyperlink jfxHyperLink = createHyperLink(db.getUrl(),
webEngine);
                        hyperLinksMap.put(db.getUrl(), jfxHyperLink);
                        rowBox.getChildren().add(jfxHyperLink);
                        leftArea.getChildren().add(rowBox);
                    }
                    webEngine.load(db.getUrl());
                }
                event.setDropCompleted(success);
                event.consume();
            }
        });  // end of scene.setOnDragDropped()

        leftArea.setAlignment(Pos.TOP_LEFT);

        // Upper and lower split pane
        SplitPane splitPane2 = new SplitPane();
        splitPane2.setOrientation(Orientation.VERTICAL);
        splitPane2.prefWidthProperty().bind(scene.widthProperty());
        splitPane2.prefHeightProperty().bind(scene.heightProperty());

        HBox centerArea = new HBox();

        centerArea.getChildren().add(newsBrief);

        HBox rightArea = new HBox();

        rightArea.getChildren().add(websiteView);

        splitPane2.getItems().add(centerArea);
        splitPane2.getItems().add(rightArea);

        // add left area
        splitPane.getItems().add(leftArea);

        // add right area
        splitPane.getItems().add(splitPane2);
        newsBrief.prefWidthProperty().bind(scene.widthProperty());
        websiteView.prefWidthProperty().bind(scene.widthProperty());
        // evenly position divider
        ObservableList<SplitPane.Divider> dividers = splitPane.getDividers();
        for (int i = 0; i < dividers.size(); i++) {
            dividers.get(i).setPosition((i + 1.0) / 3);
        }

        HBox hbox = new HBox();
        hbox.getChildren().add(splitPane);
        root.getChildren().add(hbox);

        stage.setScene(scene);
        stage.show();
```

```
    } // end of start()

    private static RssFeed parse(Document doc, String location) {

        RssFeed rssFeed = new RssFeed();
        rssFeed.link = location;

        rssFeed.channelTitle = doc.getElementsByTagName("title")
             .item(0)
             .getTextContent();

        NodeList items = doc.getElementsByTagName("item");
        for (int i=0; i<items.getLength(); i++){
            Map<String, String> childElements = new HashMap<>();
            NewsArticle article = new NewsArticle();
            for (int j=0; j<items.item(i).getChildNodes().getLength(); j++) {
                Node node = items.item(i).getChildNodes().item(j);
                childElements.put(node.getNodeName().toLowerCase(), node.getTextContent());
            }
            article.title = childElements.get("title");
            article.description = childElements.get("description");
            article.link = childElements.get("link");
            article.pubDate = childElements.get("pubdate");

            rssFeed.news.add(article);
        }

        return rssFeed;
    } // end of parse()

    private Hyperlink createHyperLink(String url, final WebEngine webEngine) {
        final Hyperlink jfxHyperLink = new Hyperlink("Loading News...");
        RssFeed aFeed = new RssFeed();
        aFeed.link = url;
        jfxHyperLink.setUserData(aFeed);
        jfxHyperLink.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent evt) {
                RssFeed rssFeed = (RssFeed)jfxHyperLink.getUserData();
                webEngine.load(rssFeed.link);
            }
        });
        return jfxHyperLink;
    }

    public static void main(String[] args){
        DBUtils.setupDb();
        Application.launch(args);
    }

}  // end of createHyperLink()
```

```
class RssFeed {
    int id;
    String channelTitle = "News...";
    String link;
    List<NewsArticle> news = new ArrayList<>();

    public String toString() {
        return "RssFeed{" + "id=" + id + ", channelTitle=" + channelTitle + ", link=" + link +
", news=" + news + '}';
    }

    public RssFeed() {
    }
    public RssFeed(String title, String link) {
        this.channelTitle = title;
        this.link = link;
    }
}

class NewsArticle {
    String title;
    String description;
    String link;
    String pubDate;

    public String toString() {
        return "NewsArticle{" + "title=" + title + ", description=" + description + ", link="
+ link + ", pubDate=" + pubDate + ", enclosure=" + '}';
    }

}
```

The following code is an exerpt from DBUtils.java. The code shows the `saveRssFeed()` method, which is responsible for persisting urls in the RSS feed:

```
    public static int saveRssFeed(RssFeed rssFeed) {
        int pk = rssFeed.link.hashCode();

        loadDriver();

        Connection conn = null;
        ArrayList statements = new ArrayList();
        PreparedStatement psInsert = null;
        Statement s = null;
        ResultSet rs = null;
        try {

            // database name
            String dbName = "demoDB";

            conn = DriverManager.getConnection(protocol + dbName
                    + ";create=true", props);
```

```
            rs = conn.createStatement().executeQuery("select count(id) from rssFeed where id =
" + rssFeed.link.hashCode());

            rs.next();
            int count = rs.getInt(1);

            if (count == 0) {

                // handle transaction
                conn.setAutoCommit(false);

                s = conn.createStatement();
                statements.add(s);

                psInsert = conn.prepareStatement("insert into rssFeed values (?, ?, ?)");
                statements.add(psInsert);
                psInsert.setInt(1, pk);
                String escapeTitle = rssFeed.channelTitle.replaceAll("\'", "''");
                psInsert.setString(2, escapeTitle);
                psInsert.setString(3, rssFeed.link);
                psInsert.executeUpdate();
                conn.commit();
                System.out.println("Inserted " + rssFeed.channelTitle + " " + rssFeed.link);
                System.out.println("Committed the transaction");
            }
            shutdown();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        } finally {
            // release all open resources to avoid unnecessary memory usage
            close(rs);
```

```
            // Statements and PreparedStatements
            int i = 0;
            while (!statements.isEmpty()) {
                // PreparedStatement extend Statement
                Statement st = (Statement) statements.remove(i);
                close(st);
            }

            //Connection
            close(conn);

        }

        return pk;
    } // end of saveRssFeed()
```

In Figure 4-11, our JavaFX reader displays three frames. The left column shows the RSS feed sources as hyperlinks. A text field at the top allows the user to enter urls for new sources, which then show up in the list underneath. The upper-right frame contains the headline, an excerpt of the article, and a view button that renders the article's web page in the bottom frame (lower-right region).

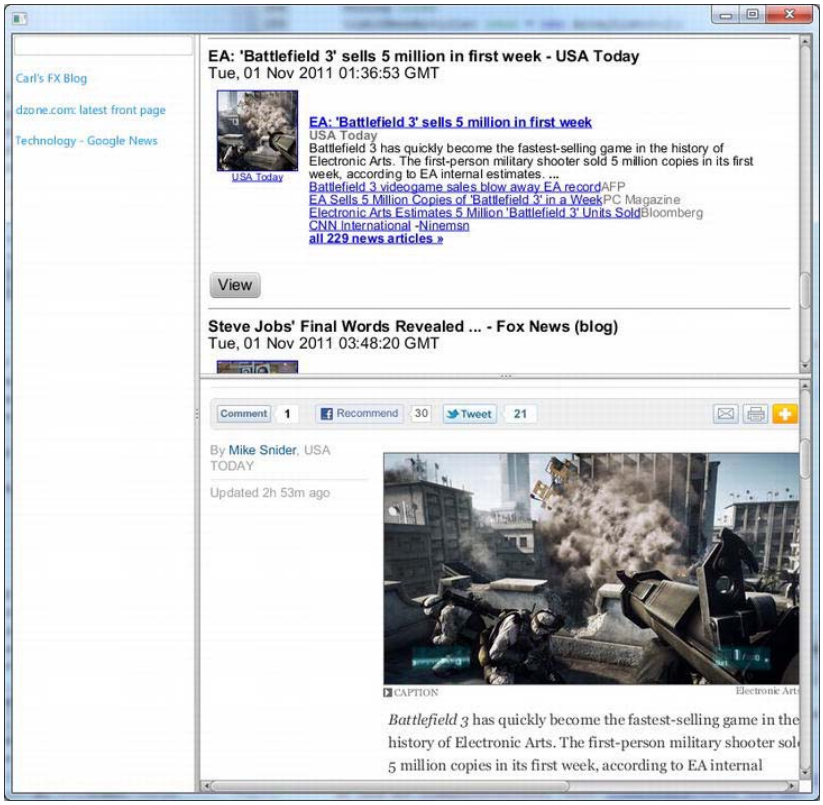

Figure 4-11. *JavaFX RSS reader*

Shown here is an example of output of the HTML to be rendered in the new headlines region (upper-right frame). You will also see the html view button responsible for notifying the application to load and render the entire article in the lower right frame region:

```
<head>
</head>
<body>
<b>Carl's FX Blog (10)</b><br />
<form>
<hr />
<b>
JavaFX Forms Framework Part 2</b><br />Mon, 03 Aug 2009 18:36:02 +0000<br />Introduction This
is the second installment of a series of blog entries relating to a proof of concept for a
JavaFX Forms Framework. Before I specify the requirements and a simple design of the FXForms
Framework, I want to follow-up on comments about tough issues relating to enterprise
application development and JavaFX. If you recall [...]<img alt="" border="0"
src="http://stats.wordpress.com/b.gif?host=carlfx.wordpress.com&amp;blog=6443320&amp;post=339&
amp;subd=carlfx&amp;ref=&amp;feed=1" width="1" height="1" /><br />
<input type="button" onclick="alert('http://carlfx.wordpress.com/2009/08/03/javafx-forms-
framework-part-2/')" value="View" />

... // the rest of the headlines

</form></body>
```

How It Works

To create an RSS reader, you will need to store feed locations for later reading. When adding a new RSS feed, you will want to locate the little orange iconic button and drag the URL address line into your JavaFX RSS reader application. I find that the drag metaphor works on my FireFox browser. However, I've provided a text field to allow you to cut-and-paste the URL. Enter a URL and press the enter key to initiate the loading of the headline news for that URL's feed. For example you can visit Google's technology news RSS feed at:
http://news.google.com/news?pz=1&cf=all&ned=us&hl=en&topic=tc&output=rss

Figure 4-12 depicts the orange RSS icon in the upper left.

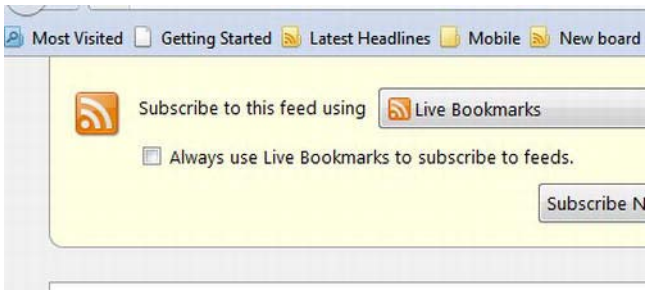

Figure 4-12. *RSS icon*

Either drag-and-drop the URL, or type it into the text field. The JavaFX RSS reader application will the save the URL location to a database. The RSS application consists of three frame regions: the RSS feed title column (left), headline news (upper right), and web site view (lower right). To display the news headlines, click the hyperlinks to the left. To show the entire article in the lower-right frame, click the View button below the headline in the upper-right frame. Before running the code, the application will require the jar libraries `derby.jar` and `derbytools.jar` included into your project classpath. These libraries allow you to save RSS URLs to an embedded JDBC database.

Similar to what you did in recipe 4-3, you retrieve news information from the Internet. The RSS retrieved will be using version 2.0. RSS is an XML standard providing really simple syndication, thus the acronym RSS. Now enough with the acronyms; let's jump into the code, shall we?

In our `start()` method, you will create a 640 by 480 white scene display area. Next, you will create a map (`TreeMap`) containing `Hyperlink` objects as values and keys representing the URL location (`String`) to the RSS feed. As before when displaying HTML content, you will need to create `WebViews`. Here you will create two `WebViews` and one `WebEngine`. The two `WebViews` will render HTML for the news headline frame region and the viewing of the entire article region (lower right). The single `WebEngine` is responsible for retrieving the RSS feed when the user clicks the left frame region containing the RSS hyperlinks.

To support the feature that allows the user to enter an RSS feed, you will need to create a text field that is able to save and render the headline news. Following is the code snippet to save an RSS URL and to add its address as a new hyperlink to the list of feeds:

```
final VBox leftArea = new VBox(10);
final TextField urlField = new TextField();
urlField.setOnAction(new EventHandler<ActionEvent>(){
    public void handle(ActionEvent ae){
        String url = urlField.getText();
        final Hyperlink jfxHyperLink = createHyperLink(url, webEngine);
        hyperLinksMap.put(url, jfxHyperLink);
        HBox rowBox = new HBox(20);
        rowBox.getChildren().add(jfxHyperLink);
        leftArea.getChildren().add(rowBox);
        webEngine.load(url);
        urlField.setText("");
    }
}); // end of urlField setOnAction()
```

News retrieval is initiated when a user clicks on a hyperlink. Once a successful retrieve has occurred on the `webEngine` (`WebEngine`) object, you will need to add a `ChangeListener` instance to respond when the state property changes to `State.SUCCEEDED`. With a valid state of `State.SUCCEEDED`, you will begin to parse the XML DOM returned from the `WebEngine`'s `getDocument()` method. Again, I provided a convenience method called `parse()` to interrogate the `Document` object representing the RSS news information.

```
RssFeed rssFeed = parse(webEngine.getDocument(), webEngine.getLocation());
```

Next, you will create an HTML page that will list the channel tile and the number of total news headlines returned. After creating the HTML to display the RSS channel title and number of articles, you will iterate over all the news headlines to build record sets or rows. Each row will contain an HTML button labeled View to notify the `WebEngine` object of an alert containing the URL of the article. When the `WebEngine` object is notified, the `OnAlert` property will contain an event handler to render the entire article in the frame in the lower-right split region. After the web page is assembled, you will call the `newsBrief` object's `getEngine().loadContent()` method to render the page. Once rendered you will save

the URL rss Feed (RssFeed) object to the database by invoking the DBUtils.saveRssFeed(rssFeed). As a convenience, the saveRssFeed() method will check for duplicates.

■ **Note:** To see the full source code relating to database persistence (DBUtils.java), please visit the book's catalog page at http://www.apress.com/9781430242574. From there, you can download the example code for the book.

The following code loads the web page to be rendered and saves the newly added rssFeed URL:

```
newsBrief.getEngine().loadContent(content);
// write to disk if not already.
DBUtils.saveRssFeed(rssFeed);
```

As in the previous recipes, you will be responding to HTML WebEvents when the new headline View button is pressed, which calls a JavaScript's alert() function. Shown following is the code snippet to handle a web event (WebEvent) containing a string of the URL that links to the entire article to be viewed in the frame to the lower right region:

```
newsBrief.getEngine().setOnAlert(new EventHandler<WebEvent<String>>(){
   public void handle(WebEvent<String> evt) {
      websiteView.getEngine().load(evt.getData());
   }
});
```

When creating the headlines region (upper right) containing HTML buttons to render the article's web page, you will notice the alert() function containing the URL to be loaded and rendered in the lower bottom split frame region. Shown following is an example of HTML generated for an headline news containing a View button that can notify the web engine's OnAlert web event (WebEvent).

```
<input type="button" onclick="alert('http://carlfx.wordpress.com/2009/08/03/javafx-forms-
framework-part-2/')" value="View" />
```

One last thing to point out is that the RSS application has missing features. One feature that comes to my mind is the ability to delete individual RSS hyperlinks on the left column region. A workaround is to remove all links by deleting the database on the file system. Because Derby is an embedded database, you can delete the directory containing the database. The JavaFX RSS application will re-create an empty database if one doesn't exist. Hopefully, you can add new features to enhance this fun and useful application.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z

JavaFX 2.0: Introduction by Example

Carl Dea

Apress®

JavaFX 2.0: Introduction by Example

ISBN-13 (pbk): 978-1-4302-4257-4

ISBN-13 (electronic): 978-1-4302-4258-1

President and Publisher: Paul Manning
Lead Editor: Jonathan Gennick
Technical Reviewer: David Coffin
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Annie Beck
Copy Editor: Nancy Sixsmith
Compositor: Bytheway Publishing Services
Indexer: BIM Indexing & Proofreading Services
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

Contents

About the Author

Carl P. Dea is currently a Software Engineer working for BCT-LLC on projects with high performance computing (HPC) architectures. He has been developing software for 15 years with many clients from Fortune 500 companies to nonprofit organizations. He has written software ranging from mission-critical applications to web applications. Carl has been using Java since the very beginning and he also is a huge JavaFX enthusiast dating back when it used to be called F3. His passion for software development started when his middle school science teacher had shown him the TRS-80 computer. Carl's current software development interests are rich client applications, game programming, Arduino, mobile phones and tablet computers. When he's not working, he and his wife love to watch their daughters perform at gymnastics meets. Carl lives on the East Coast in Pasadena (aka "The Dena"), Maryland, USA.

About the Technical Reviewer

■ **David Coffin** is the author of *Expert Oracle* and *Java Security* from Apress. He is an IT analyst working at the Savannah River Site, a large Department of Energy facility. For more than 30 years, David's expertise has been in multiplatform network integration and systems programming. Before coming to the Savannah River Site, he worked for several defense contractors and served as the technical lead for office and network computing at the National Aerospace Plane Joint Program Office at Wright-Patterson Air Force Base in Ohio. As a perpetual student, he has one master's degree and many hours toward another. As a family man, David has raised eight children. He is a triathlete and distance swimmer who competes in the middle of the pack. He is also a classical guitar player, but he's not quitting his day job.

Acknowledgments

I would like to thank my wife, Tracey, my daughters, Caitlin and Gillian, for their loving support and sacrifices. A special thanks to my daughter Caitlin, who helped with illustrations and brainstorming fun examples. A big thanks to Jim Weaver for recommending me to this project and being so encouraging. I would also thank Josh Juneau for his advice and guidance throughout this journey.

Thanks also to David Coffin with his uncanny ability to know my intentions and provide great feedback. I want to thank the wonderful people at Apress for their professionalism. A special thanks to Jonathan Gennick for believing in me and whipping me into shape.

Thanks to Annie Beck for keeping me on track when things got rough.

Thanks to all who follow me on Twitter especially the ones related to Java Swing, RIA, and JavaFX. Also, thanks to the authors (Jim Weaver, Weiqi Gao, Stephen Chin, and Dean Iverson) of the Pro JavaFX book for allowing me to tech review chapters. Another thanks to Stephen Chin and Keith Combs for heading up the awesome JavaFX User Group.Most importantly all the past JavaFX 1.x book authors that helped inspire me. Thanks goes out to my employer BCT-LLC and their customers who make a lot of things possible. Lastly, I want to give a big kudos and acknowledgment to the people at Oracle who helped me (directly or indirectly) as JavaFX 2.0 was being released: Jonathan Giles, Jasper Potts, Michael Heinrichs, Richard Bair, Amy Fowler, David DeHaven, Nicolas Lorain, Kevin Rushforth, Sheila Cepero, Gail Chappell, Cindy Castillo, Scott Hommel, Joni Gordon, Alexander Kouznetsov, Irina Fedortsova, Dmitry Kostovarov, Alla Redko, Igor Nekrestyanov, Nancy Hildebrandt, and all the Java, JavaFX, and NetBeans teams involved. Whether, then, you eat or drink or whatever you do, do all to the glory of God (1 Corinthians 10:31).