

THE EXPERT'S VOICE® IN JAVA

SECOND EDITION

JavaFX 8

Introduction by Example

*RAPIDLY DEVELOP RICH-CLIENT
APPLICATIONS THAT RUN ON
ANY PLATFORM*

Carl Dea, Mark Heckler, Gerrit Grunwald, Jose Pereda Ph.D, Sean M Phillips

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.



Contents at a Glance

About the Authors.....	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Introduction	xxiii
■ Chapter 1: Getting Started	1
■ Chapter 2: JavaFX Fundamentals	31
■ Chapter 3: Lambdas and Properties	61
■ Chapter 4: Layouts and UI Controls	91
■ Chapter 5: Graphics with JavaFX	123
■ Chapter 6: Custom UIs	151
■ Chapter 7: Media with JavaFX.....	183
■ Chapter 8: JavaFX on the Web.....	215
■ Chapter 9: JavaFX 3D	241
■ Chapter 10: JavaFX and Arduino	269

■ Chapter 11: JavaFX on the Raspberry Pi.....	305
■ Chapter 12: JavaFX and Gestures.....	339
■ Chapter 13: References	365
Index.....	383



Introduction

Welcome to *JavaFX 8: Introduction by Example*.

What is JavaFX?

JavaFX is Java's next-generation graphical user interface (GUI) toolkit that allows 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 8 is a pure Java language application programming interface (API). The goal of JavaFX is to be used across many types of devices, such as embedded devices, smartphones, TVs, tablet computers, and desktops.

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

The industry is moving toward multi-core/multi-threading platforms with GPUs. JavaFX 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

Before the creation of JavaFX, the development of rich client-side applications 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 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 GUI toolkits. JavaFX 8 is a pure Java (language) API that allows developers to leverage existing Java libraries and tools.



Figure FM-1. JavaFX

Depending on who you talk to, you are likely to encounter different definitions of “user experience” (or in the UI world, UX). But one fact 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, the same person) to create functional yet aesthetically 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 (we’re all about the user of the API’s perspective).

Although this book doesn’t go through an exhaustive study of all of JavaFX 8’s capabilities, you will find common use cases that can help you build richer applications. Hopefully, this book will lead you in the right direction by providing practical, real-world examples.

Some History

In 2005 Sun Microsystems acquired the company SeeBeyond, where a software engineer named 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, which included its plans to phase out the JavaFX scripting language and re-create the JavaFX platform for the Java platform as Java-based APIs. As promised based on the 2010 roadmap, JavaFX 2.0 SDK was released at JavaOne, in October 2011. In addition to the release of JavaFX 2.0, Oracle announced its commitment to take steps to open-source JavaFX, 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.

Between JavaFX 2.1 and 2.2 the number of new features grew rapidly. Please refer to Table FM-1 for the numerous features included between versions 2.1 and 2.2. JavaFX 2.1 was the official release of the Java SDK on a MacOS. JavaFX 2.2 was the official release of the Java SDK on a Linux operating system.

The new Java 8 release was announced March 18, 2014. Java 8 has many new APIs and language enhancements, which includes lambdas, Stream API, Nashorn JavaScript engine, and JavaFX APIs. Relating to JavaFX 8, the following are such as: 3D graphics, Rich text support, and Printing APIs. To see all of the new features in Java 8 visit <http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>.

Table FM-1. Historical Timeline of Major JavaFX Releases

Release Date	Version	Platform	Description
December 4th 2008	1.0	Windows and MacOS	JavaFX Script Language, Production Suite, Media Playback
February 12th 2009	1.1	Windows and MacOS	New mobile development
June 2nd 2009	1.2	Windows, MacOS, Linux, Solaris	Skinnable UI controls, Charting API, and Performance improvements
April 22nd 2010	1.3	Windows, MacOS, Linux, Solaris	JavaFX Composer, TV Emulator, Mobile Emulator
October 3rd 2011	2.0	Windows	Rewritten for the Java Language
April 27, 2012	2.1	Windows, MacOS	The official version for the MacOS platform was released. Media supports H.264/MPEG-4 AVC. Supports a JavaScript bridge for web engine.
August 14, 2012	2.2	Windows, MacOS and Linux	The official version for the Linux platform was released. The following are the new APIs and tools: Canvas, HTTP Live Streaming, Touch events, Gestures, Image manipulation APIs and native application packaging.
March 18, 2014	8.0	Windows, MacOS and Linux	JavaFX 8 supports the following APIs: 3D graphics, Rich text support, Printing APIs and a JVM/JDK for embedded systems.

What You Will Learn in This Book

The title of the book says it all: *JavaFX 8: Introduction by Example*. In this book you will be learning the new JavaFX 8 capabilities by following practical examples. These examples will, in turn, provide you with the knowledge needed to create your own rich client applications. Following Java's mantra "Write once, run anywhere," JavaFX also preserves this same sentiment. Because JavaFX 8 is written entirely in Java the language, you will feel right at home.

Most of the examples can be compiled and run under Java 7. However, most of them are designed to take advantage of Java 8's language enhancements, and therefore Java 8 will be required. An example of a new Java 8 language feature used throughout is called *lambda expressions* and is

discussed in Chapter 3. While working through this book with JavaFX 8 and Java 8, you will see that the new APIs and language enhancements will help you to become a more productive developer. Having said this, we encourage you to explore all of the new Java 8 capabilities.

This book covers JavaFX 8's fundamentals, lambdas, properties, layouts, UI controls, Graphics, Animation, Custom UIs, Charts, Media, Web, 3D, Arduino, RaspberryPi, Touch events, and gestures.

Who This Book Is For

If you are a Java developer looking 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. Also, if you are an experienced Java Swing, Flash/Flex, SWT or web developer who wants to learn how to create high-performance rich client-side applications, this is the book for you.

How This Book Is Structured

This book is arranged in a natural progression from beginner to intermediate and advanced level concepts. For the Java developer, none of the concepts mentioned in this book should be extremely difficult to figure out. This book is purely example-based and discusses major concepts before demonstrating applications. For each project, after showing the output from executing the code, we'll provide a detailed explanation walking through the code. Each example can be easily adapted to meet your own needs when developing a game, media player, or your usual enterprise application. The more experienced a Java UI developer you are, the more freedom you'll have to jump around to different chapters and examples throughout the book. However, any Java developer can progress through the book and learn the skills needed to enhance their everyday GUI applications.

Getting Started

To get started using this book to learn JavaFX 8, you will want to set up your development environment to compile and run many of its examples. In this chapter you will learn how to install the required software, such as the Java development kit (JDK) and the NetBeans integrated development environment (IDE). After installing the required software, you will begin by creating a traditional JavaFX “Hello World” example. Once you are comfortable with the development environment, we will walk through the JavaFX Hello World source code. Finally, you will get a glimpse of how to package your application as a standalone application to be launched and distributed.

I realize many developers are partial to their IDEs and editors, so I’ve provided two methods to compile and run the Hello World example. In the first method you will be using the Netbeans IDE and in the second method you will learn how to use the command line (terminal window) to compile and launch JavaFX applications. The second method is an approach for those who are not fond of fancy IDEs.

If you are already familiar with the installation of the Java Development Kit (JDK) and the NetBeans IDE, you can skip to Chapter 2, which covers the fundamentals (JavaFX 2D). Let’s get started!

Installing Required Software

When using this book you’ll need to download and install Java 8 Java Development Kit (JDK) or a later version. Get it from the following URL:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Also install NetBeans 8 or greater:

<https://netbeans.org/downloads>

Currently JavaFX 8 runs on the following operating systems:

- Windows OS (XP, Vista, 7, 8) 32- and 64-bit
- Mac OS X (64-bit)

- Linux (32- and 64-bit)
- Linux ARMv6/7 VFP, HardFP ABI (32-bit)
- Solaris (32- and 64-bit)

After downloading the appropriate software versions for your operating system, you will install the Java 8 JDK and the NetBeans 8 IDE.

Installing the Java 8 Development Kit

The first thing to do is install the correct version of Java. You want the development kit, not the runtime. Follow the steps outlined in this section to download and install Java 8. Download the Java 8 JDK from the following location:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

In the following steps I will be using the Java 8 JDK 32-bit version for the Windows 7 operating system as an example. The steps are similar on other operating systems and JDK versions; however, if your environment is different, please refer to <http://www.oracle.com/technetwork/java/javase/overview/index.html> for additional details. The following are steps to install the Java 8 JDK:

1. Install the Java 8 JDK. Launch the binary executable. The screen in Figure 1-1 will appear once you've launched the JavaFX SDK setup executable. That's a security warning you'll commonly receive on the Windows platform to inform you about installing software. Typically other operating systems will pop up a similar warning dialog box to alert you of risks of installing or running binary files. Click the Run button to proceed. Typically you will need administrator rights on the machine to install software. On Windows you can right-click the executable and run as Administrator.

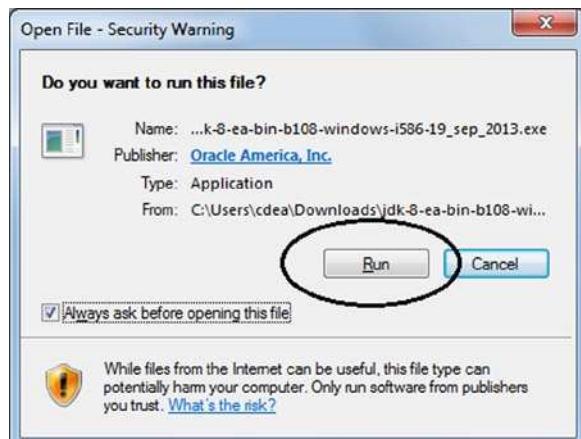


Figure 1-1. The Windows Security Warning dialog window

2. Begin the setup of the Java 8 JDK. The screen in Figure 1-2 will appear, to begin the installation process. Click the Next button to begin installation.



Figure 1-2. Java SE Development Kit 8 setup

3. Select optional features such as additional development tools, source code, and the Java runtime environment. Accept the default selections and click the Next button to install software components. Figure 1-3 shows the screen for optional features to install.



Figure 1-3. Java SE Development Kit 8 optional features

4. The installation process will display a progress bar, as shown in Figure 1-4.



Figure 1-4. Java SE Development Kit 8 installation in progress

5. Change the installation directory of the Java runtime. Accept the defaults and click Next to proceed. Figure 1-5 shows the Destination Folder dialog, where you can change the Java runtime installation directory.



Figure 1-5. The Java 8 runtime destination directory

6. Complete the installation of the Java 8 SE Development Kit. Click the Close button to exit. Shown in Figure 1-6 is the prompt that you have successfully installed the Java 8 SE Development Kit.



Figure 1-6. Java 8 runtime destination directory

Setting Environment Variables

Now you need to set a couple of key environment variables. How you set them and the values they should be set to vary depending on your choice of operating system. The two variables to set are:

JAVA_HOME: Tells your operating system where the Java installation directory lives.

PATH: Specifies where the Java executable directory resides. This environment variable lets the system search paths or directories containing executable files. Java executables reside in the bin directory under the `JAVA_HOME` home directory.

Following are the variables to set in 32-bit Windows:

```
set JAVA_HOME="C:\Program Files (x86)\Java\jdk1.8.0"
set PATH=%JAVA_HOME%\bin;%PATH%
```

Set the same variables in 64-bit Windows, but the values are different:

```
set JAVA_HOME="C:\Program Files\Java\jdk1.8.0"
set PATH=%JAVA_HOME%\bin;%PATH%
```

On Unix-based platforms such as Mac OS X and Linux, your settings and how you make them depend somewhat upon which shell you are using. The following examples show how to set the needed variables for the Bash and Bourne, and the CSH shells:

```
# bash, bourne shell environments
export JAVA_HOME=/usr/java/jdk1.8.0
export PATH=$PATH:$JAVA_HOME/bin
```

```
#csh environments  
setenv JAVA_HOME /usr/java/jdk1.8.0  
setenv PATH ${JAVA_HOME}/bin:${PATH}
```

These statements will set environment variables temporarily for the current terminal window session. To make JAVA_HOME and PATH more permanent you will want to add them to your system upon logon such that they are always made available whenever you boot or login. Depending on your operating system you will need to be able to edit environment variable names and values. In the Windows environment you can use the keyboard shortcut Windows logo key + Pause/Break key, and then click the Advanced system settings to display the Systems Property dialog.

Next, click Environment Variables. This is where you can add, edit, and delete environment variables. You will add or edit the JAVA_HOME environment variable by using the installed home directory as the value. Shown in Figure 1-7 is the Environment Variables dialog on the Windows operating system.

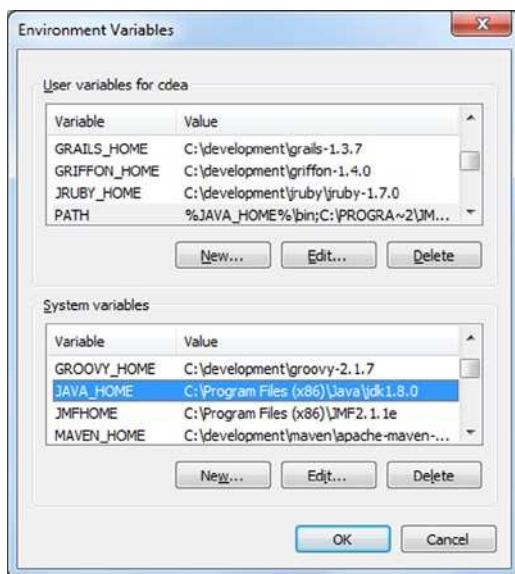


Figure 1-7. Windows Environment Variables

- To set your JAVA_HOME for the Mac OS X platform, you will need to launch a terminal window to edit your home directory's .bash_profile file (for example, named `~/ .bash_profile`) by adding the export commands shown earlier.
- On Linux and other Unix operating systems that use Bash shell environments launch a terminal window and edit either the `~/.bashrc` or `~/.profile` file to contain the export commands.
- On Linux and other Unix operating systems that use C shell (csh) environments, launch a terminal window and edit either the `~/.cshrc` or `~/.login` file to contain the setenv commands.

Once you've set up your path and JAVA_HOME environment variables, you will want to verify by launching a terminal window and executing the following two commands from the command prompt:

```
java -version  
javac -version
```

The output in each case should display a message indicating the Java 8 version of the language and runtime.

Installing the NetBeans IDE

When developing JavaFX applications you will be using the NetBeans IDE. Be sure to download the correct NetBeans version containing JavaFX. To install the NetBeans IDE, follow these steps:

1. Download the NetBeans IDE from the following location:

<https://netbeans.org/downloads/index.html>

2. Install NetBeans 8 or later. Launch the binary executable. The screen in Figure 1-8 will appear once you've launched the NetBeans installation executable. On the Windows platform you will receive a security warning to inform the user about installing software.



Figure 1-8. Windows security warning

3. Click on the Run button to proceed. Figure 1-8 shows a Windows security warning prompt that appears before launching binary files.
4. Begin the NetBeans IDE installation process. Click the Next button. Figure 1-9 prompts the user to begin the installation process.

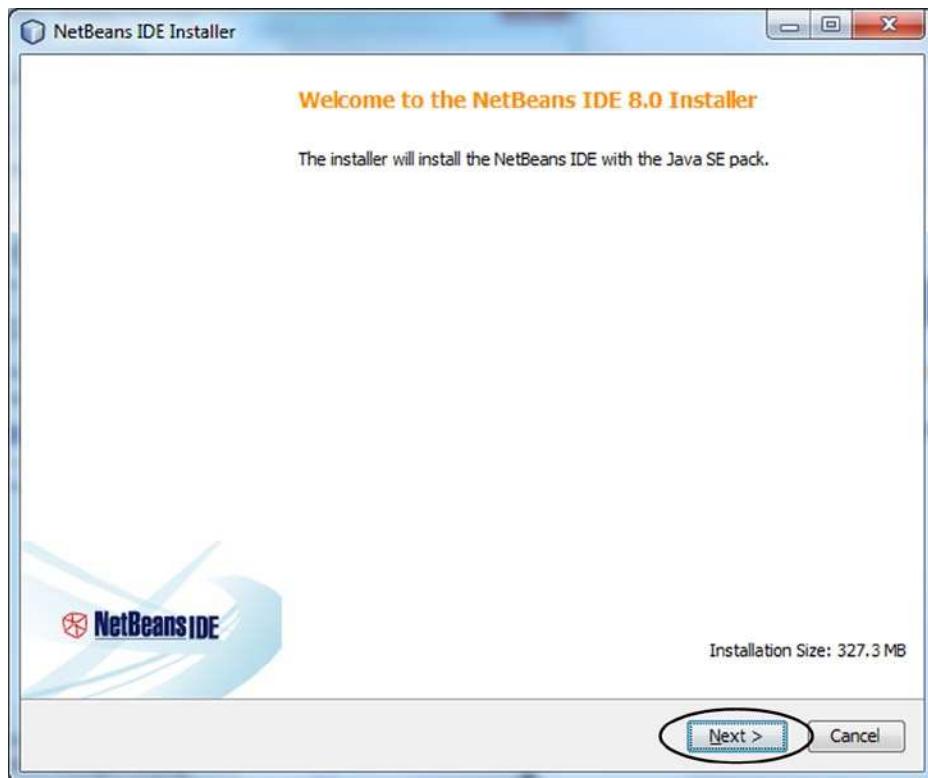


Figure 1-9. NetBeans IDE installer

5. Read and accept the NetBeans license agreement. After reading and agreeing to the terms, click the “I accept the terms in the license agreement” check box and then click Next to proceed. Figure 1-10 shows the NetBeans License Agreement.

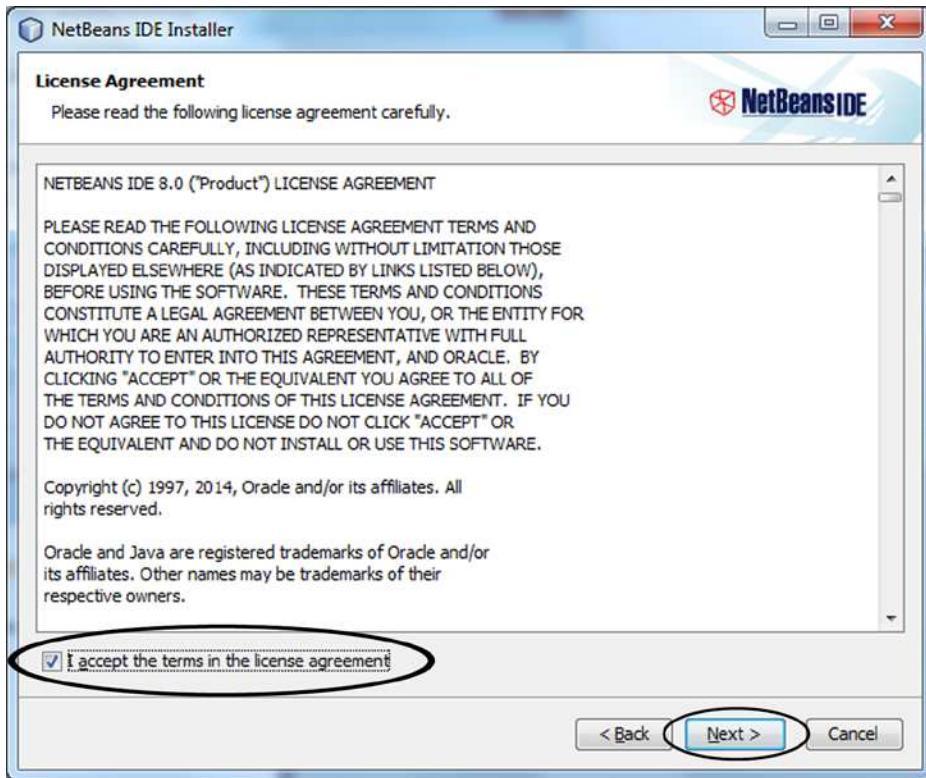


Figure 1-10. NetBeans license agreement

6. Read and accept the JUnit license agreement. After reading and agreeing to the terms, click “I accept the terms in the license agreement. Install JUnit” and then click the Next button to proceed.
7. Shown in Figure 1-11 is the JUnit License Agreement. JUnit is a Java unit testing library.

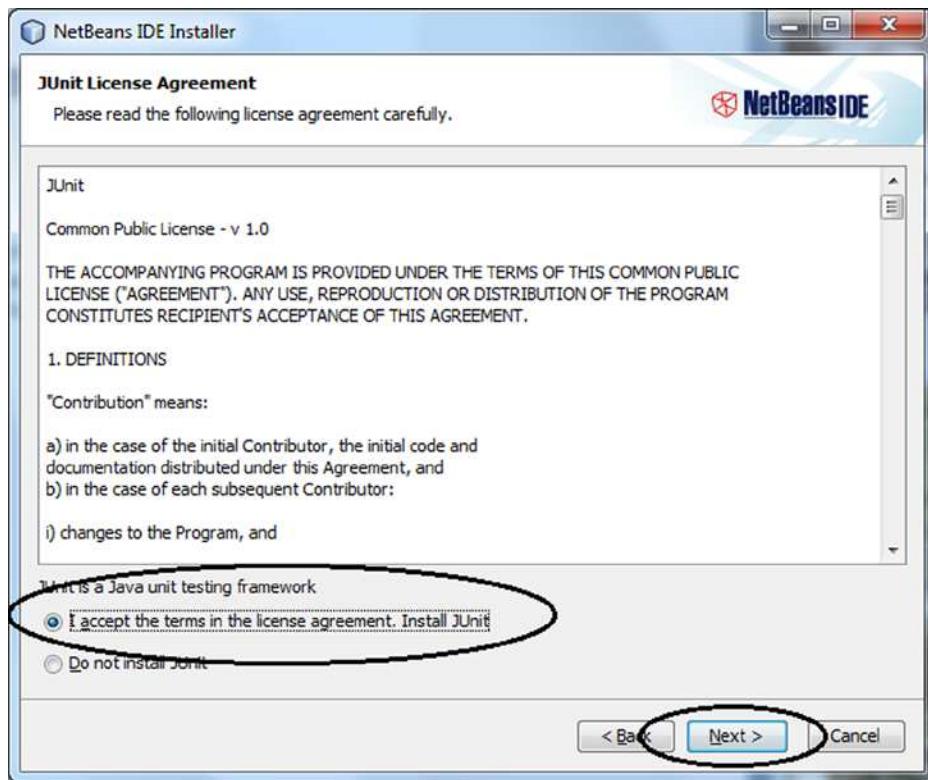


Figure 1-11. JUnit license agreement

8. Choose the destination directory to install NetBeans and select the Java 8 SE Development Kit previously installed. As usual, keep the defaults for the destination to install the NetBeans IDE. However, you will need to select the correct JDK.
9. Figure 1-12 shows a drop-down where you can specify the JDK for the NetBeans IDE. You will select the Java 8 JDK.

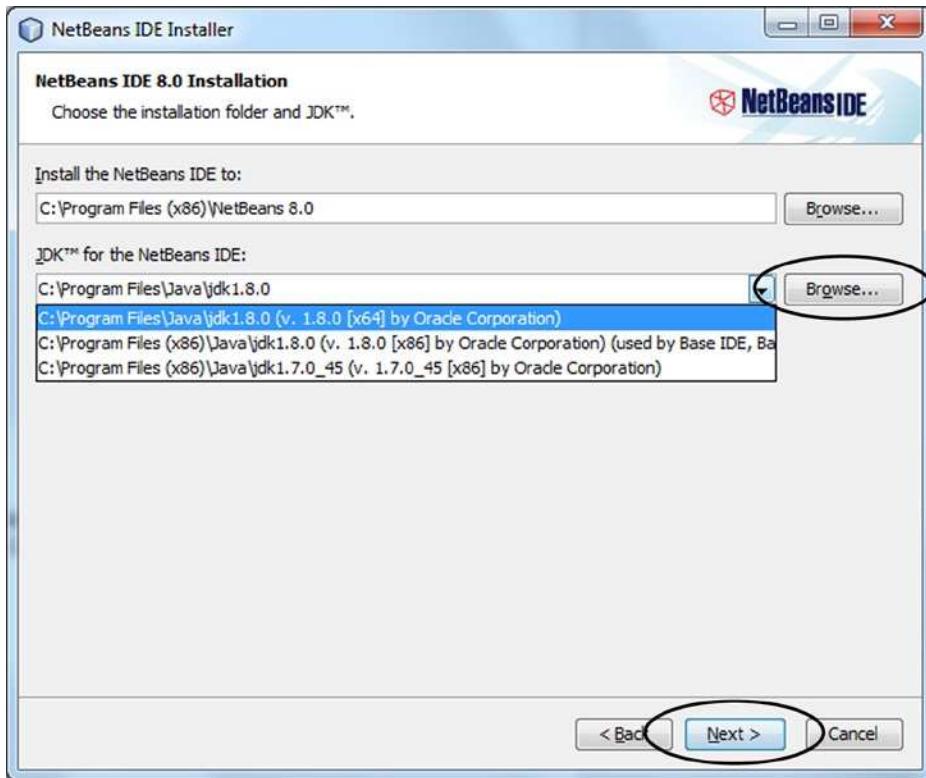


Figure 1-12. NetBeans IDE installation

10. The next stage is choosing the installation directory folders for NetBeans and the JDK; accept the defaults and click Next to proceed with installation. Figure 1-13 depicts the installation destination directories for NetBeans IDE and JDK.

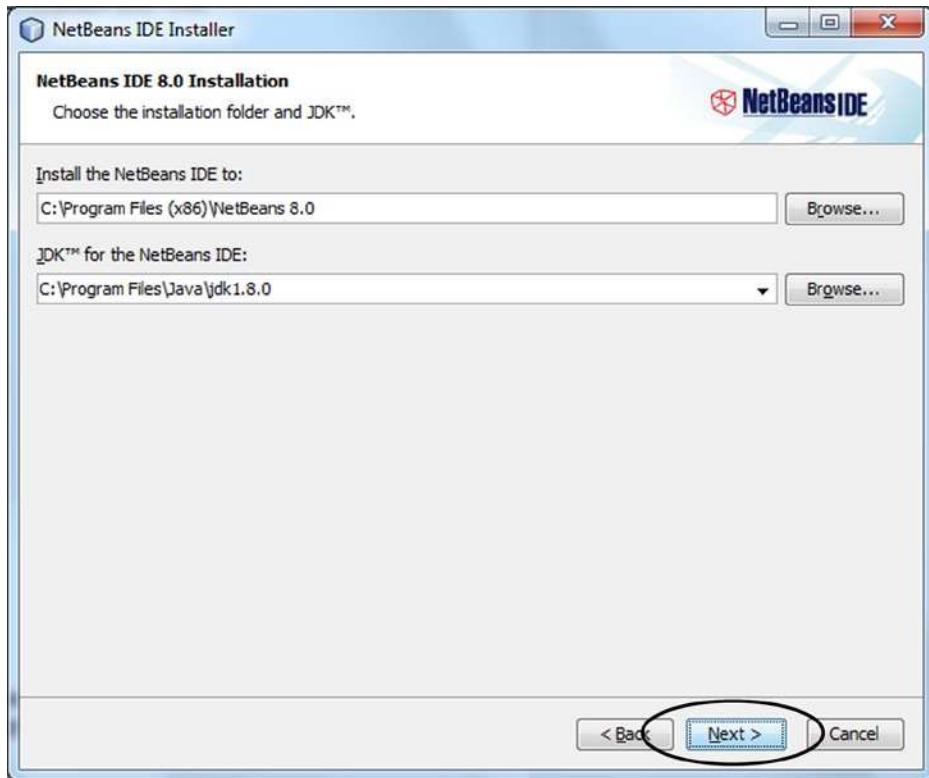


Figure 1-13. NetBeans IDE installation selecting JDK 8

11. On the Summary screen the Check for Updates option allows the installer to retrieve any updates or patches to the current NetBeans version and any other plug-in dependencies. Accept the defaults and click the Install button. As shown in Figure 1-14, the installer defaults to having Check for Updates selected.

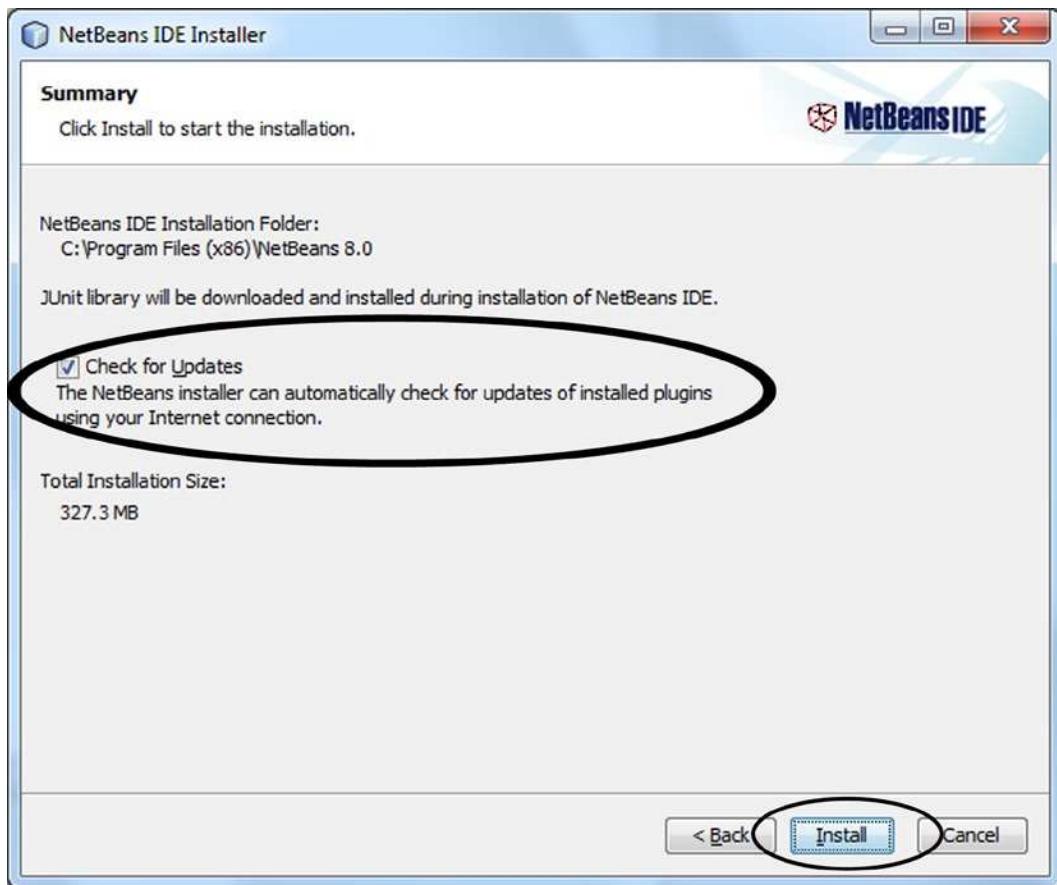


Figure 1-14. Check for Updates

Shown in Figure 1-15 is the installation progress bar.

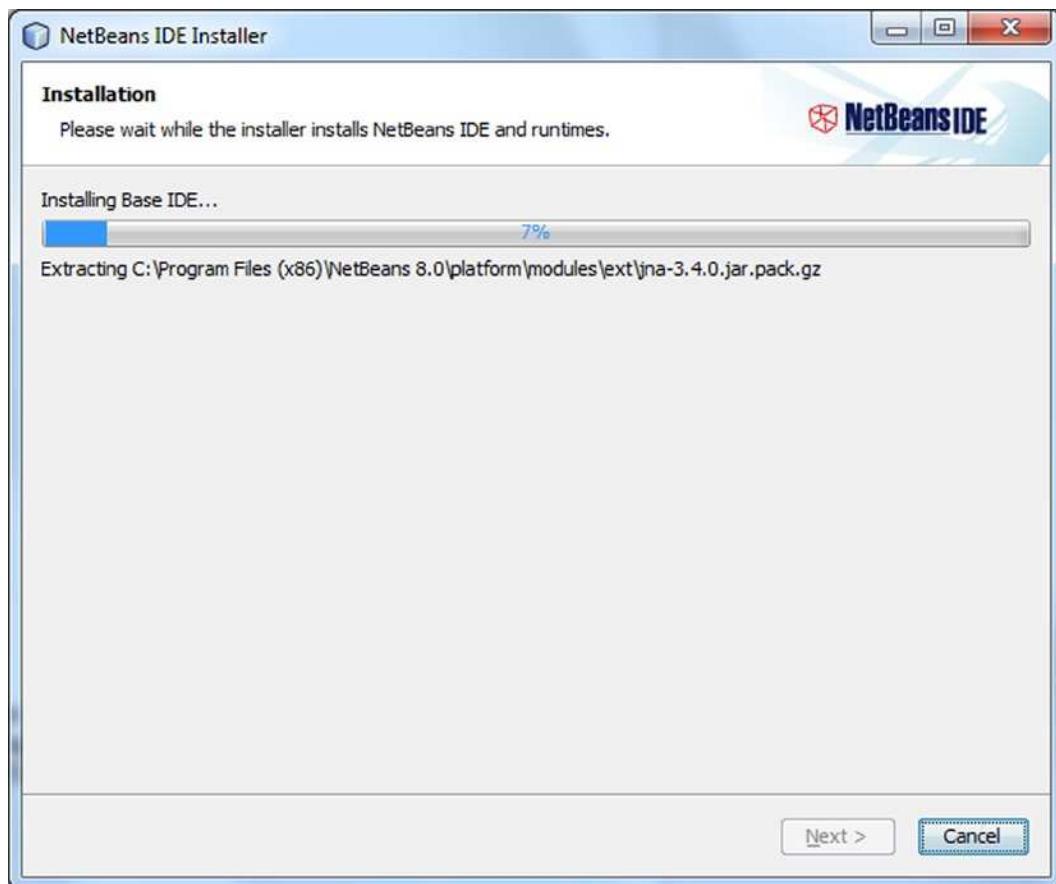


Figure 1-15. Installation progress

12. To complete the setup, in Figure 1-16 you will be prompted for an optional check box to contribute to the NetBeans team anonymous data usage to help diagnose issues and enhance the product. Once you have decided whether to contribute, click the Finish button to complete the installation.

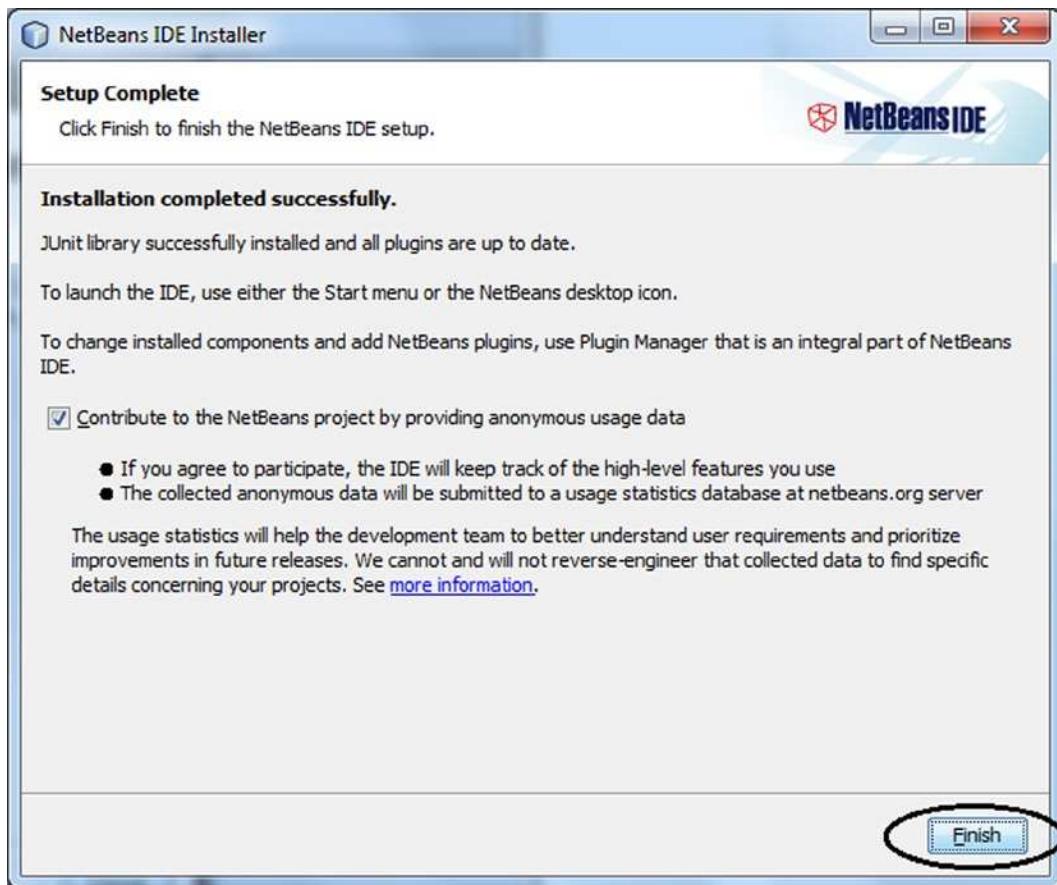


Figure 1-16. Setup complete

13. Launch the NetBeans IDE. You should see the opening page shown in Figure 1-17.

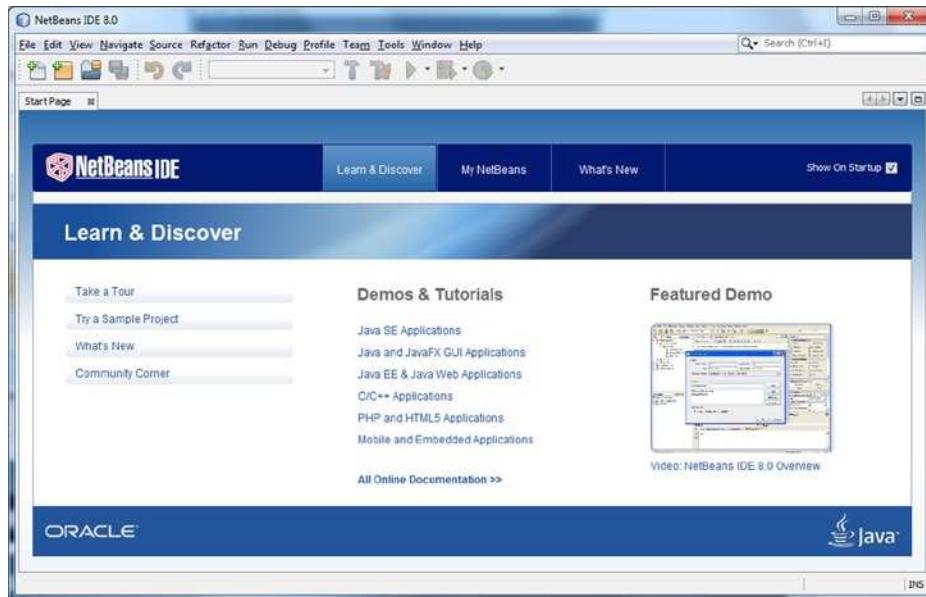


Figure 1-17. NetBeans start page

Now you are ready to go forward and create JavaFX applications.

Creating a JavaFX Hello World Application

You'll be learning two methods in this section to develop, compile, and run a JavaFX-based HelloWorld application. In the first method you will be using the NetBeans IDE we just installed and in the second method you will develop a JavaFX application without using an IDE.

Using the Netbeans IDE

To quickly get started with creating, coding, compiling, and running a simple JavaFX HelloWorld application using the NetBeans IDE, follow the steps outline in this section. You'll want to go deeper in the next section, to get an understanding of what the IDE is doing behind the scenes on your behalf. For now, though, let's just get some code written and running.

1. Launch NetBeans IDE.
2. On the File menu, select New Project.
3. Under Choose Project and Categories, select the JavaFX folder.
4. Under Projects, select Java FX Application, and click Next.

5. Specify HelloWorld for your project name.
6. Change or accept the defaults for the Project Location and Project Folder fields. Figure 1-18 shows the New JavaFX Application dialog wizard to create a simple HelloWorld application.

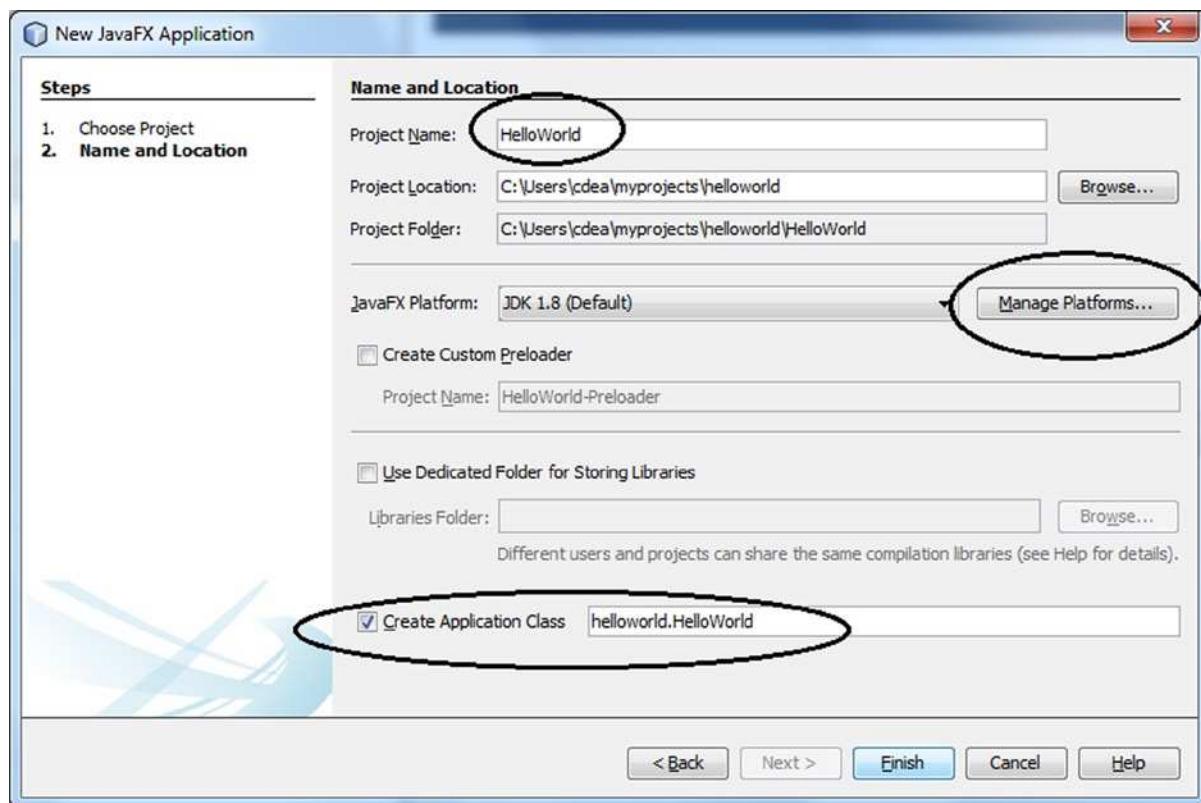


Figure 1-18. New JavaFX Application wizard

7. On this screen click the Manage Platforms button. Make sure the field Platform Name contains JDK 1.8. Figure 1-19 depicts the JDK Home, Classes, Sources, and Javadoc directory locations. Once you have verified them, click the Close button.

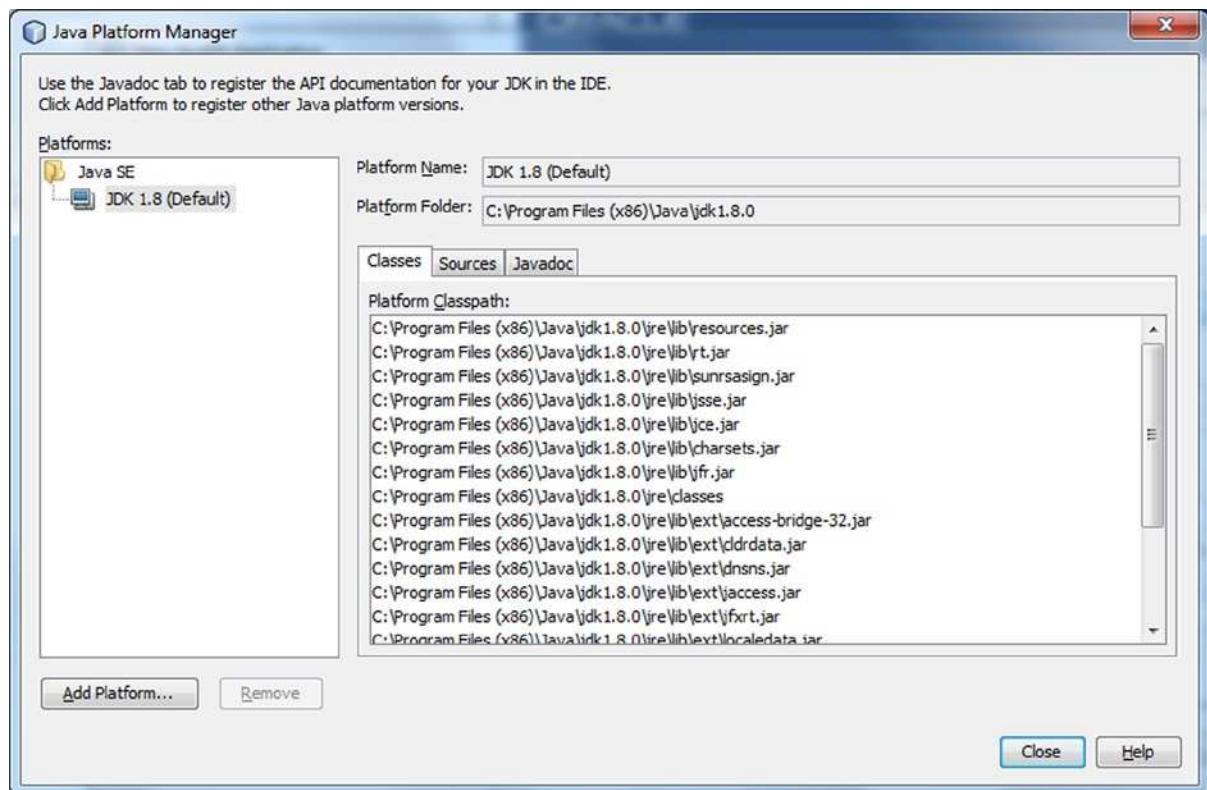


Figure 1-19. The Java Platform Manager window containing JDK 1.8 as a managed platform

8. In Figure 1-18 make sure the Create Application Class option is selected. Then click Finish. In a few seconds NetBeans will generate all the necessary files for the HelloWorld project. After NetBeans finishes, the project will show up on the left Projects tab. Figure 1-20 shows a newly created HelloWorld project.

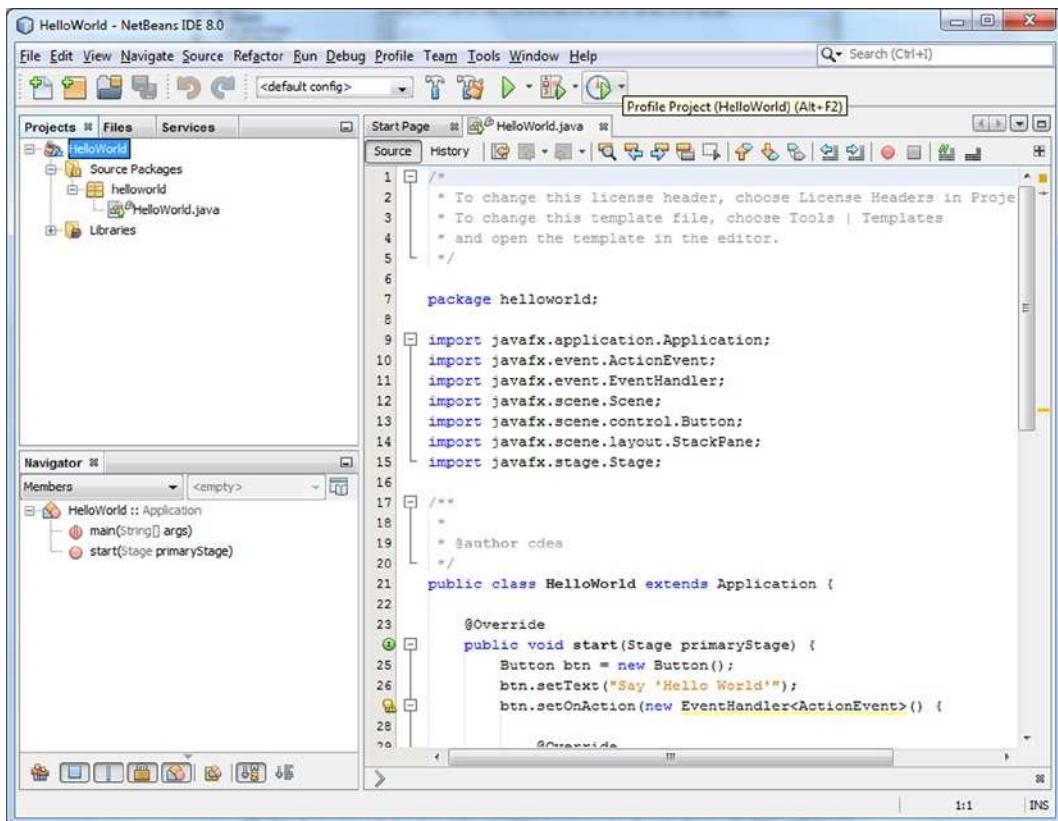


Figure 1-20. A newly created *HelloWorld* project

Note In the NetBeans IDE, which may have other projects, you may encounter a particular project set as the main project. It is advised to set the main project as None by choosing Run > Set Main Project > select **None** (as shown in Figure 1-22). By doing this it allows a project to run just by clicking on the green arrowed button. Assuming the project is selected on the left in the Projects tab, NetBeans will automatically search a project's JavaFX main application class to launch the application. See step 10, following.

9. In the NetBeans IDE on the Projects tab (left), select the newly created project. Open the Project Properties dialog box from the File menu option Project Properties. Here you will verify that the Source/Binary format setting is using JDK 8, as shown in Figure 1-21. Click Sources under Categories.

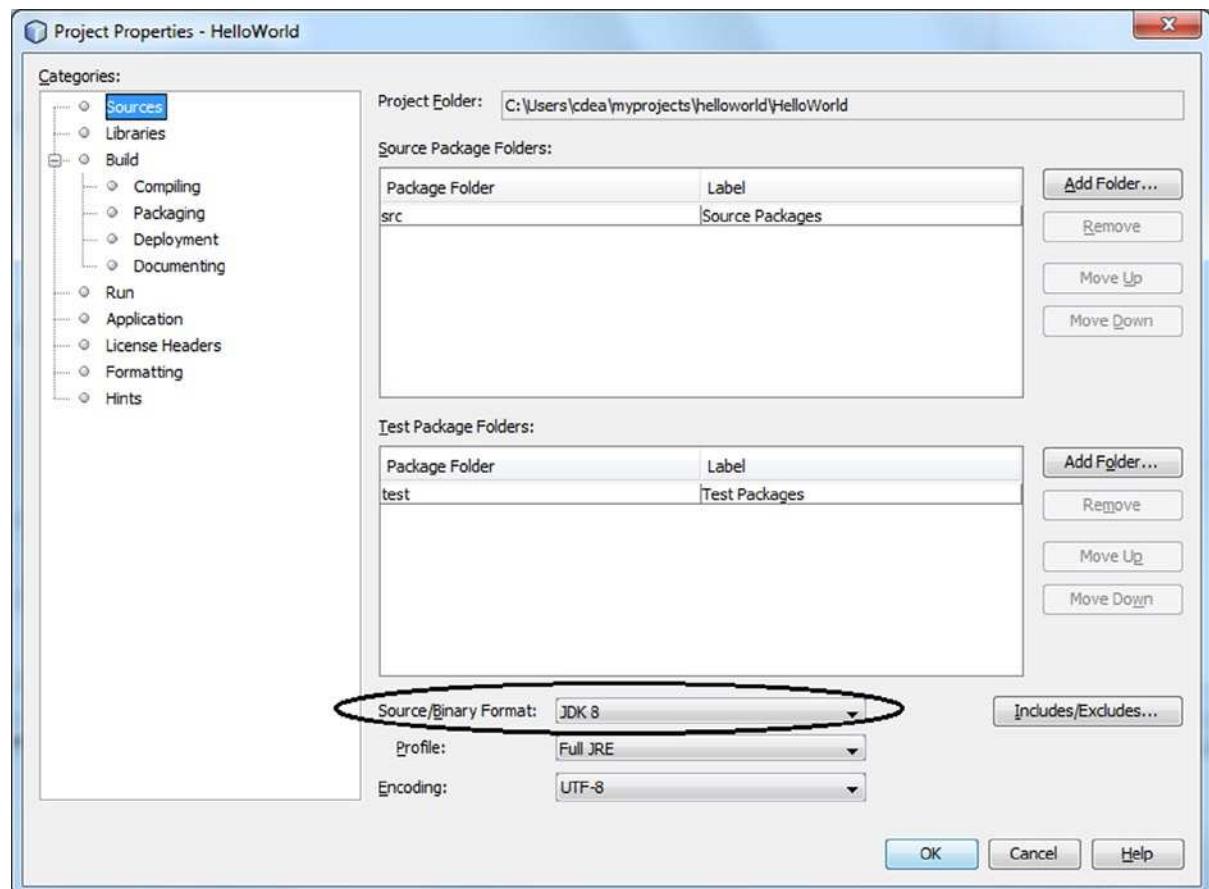


Figure 1-21. Project Properties

10. To run and test your JavaFX Hello World application, ensure the project folder is selected (left Projects tab), then click on the green Run button to execute the HelloWorld project. Notice in Figure 1-22 showing the Set Main Project menu selection set to **None**. This will ensure the HelloWorld or any other project can run without having to be so explicit. This frees the developer to switch projects quickly.

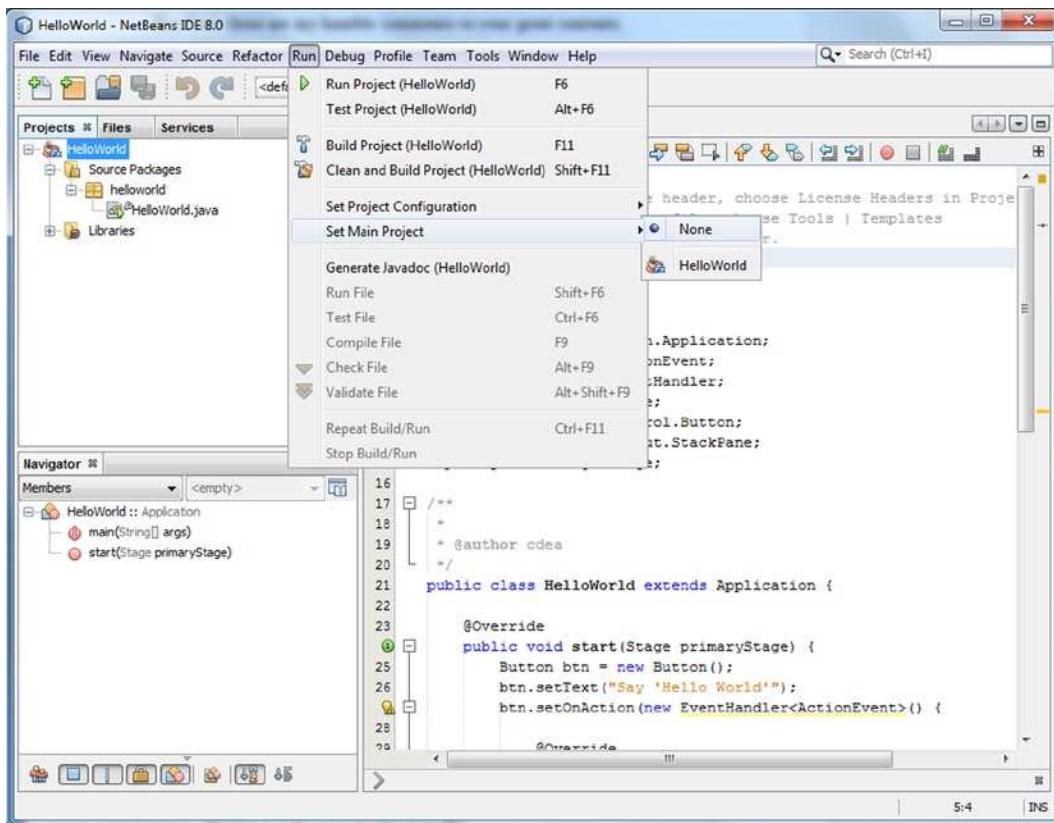


Figure 1-22. Setting the *HelloWorld* project as the Main Project

11. Select the Run menu and Run Main Project or hit the F6 key. You can also click the green arrow icon on the tool bar. After you hit the Run option, the output should look like Figure 1-23.



Figure 1-23. JavaFX Hello World launched from the NetBeans IDE

You shouldn't encounter any difficulty when following steps 1 through 8. However, steps 9 ensures that your project's source/binary format is using JDK 8. Because of the new additions to the Java language in Java 8, such as lambdas, most of the source code in this book will rely on the new syntax and therefore will not be backward-compatible with prior versions of Java 8. Typically, developers have prior versions of the Java development kit such as JDK 7. NetBeans allows you to switch between different JDKs when projects require older versions. An important thing to note is that in early releases of JavaFX 2.0, the software was packaged separately from the Java SDK, which caused some software versioning confusion. Thank goodness it is now just one download (JDK 8)! If you are still using Java 7 SDK, the latest download will also contain JavaFX in one bundle since JavaFX 2.1.

Using the Command-Line Prompt

The second method of developing JavaFX 8 applications is to use a common text editor and the command line prompt (terminal). By learning how to compile and execute applications on the command-line prompt you will learn about the classpath and where the executable files reside. This exposure should sharpen your skills when you are in environments where fancy IDEs and/or editors aren't easily available.

Working from the command line you will basically use popular text editors such as vi, Emacs, or Notepad to code your JavaFX Hello World application. An example is shown in Listing 1-1.

Listing 1-1. A JavaFX Hello World application from HelloWorldMain.java

```
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;

/**
 * A JavaFX Hello World
 * @author carldea
 */
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();
}
}
```

Once the Java file is created you will use the command-line prompt or terminal window 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.

1. Copy and paste the code from Listing 1-1 into a text editor, and save the file as `HelloWorldMain.java`.
2. After saving the file named `HelloWorldMain.java`, on the command-line prompt you will navigate to the directory location of the file. The examples to follow assume you have saved the file to a directory named `myprojects/helloworld`. Change directory as follows on Mac OS X, Linux, and Unix-based operating systems:

cd ~/myprojects/helloworld

3. Change directory on Windows operating systems using the following command:

```
C:\Users\myusername>cd myprojects\helloworld
C:\Users\myusername\myprojects\helloworld>
```

4. Compile the source code file `HelloWorldMain.java` using the Java compiler `javac` with the `-d` switch with the current path. Type the following command:

```
C:\Users\myusername\myprojects\helloworld>javac -d . HelloWorldMain.java
```

Notice the `-d .` before the filename. The dot denotes the current directory. The `-d` option (destination directory) lets the Java compiler know where to put compiled class files based on their package namespace. In this scenario, the `HelloWorldMain` package statement (namespace) is `helloworldmain`, which will create a subdirectory under the current directory assuming we are in the `myprojects/helloworld` directory.

When finished compiling, your directory structure should resemble the following:

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

5. 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:

```
C:\Users\myusername\myprojects\helloworld>java helloworldmain.HelloWorldMain
```

Shown in Figure 1-24 is the output of a simple JavaFX Hello World application launched from the command-line prompt.

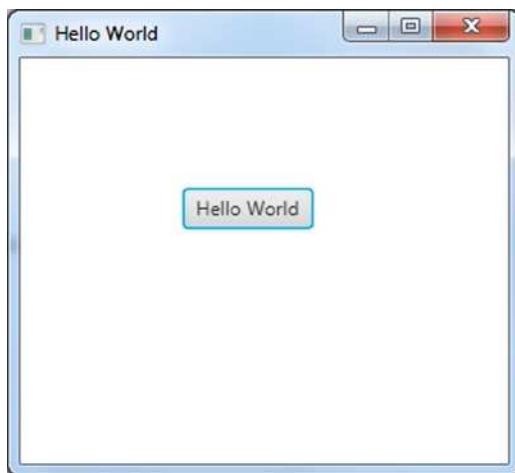


Figure 1-24. JavaFX Hello World launched from the command-line prompt

Walking Through the Code

You'll notice in the source code that JavaFX-based applications extend (inherit) from the `javafx.application.Application` class. The `Application` class provides application life cycle functions such as initializing, launching, starting, and stopping during runtime. This provides a mechanism for Java applications to launch JavaFX GUI components separate from the main thread. The code in Listing 1-2 is a skeleton of the JavaFX Hello World application, having a `main()` method and an overridden `start()` method.

Listing 1-2. A skeleton version of the file HelloWorldMain.java

```
public class HelloWorldMain extends Application {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // Main thread  
        Application.launch(args);  
    }  
  
    @Override  
    public void start(Stage primaryStage) {  
        // JavaFX application thread  
  
        // JavaFX code here...  
    }  
}
```

Here, 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. To access any arguments passed into the `launch()` method you can invoke the `getParameters()` method of the `Application` class. Please see the Javadoc documentation for details on various ways to access named and raw arguments. After the `Application.launch()` method has executed, the application will enter a ready state, and the framework internals will invoke the `start()` method to begin. At this point, the program execution occurs on the JavaFX application thread and not on the main thread. When the `start()` method is invoked, a JavaFX `javafx.stage.Stage` object is available for the developer to use and manipulate. Following is the overridden `Application start()` method:

```
@Override  
public void start(Stage primaryStage) {...}
```

When the program begins in the `start()` method, a separate thread of execution occurs, called the JavaFX application thread. Keep in mind that running on the JavaFX application thread is synonymous with running on Java Swing's event dispatch thread. Later in this book, you will learn how to create background processes to avoid blocking the JavaFX application thread. When you know how to build applications to avoid blocking the GUI, the user will notice that your application is much more responsive (snappy) under heavy usage. Mastering responsiveness in GUI applications is an important concept in enhancing usability and the overall user experience.

JavaFX Scene Graph

You'll notice that some objects are oddly named, such as `Stage` and `Scene`. The designers of the API have modeled things on the idea of a theater or a play in which actors perform in front of an audience. In this analogy, a play consists of 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` on the desktop. Depending on the device, such as a Raspberry Pi (Raspbian), there may be only one stage. You can think of a `Scene` object as a content pane, similar to Java Swing's `JPanel`, capable of holding zero-to-many `Node` objects (children).

Proceeding with our example, in the `start()` method we see that for a JavaFX desktop window (`stage`) you can set the title bar using the `setTitle()` method. Next, you create a root node (`Group`), which is added to the `Scene` object as the top-level surface for the application window. The following code snippet shows how to set the title and create the scene:

```
primaryStage.setTitle("Hello World");
Group root = new Group();
Scene scene = new Scene(root, 300, 250);
```

JavaFX Node

A JavaFX Node is a fundamental base class for all scene graph nodes to be rendered. The following graphics capabilities can be applied to Nodes: scaling, transforms, translations, and effects.

Some of the most 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 such as the `Group` or `Pane` class. We'll learn more about the `Group` class later when we look at the `ObservableList` class, but for now you can think of them as Java List or Collection classes that are capable of holding child `Node` objects. In the following code a button (`Button`) node is created to be positioned on the scene and set with an event handler (`EventHandler<ActionEvent>`) that responds when the user presses the button. The handler code will output the word "Hello World" on the console.

```
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);
```

Once the child nodes have been added to our root `Group` via the `getChildren().add()` method, we set the `primaryStage`'s `scene` and call the `show()` method on the `Stage` object to display a JavaFX application window. By default the window will allow a user to minimize, maximize, and close (exit) the application. Following is the code to set the scene and display (show) the JavaFX application window (`Stage`):

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

Packaging a JavaFX Application

At some point you will want to distribute or deploy your JavaFX application. To handle the numerous application packaging and deployment strategies Oracle's JavaFX team has created a JavaFX Packager tool to assist developers to build, package, and deploy their applications. To learn more about how to use JavaFX Packager tool, please see Oracle's "Deploying JavaFX Applications" at <http://docs.oracle.com/javafx/2/deployment/jfxpub-deployment.htm>.

To give you a taste of the JavaFX Packager tool's capabilities, I will show you how to package the `HelloWorldMain` classes into a single executable jar file that a user can double-click or launch from a command prompt (terminal).

```
cd myprojects/helloworld  
javac -d . HelloWorldMain.java  
javafxpackager -createjar -appclass helloworldmain.HelloWorldMain -srcdir . -outdir out  
-outfile helloworld.jar -v
```

Table 1-1 describes the common options and switches that are used in building a JavaFX jar executable application.

Table 1-1. Common `javafxpackager` Options to Build an Executable jar

Option/Switch	Example	Description
<code>-createjar</code>	<code>--</code>	Creates a JavaFX JAR executable application.
<code>-appclass</code>	<code>helloworldmain.HelloWorldMain</code>	Specifies the fully qualified name of the class containing the <code>main()</code> method.
<code>-srcdir</code>	<code>.</code>	The top-level location of the parent directory holding the compiled classes (current directory).
<code>-outdir</code>	<code>out</code>	The destination where the packaged jar file will be created.
<code>-outfile</code>	<code>helloworld.jar</code>	Specifies the name of the executable jar file.
<code>-v</code>	<code>--</code>	Allows verbose displays logging information when executing <code>javafxpackager</code> .

To run the jar executable on the command line, you simply type the following and press Enter:

```
javaw -jar out/helloworld.jar
```

There are many ways to package and deploy JavaFX applications. To learn more, please see Oracle's "Deploying JavaFX Applications" at the following URL:

<http://docs.oracle.com/javafx/2/deployment/jfxpub-deployment.htm>

Also see the following, related articles:

- "Packaging JavaFX Applications as Native Installers" at <http://fxexperience.com/2012/03/packaging-javafx-applications-as-native>
- "JavaFX And Maven" at <http://www.zenjava.com/2012/02/17/javafx-and-maven>
- "JavaFX Gradle Plugin 0.3.0 Released" at <https://bitbucket.org/shemnon/javafx-gradle>

Summary

So far you have managed to download and install both Java 8 JDK and NetBeans IDE. After successfully installing prerequisite software, you created a JavaFX Hello World GUI application through the NetBeans IDE and used a text editor to compile and run the binary class via a command-line prompt (terminal window). After learning two methods of compiling and running a JavaFX application, you did a quick code walkthrough of the source file `HelloWorldMain.java`. You also learned to package a JavaFX application as a standalone jar executable. Next, in Chapter 2 you will learn the fundamentals of JavaFX 8 such as drawing and coloring shapes as well as drawing text and changing text fonts.

JavaFX Fundamentals

There is nothing more frustrating than receiving complicated advice as a solution to a problem. Because of this, I have always made it a point to focus on the fundamentals. In order to render graphics on the JavaFX scene, you will need basic shapes and colors. Expanding upon your knowledge of the JavaFX scene graph, which you visited toward the end of Chapter 1, you will learn in this chapter how to draw and manipulate 2D shapes on the JavaFX scene graph.

Even more fundamental than a shape object is JavaFX's `javafx.scene.Node` class. The `Node` class is a fundamental base class for all JavaFX scene graph nodes. The `Node` class provides the ability to transform, translate, and apply effects to any node. Many examples in this chapter involve the `javafx.scene.shape.Shape` class, which is a descendant of the `Node` class. Therefore, the `Shape` class will inherit all of the `Node` class' capabilities. In this chapter you will explore many derived classes that inherit from `Shape`. You will begin this chapter by examining JavaFX's most basic shape, the `Line` node.

JavaFX Lines

Rendering lines in JavaFX is conceptually similar to Euclidean geometry in that two (x, y) coordinate points connect to draw a segment into space. When drawn on the JavaFX scene graph, lines are rendered using the screen coordinate space (system). In geometry, a line is defined as a segment connected by two points in space, although it lacks a width (thickness) and color value. Does this mean that the line is invisible? In the computer world, physical devices plot pixels and shapes that occupy real surfaces. Monitors and printers are examples of such surfaces. Because of this method of production, modern graphics programming has adopted the standard screen coordinate system. In this system, lines that are drawn are visible and have attributes of width and color.

In a nutshell, the standard screen coordinate system places (0, 0) at the upper-left corner; this differs from the Cartesian coordinate system, where (0, 0) is placed at the center of the graphing area or the point at which the x and y axes converge. In both coordinate systems, the x coordinate has the same effect when moving a point along the x axis. However, when using the screen coordinate system to move along the y axis, the effect is opposite that of the Cartesian system. In other words, the y coordinate's value increases when moving a point in a downward direction (top to bottom). Figure 2-1 illustrates shapes drawn using the screen coordinate system. Also note that using negative values will plot objects off the screen.

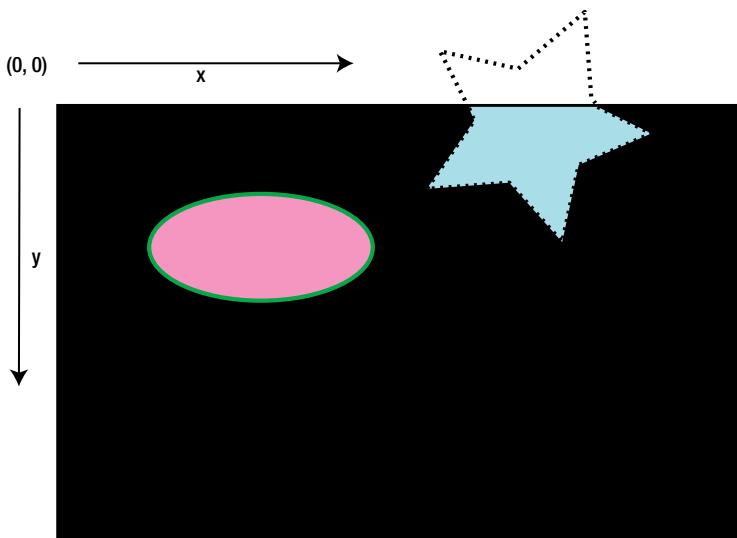


Figure 2-1. The screen coordinate system

As you learn more about JavaFX, you will discover many scene graph objects, such as lines, circles, and rectangles. These objects are derived classes of the Shape class. All shape objects can perform geometric operations between two shaped areas, such as subtract, intersect and union. By mastering the Shape API, you will begin to see endless possibilities. Now let's discuss how to create lines.

To draw lines in JavaFX, you will use the `javafx.scene.shape.Line` class. When creating a Line instance you need to specify a start (x, y) coordinate and an end coordinate to draw a line. There are two ways to specify the start and end points when creating Line nodes. The first method is to use a constructor with the parameters `startX`, `startY`, `endX`, and `endY`. The data types of all the parameters are double values, which provide floating-point decimal precision. The following line has a start point (100, 10) and end point (10, 110) created using a constructor.

```
Line line = new Line(100, 10, 10, 110);
```

The second method to create a line node is to instantiate a Line class by using the empty default constructor and subsequently setting each attribute using associated setter methods. The following code snippet shows how to specify a line's start and end points using setter methods.

```
Line line = new Line();
line.setStartX(100);
line.setStartY(10);
line.setEndX(10);
line.setEndY(110);
```

Line nodes drawn on the scene graph default to a stroke width of 1.0 (double) and a stroke color of black (`Color.BLACK`). According to the Javadoc, all shapes have a stroke color of null (no color) except Line, Polyline, and Path nodes.

Now that you know how to draw lines on the scene graph, you are probably wondering how to be more creative with lines. Creating different kinds of lines is simple; you basically set properties inherited from the parent class (`javafx.scene.shape.Shape`). Table 2-1 shows the properties you can set on a line (Shape). To retrieve or modify each property, you will use its appropriate getter and setter methods. The table lists each property name and its data type, with a description. Please refer to the Javadoc documentation for more details.

Table 2-1. The `javafx.scene.shape.Shape` Class Properties

Property	Data Type	Description
<code>fill</code>	<code>javafx.scene.paint.Paint</code>	A color to fill inside a shape.
<code>smooth</code>	<code>Boolean</code>	True turns on anti-aliasing, otherwise false.
<code>strokeDashOffset</code>	<code>Double</code>	The offset (distance) into the dashed pattern.
<code>strokeLineCap</code>	<code>javafx.scene.shape.StrokeLineCap</code>	The cap style on the end of a line or path. There are three styles: <code>StrokeLineCap.BUTT</code> , <code>StrokeLineCap.ROUND</code> , and <code>StrokeLineCap.SQUARE</code> .
<code>strokeLineJoin</code>	<code>javafx.scene.shape.StrokeLineJoin</code>	Decoration when lines meet. There are three types: <code>StrokeLineJoin.MITER</code> , <code>StrokeLineJoin.BEVEL</code> , and <code>StrokeLineJoin.ROUND</code> .
<code>strokeMiterLimit</code>	<code>Double</code>	The limit of a miter joint. Used along with the miter join decoration (<code>StrokeLineJoin.MITER</code>).
<code>stroke</code>	<code>javafx.scene.paint.Paint</code>	A color for the shape's line stroke.
<code>strokeType</code>	<code>javafx.scene.shape.StrokeType</code>	Where to draw the stroke around the boundary of a Shape node. There are three types: <code>StrokeType.CENTERED</code> , <code>StrokeType.INSIDE</code> , and <code>StrokeType.OUTSIDE</code> .
<code>strokeWidth</code>	<code>Double</code>	A stroke line's width.

Drawing Lines

To get a better idea of how you would use a shape's properties based on Table 2-1, let's look at an example. Figure 2-2 is the output of Listing 2-1, the `DrawingLines.java` source code, demonstrating the drawing of JavaFX lines. The JavaFX application in the listing draws three lines with various properties modified. Some common properties used in this example are stroke dash offset, stroke line cap, stroke width, and stroke color.

In Figure 2-2 you can see that the first line (top) is a thick red line with a dashed stroke pattern. The second line is a thick white line having rounded end caps (line cap). Last is an ordinary blue line having the same thickness as the others. You will also notice in Figure 2-2 underneath the blue line are a label and slider control; these allow the user to change the red (top) line's stroke dash offset property dynamically. The label control displays the dash offset value as the slider control moves.

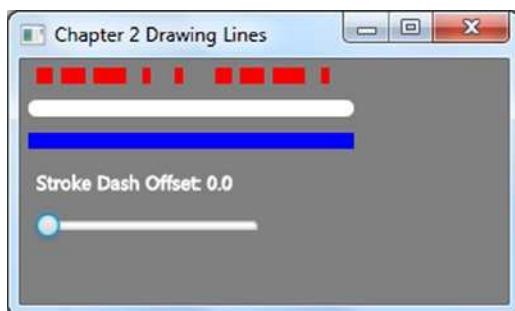


Figure 2-2. Drawing lines

Listing 2-1 shows the source code for DrawingLines.java.

Listing 2-1. DrawingLines.java

```
package jfx8ibe;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.scene.shape.StrokeLineCap;
import javafx.scene.text.Text;
import javafx.stage.Stage;

/**
 * Drawing Lines
 * Listing 2-1 DrawingLines.java
 * @author carldea
 */
public class DrawingLines extends Application {

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chapter 2 Drawing Lines");

        Group root = new Group();
        Scene scene = new Scene(root, 300, 150, Color.GRAY);
```

```
// Red line
Line redLine = new Line(10, 10, 200, 10);

// setting common properties
redLine.setStroke(Color.RED);
redLine.setStrokeWidth(10);
redLine.setStrokeLineCap(StrokeLineCap.BUTT);

// creating a dashed pattern
redLine.getStrokeDashArray().addAll(10d, 5d, 15d, 5d, 20d);
redLine.setStrokeDashOffset(0);

root.getChildren().add(redLine);

// White line
Line whiteLine = new Line(10, 30, 200, 30);
whiteLine.setStroke(Color.WHITE);
whiteLine.setStrokeWidth(10);
whiteLine.setStrokeLineCap(StrokeLineCap.ROUND);

root.getChildren().add(whiteLine);

// Blue line
Line blueLine = new Line(10, 50, 200, 50);
blueLine.setStroke(Color.BLUE);
blueLine.setStrokeWidth(10);

root.getChildren().add(blueLine);

// slider min, max, and current value
Slider slider = new Slider(0, 100, 0);
slider.setLayoutX(10);
slider.setLayoutY(95);

// bind the stroke dash offset property
redLine.strokeDashOffsetProperty()
    .bind(slider.valueProperty());
root.getChildren()
    .add(slider);

Text offsetText = new Text("Stroke Dash Offset: 0.0");
offsetText.setX(10);
offsetText.setY(80);
offsetText.setStroke(Color.WHITE);

// display stroke dash offset value
slider.valueProperty()
    .addListener((ov, curVal, newVal) ->
        offsetText.setText("Stroke Dash Offset: " + slider.getValue()));
root.getChildren().add(offsetText);
```

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

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

DrawingLines.java begins by setting the title of the Stage window using the `setTitle()` method. Then it creates a root node (`javafx.scene.Group`) for the Scene object. All root nodes (those extending `javafx.scene.Parent`) have a method `getChildren().add()` to allow any JavaFX node to be added to the scene graph. By default the Scene object will be filled with a white background; however, because one of our lines is white, the code will set the scene to have a color of gray (`Color.GRAY`). This will allow some contrast in order to see the white line. Next is the code that creates the three lines (red, white and blue).

In the first line, the code sets the common properties of a Line node. The common properties are stroke color, stroke width, and stroke line cap. As noted earlier, lines do not have a shape internally, so a line's fill color property is set to null (no color) and stroke color defaults to black. To set the stroke color you can use built-in JavaFX colors by using the `javafx.scene.paint.Color` class. For instance, for the color red you will use `Color.RED`. There are many ways to specify color, such as using RGB, HSB, or Web hex values. All three methods also have the ability to specify alpha value (transparency). Later, you will see additional methods for coloring shapes. Please refer to the Javadoc to learn more about color (`javafx.scene.paint.Color`). After setting the stroke's outline color you can set the line's stroke width (thickness) by using the `setStrokeWidth()` method. Shapes also have a stroke line cap property. This property specifies the style of the ends of the line. For instance, specifying the stroke line cap as `butt` (`StrokeLineCap.BUTT`) will make a flat square end, while a round (`StrokeLineCap.ROUND`) style will appear with a rounded end. The following code snippet sets a line node's common shape properties:

```
// setting common properties
redLine.setStroke(Color.RED);
redLine.setStrokeWidth(10);
redLine.setStrokeLineCap(StrokeLineCap.BUTT);
```

After setting the common properties on the red Line node, the example code creates a dashed pattern. To form a dashed pattern you would simply add double values to the `getStrokeDashArray().addAll()` method. Each value represents the number of pixels for a dash segment. To set the dash segment array, the first value (10d) is a visible dash segment 10 pixels wide. Next is a five-pixel empty segment (not visible). Following that is a visible dash segment 15 pixels wide, and so on. Because the array has an odd number of values, you can see that as the pattern repeats itself, the first value (10d) becomes a 10-pixel empty segment (not visible). Following is the code to create a dashed pattern for a Line node:

```
// creating a dashed pattern
redLine.getStrokeDashArray().addAll(10d, 5d, 15d, 5d, 20d);
redLine.setStrokeDashOffset(0);
```

By default the dash offset property value is zero. To see what happens when the offset property changes, the user can drag the slider thumb to the right to increase the offset. The stroke dash offset is the distance into the current pattern to begin the line drawing.

Because the other two lines are pretty much the same in the way they modify common properties, I will not explain them any further. I trust you now understand the fundamentals of creating lines. One last thing to mention about this example is the Slider control and the way it is wired up using binding. We will discuss binding further in Chapter 3.

Notice in Listing 2-1 also that after the three lines just discussed, the slider control has handler code that updates a Text node dynamically to display the dash offset value. Also notice the invocation to the addListener() method, containing concise code added as a change listener. This may look odd to you; however, it reflects a new Java 8 feature called *lambda expressions*, which you will learn more about in Chapter 3. The following code snippet creates a change listener using lambdas instead of an anonymous inner class (ChangeListener).

```
// display stroke dash offset value
slider.valueProperty()
    .addListener( (ov, oldVal, newVal) ->
        offsetText.setText("Stroke Dash Offset: " + slider.getValue()));
```

Learning about how to draw lines will help you apply the same knowledge to any Shape node in JavaFX. These important concepts will allow you to create any kind of shape styled the way you see fit. Speaking of shapes, that's what we'll discuss next.

Drawing Shapes

If you are familiar with drawing programs that allows a user to draw shapes on the canvas, you'll notice a lot of similarities when using the JavaFX's Shape API. The JavaFX Shape API allows you to create many common shapes such as lines, rectangles, and circles. Some of the more complex shapes are Arc, CubicCurve, Ellipse, and QuadCurve. JavaFX also lets you create custom shapes for situations where the predefined stock shapes don't fit the bill. In this section you will be exploring basic shapes, complex shapes, and custom shapes. I will not demonstrate all of the shapes available to JavaFX, because of space limitations. To see all the available shapes, refer to the Javadoc documentation for details.

As stated earlier, JavaFX has common shapes such as lines, rectangles and circles. In the section "Drawing Lines," you learned the basic method of changing any shape's stroke color, stroke width, and many other attributes. Knowing these skills will be useful throughout this chapter. Let's begin by getting familiar with the rectangle shape.

Drawing rectangles on the scene graph is quite easy. Just as you learned in a geometry classroom, you specify a width, height, and an (x, y) location (upper-left corner) to position the rectangle on the scene graph. To draw a rectangle in JavaFX you use the javafx.scene.shape.Rectangle class. In addition to common attributes, the Rectangle class also implements an *arc width* and *arc height*. This feature will draw rounded corners on a rectangle. Figure 2-3 shows a rounded rectangle, which has both an *arc width* and an *arc height*.

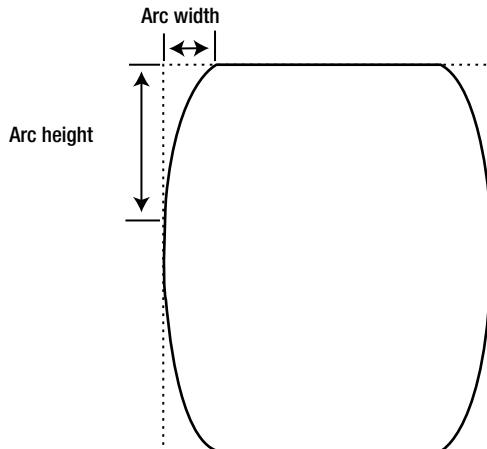


Figure 2-3. A rounded rectangle's arc width and arc height

Following is a code snippet that draws a rectangle positioned at (50, 50) with a width of 100, a height of 130, an arc width of 10, and an arc height of 40.

```
Rectangle roundRect = new Rectangle();
roundRect.setX(50);
roundRect.setY(50);
roundRect.setWidth(100);
roundRect.setHeight(130);
roundRect.setArcWidth(10);
roundRect.setArcHeight(40);
```

Having the basic knowledge of simple shapes such as lines and rectangles you will inevitably come across other simple shapes having equivalent properties that we can manipulate in the same way. Armed with this knowledge of simple shapes let's look at creating more complex shapes.

Drawing Complex Shapes

Learning about simple shapes is great, but to create more complex shapes you need to see what other built-in shapes that the JavaFX API has to offer. Exploring the Java documentation (`javafx.scene.shape.*`), you will discover many derived shape classes to choose from. The following are all the currently supported shapes:

- Arc
- Circle
- CubicCurve
- Ellipse
- Line
- Path

- Polygon
- Polyline
- QuadCurve
- Rectangle
- SVGPath
- Text (which is considered to be a type of shape)

A Complex Shape Example

To demonstrate the drawing of complex shapes, the next code example draws four fun cartoon-like shapes. Normally cartoon drawings have a thick stroke width, similar to a penciled-in outline. The first shape in the example is a sine wave; the second an ice cream cone, the third a smile, and the last a donut. Before studying the code you may want to see the shapes drawn onto the scene, so you can visualize each shape as you look at the code listing. Figure 2-4 is the output of Listing 2-2, depicting four complex shapes.

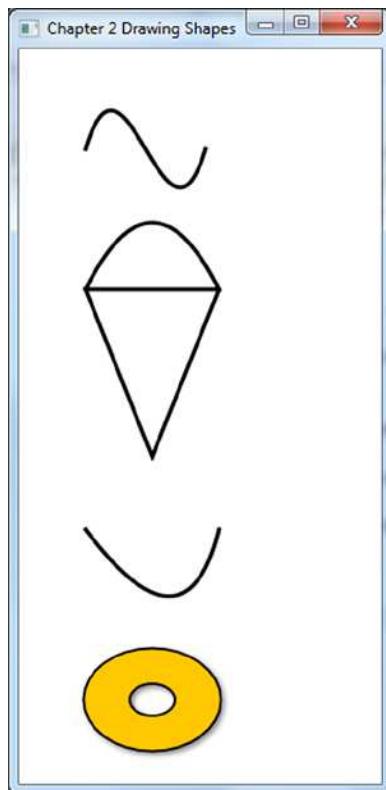


Figure 2-4. Drawing complex shapes

The DrawingShape.java code shown in Listing 2-2 demonstrates the drawing of the shapes you see in Figure 2-4. The first complex shape involves a cubic curve (CubicCurve) that is drawn in the shape of a sine wave. The next shape is an ice cream cone; it uses the Path class, which 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; to create this donut shape you will create two (Ellipse) shapes (one smaller and one larger) and subtract one. For brevity, Listing 2-2 shows just the start() method, containing the main JavaFX elements. To get the full code listing, download the example code from the book's web site.

Listing 2-2. DrawingShape.java

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Chapter 2 Drawing Shapes");
    Group root = new Group();
    Scene scene = new Scene(root, 306, 550, Color.WHITE);

    // Sine wave
    CubicCurve cubicCurve = new CubicCurve(
        50, // start x point
        75, // start y point
        80, // control x1 point
        -25, // control y1 point
        110, // control x2 point
        175, // control y2 point
        140, // end x point
        75); // end y point
    cubicCurve.setStrokeType(StrokeType.CENTERED);
    cubicCurve.setStroke(Color.BLACK);
    cubicCurve.setStrokeWidth(3);
    cubicCurve.setFill(Color.WHITE);

    root.getChildren().add(cubicCurve);

    // Ice cream cone
    Path path = new Path();
    path.setStrokeWidth(3);

    // create top part beginning on the left
    MoveTo moveTo = new MoveTo();
    moveTo.setX(50);
    moveTo.setY(150);

    // curve ice cream (dome)
    QuadCurveTo quadCurveTo = new QuadCurveTo();
    quadCurveTo.setX(150);
    quadCurveTo.setY(150);
    quadCurveTo.setControlX(100);
    quadCurveTo.setControlY(50);
```

```
// cone rim
LineTo lineTo1 = new LineTo();
lineTo1.setX(50);
lineTo1.setY(150);

// left side of cone
LineTo lineTo2 = new LineTo();
lineTo2.setX(100);
lineTo2.setY(275);

// right side of cone
LineTo lineTo3 = new LineTo();
lineTo3.setX(150);
lineTo3.setY(150);

path.getElements().addAll(moveTo, quadCurveTo, lineTo1, lineTo2 , lineTo3);

path.setTranslateY(30);

root.getChildren().add(path);

// A smile
QuadCurve quad = new QuadCurve(
    50, // start x point
    50, // start y point
    125, // control x point
    150, // control y point
    150, // end x point
    50); // end y point
quad.setTranslateY(path.getBoundsInParent().getMaxY());
quad.setStrokeWidth(3);
quad.setStroke(Color.BLACK);
quad.setFill(Color.WHITE);

root.getChildren().add(quad);

// outer donut
Ellipse bigCircle = new Ellipse(
    100, // center x
    100, // center y
    50, // radius x
    75/2); // radius y
bigCircle.setStrokeWidth(3);
bigCircle.setStroke(Color.BLACK);
bigCircle.setFill(Color.WHITE);
```

```
// donut hole
Ellipse smallCircle = new Ellipse(
    100, // center x
    100, // center y
    35/2, // radius x
    25/2); // radius y

// make a donut
Shape donut = Path.subtract(bigCircle, smallCircle);
donut.setStrokeWidth(1.8);
donut.setStroke(Color.BLACK);

// orange glaze
donut.setFill(Color.rgb(255, 200, 0));

// add drop shadow
DropShadow dropShadow = new DropShadow(
    5, // radius
    2.0f, // offset X
    2.0f, // offset Y
    Color.rgb(50, 50, 50, .588));

donut.setEffect(dropShadow);

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

root.getChildren().add(donut);

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

Four shapes are drawn in Figure 2-4. Each shape will be detailed further in the following sections, which describe the code and the reasoning behind the creation of each of the four shapes.

The Cubic Curve

In Listing 2-2 the first shape, drawn as a sine wave, is really 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 to set are `startX`, `startY`, `controlX1` (control point1 X), `controlY1` (control point1 Y), `controlX2` (control point2 X), and `controlY2` (control point2 Y), `endX`, `endY`. Figure 2-5 shows a cubic curve with control points influencing the curve.

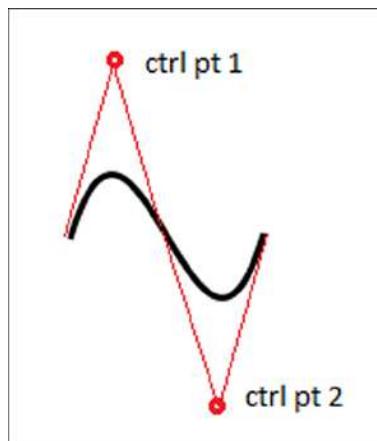


Figure 2-5. Cubic curve

The startX, startY, endX, and endY parameters are the starting and ending points of a curved line, and controlX1, controlY1, controlX2, and controlY2 are control points. The control point (controlX1, controlY1) is a point in screen space that will influence the line segment between the start point (startX, startY) and the midpoint of the line. The point (controlX2, controlY2) is another control point that will influence the line segment between the midpoint of the line and its end point (endX, endY). A control point is a point that pulls the curve toward the direction of itself. A definition of a control point is a line perpendicular to a tangent line on the curve. In our example, we simply have a control point 1 above the line to pull the curve upward to form a hill and control point 2 below the line to pull the curve downward to form a valley.

Note All older JavaFX 2.x Builder classes are deprecated in JavaFX 8. Be advised that the previous edition of this book used Builder classes. Shape classes using deprecated builder classes should be converted in favor of constructors and setter methods when specifying properties.

At this point I want to mention that beginning with JavaFX 8, Oracle's JavaFX team has decided to deprecate the many Builder classes in JavaFX. The decision to deprecate builder classes was mainly due to two bugs found in Java 6 & 7 that were fixed in Java 8. The bug fixes would later cause binary compatibility issues that would break the builder classes. To get the full story please see chapter 13 in the section under JavaFX 8's Features which references links to the discussions about the issue of builder classes. Knowing this will allow you to recognize older JavaFX 2.x code so that refactoring should be painless. The first edition of this book used Builder classes quite frequently, and they are an elegant way to create JavaFX objects. Although the Builder classes are deprecated, it doesn't prevent us from creating APIs with the builder pattern in mind. To create objects without an associated builder class, just find the appropriate constructor to be instantiated or use setter methods to set the object's properties.

The Ice Cream Cone

The ice cream cone shape is created using the `javafx.scene.shape.Path` class. Each path element is created and added to the `Path` object. Also, each element is not considered a graph node (`javafx.scene.Node`). This means that path elements do not extend from the `javafx.scene.shape.Shape` class and they cannot be child nodes in a scene graph to be displayed. Figure 2-6 shows an ice cream cone shape.

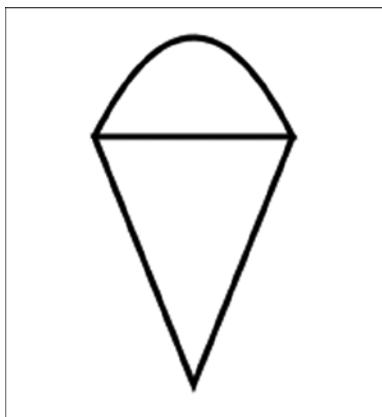


Figure 2-6. The ice cream cone is drawn using JavaFX's Path node

Path elements actually extend from the `javafx.scene.shape.PathElement` class, which is used only 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 are path elements, not Shape nodes.

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. 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().addAll(moveTo, quadCurveTo, lineTo1, lineTo2 , lineTo3);
```

The Smile

To render the smile shape, the code uses the `javafx.scene.shape.QuadCurve` class. This is similar to the cubic curve example described earlier in the first shape. Instead of two control points you only have one control point. Again, a control point influences a line by pulling the midpoint toward it. Shown in Figure 2-7 is a `QuadCurve` shape with a control point below its starting and ending points to form a smile.

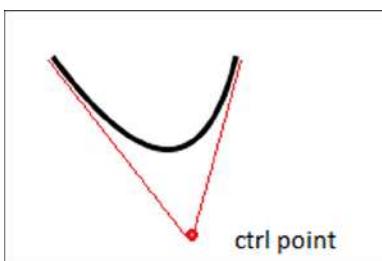


Figure 2-7. Quadratic curve

The following code draws a quadratic curve with a stroke thickness of three pixels filled with the color white:

```
// A smile
QuadCurve quad = new QuadCurve(
    50, // start x point
    50, // start y point
    125, // control x point
    150, // control y point
    150, // end x point
    50); // end y point

quad.setStrokeWidth(3);
quad.setStroke(Color.BLACK);
quad.setFill(Color.WHITE);
```

The Donut

Last is our tasty donut shape with an interesting drop shadow effect shown in Figure 2-8. This custom shape was created using geometric operations such as subtract, union, intersection, and so on. With any two shapes you can perform geometric operations to form an entirely new shape object. All of the operations can be found in the `javafx.scene.shape.Path` class.

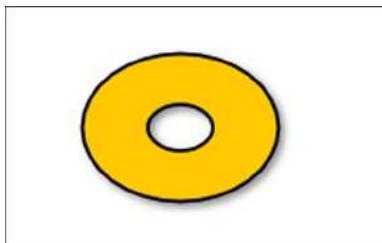


Figure 2-8. A donut shape created by using shape subtraction

To create the donut shape, you begin by creating two circular ellipse (`javafx.scene.shape.Ellipse`) instances. Subtracting the smaller ellipse (donut hole) from the larger ellipse area creates a newly derived Shape object, which is returned using the static `Path.subtract()` method. The following code snippet 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 to the donut shape. 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 the code will be drawing the donut shape once and use the `setEffect()` method to apply a `DropShadow` object instance. To cast the shadow offset, call the `setOffsetX()` and `setOffsetY()` methods. Typically if the light source is from the upper left, the shadow is shown from the lower right of the shape.

One last thing to point out is that all shapes in the example are initially drawn to be positioned underneath one another. Looking back at Listing 2-2, you'll notice that as each shape was created, its `translateY` property was set to reposition or shift each 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 1.

As each shape is rendered beneath another in this example, you may invoke the `getBoundsInParent()` method to return the information about the bounding region of a node, such as its width and height. The `getBoundsInParent()` calculation for the height and width includes the node's actual dimensions (width and height) plus any effects, translations, and transformations. For example, a shape that has a drop shadow effect increases its width by including the shadow.

Figure 2-9 is a dashed red rectangle surrounding a `Rectangle` node inside a parent node better known as the *bounding rectangle in parent* (Bounds in Parent). You will notice that the width and height calculations include *transforms*, *translates*, and *effects* applied to the shape. In the figure, the transform operation is a rotation and the effect is a drop shadow.

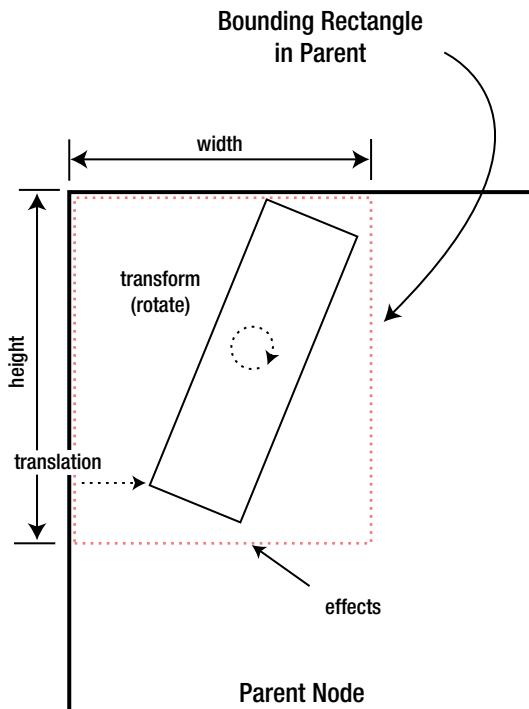


Figure 2-9. Bounding rectangle in parent

Painting Colors

I've mentioned that various drawing programs have their own tools to draw shapes. Drawing programs also provide the ability to paint shapes, using a color palette. Typically, paint programs have a paint bucket tool to fill areas on the canvas. Often, photo editing software packages have the ability to fill using a gradient color or change the lighting of colors. In JavaFX you can also apply colors (Paint) to objects using similar techniques. In this section you will see an example of a JavaFX application that will showcase three shapes filled with colors (Paint).

An Example of Color

To demonstrate common color fills, Figure 2-10 illustrates an application that displays three shapes, with different color fills. It depicts the following three shapes: ellipse, rectangle, and rounded rectangle. Each shape has a gradient fill. The first shape, the ellipse, is filled with a red radial gradient. The second is a rectangle filled with a yellow linear gradient. Finally, the rounded rectangle has a green cycle reflect gradient fill. You'll also notice a thick black line behind the yellow rectangle to illustrate the alpha level (transparency) of colors.

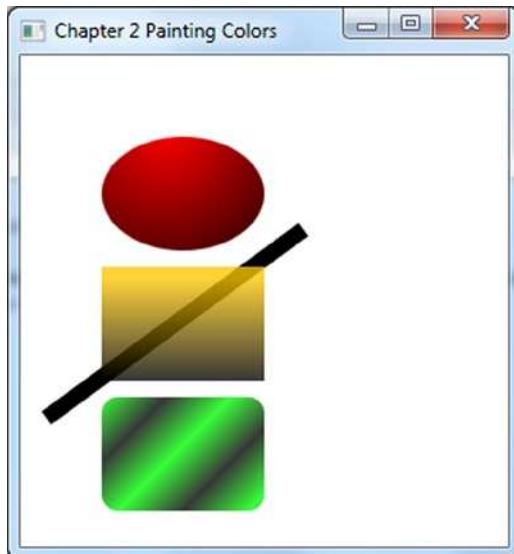


Figure 2-10. Colored shapes

Note The example to follow touches on basic techniques for creating solid colors, gradient colors, and translucent colors. There are also advanced strategies, such as the `ImagePattern` API and `Blend` API. You can read about those, if you are interested, in the API documentation viewable through Javadoc.

In JavaFX, all shapes can be filled with simple colors and gradient colors. As a reminder, according to the Javadoc all shape nodes are filled with the color black by default except for `Line`, `Polyline`, and `Path` class (descendents of `java.scene.shape.Shape`). Listing 2-3 uses the following main classes that will be used to fill the shape nodes shown in Figure 2-10:

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

Note In figure 2-10, the `blackLine`, `rectangle`, and `roundRectangle` shapes are all positioned by the `setTranslateY()` method relative to the `ellipse` shape and to one another.

Listing 2-3. ***PaintingColors.java***

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Chapter 2 Painting Colors");
    Group root = new Group();
    Scene scene = new Scene(root, 350, 300, Color.WHITE);

    // Red ellipse with radial gradient color
    Ellipse ellipse = new Ellipse(100, // center X
        50 + 70/2, // center Y
        50, // radius X
        70/2); // radius Y
    RadialGradient gradient1 = new RadialGradient(
        0, // focusAngle
        .1, // focusDistance
        80, // centerX
        45, // centerY
        120, // radius
        false, // proportional
        CycleMethod.NO_CYCLE, // cycleMethod
        new Stop(0, Color.RED), // stops
        new Stop(1, Color.BLACK)
    );
    ellipse.setFill(gradient1);
    root.getChildren().add(ellipse);
    double ellipseHeight = ellipse.getBoundsInParent()
        .getHeight();

    // thick black line behind second shape
    Line blackLine = new Line();
    blackLine.setStartX(170);
    blackLine.setStartY(30);
    blackLine.setEndX(20);
    blackLine.setEndY(140);
    blackLine.setFill(Color.BLACK);
    blackLine.setStrokeWidth(10.0f);
    blackLine.setTranslateY(ellipseHeight + 10);

    root.getChildren().add(blackLine);

    // A rectangle filled with a linear gradient with a translucent color.
    Rectangle rectangle = new Rectangle();
    rectangle.setX(50);
    rectangle.setY(50);
    rectangle.setWidth(100);
    rectangle.setHeight(70);
    rectangle.setTranslateY(ellipseHeight + 10);

    LinearGradient linearGrad = new LinearGradient(
        0, // start X
        0, // start Y
        0, // end X
```

```

    1, // end Y
    true, // proportional
    CycleMethod.NO_CYCLE, // cycle colors
    // stops
    new Stop(0.1f, Color.rgb(255, 200, 0, .784)),
    new Stop(1.0f, Color.rgb(0, 0, 0, .784)));

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

// A rectangle filled with a linear gradient with a reflective cycle.
Rectangle roundRect = new Rectangle();
roundRect.setX(50);
roundRect.setY(50);
roundRect.setWidth(100);
roundRect.setHeight(70);
roundRect.setArcWidth(20);
roundRect.setArcHeight(20);
roundRect.setTranslateY(ellipseHeight + 10 +
    rectangle.getHeight() + 10);

LinearGradient cycleGrad = new LinearGradient(
    50, // start X
    50, // start Y
    70, // end X
    70, // end Y
    false, // proportional
    CycleMethod.REFLECT, // cycleMethod
    new Stop(0f, Color.rgb(0, 255, 0, .784)),
    new Stop(1.0f, Color.rgb(0, 0, 0, .784))
);

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

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

```

When specifying color values, the `PaintingColors.java` code uses the colors in the default RGB color space. To create a color, the code uses the `Color.rgb()` method. This method takes three integer values, representing red, green, and blue components. Another overloaded method takes three integer values and a fourth parameter with a data type of double. This fourth parameter is the alpha channel, which sets the opacity of the color. This value is between zero (0) and one (1).

Note Keep in mind that there are other ways to create color, such as by specifying hue, saturation, and brightness (HSB). To create a color using HSB, you would invoke the `Color.hsb()` method. Another way to specify color values that is common to web development in HTML and CSS is the use of RGB hexadecimal values. Developers who are familiar with this convention can use the `Color.web()` method.

After drawing the ellipse shape, the code invokes the `setFill()` method using a radial gradient to give the appearance of a 3D spherical object. Next, it creates a rectangle filled with a yellow semitransparent linear gradient. Added behind the yellow rectangle is a thick black line to demonstrate the semitransparent color. Finally, the code implements a rounded rectangle filled with a green-and-black reflective linear gradient resembling 3D tubes in a diagonal direction. Each shape and its associated color fill will be discussed in more detail in the following sections.

Gradient Color

Creating a gradient color in JavaFX involves five things:

1. A starting point to begin the first stop color.
2. The end point, representing the end stop color.
3. The proportional property to specify whether to use standard screen coordinates or unit square coordinates.
4. The Cycle method to specify one of the three enums: NO_CYCLE, REFLECT, or REPEAT.
5. An array of stop (Stop) colors. Each stop containing a color will begin by painting the first stop color, then interpolating to the second stop color, and so on.

When dealing with either linear or radial gradient color, pay special attention to the proportional attribute. By setting this attribute to false, you can draw a line (the gradient axis) having a beginning point (start X, start Y) and an end point (end X, end Y) based on standard screen (x, y) coordinates.

However, if the proportional attribute is set to a value of true, the gradient axis line beginning and ending points will be represented as unit square coordinates. This means that x, y coordinates for begin and end points must be between 0.0 and 1.0 (double). This strategy is more compact and easier to define than screen coordinates on the scene graph.

Radial Gradient

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`).

Table 2-2 presents the JavaFX 8 Javadoc definitions found for the `RadialGradient` class. You can find this documentation at the following URL:

<http://docs.oracle.com/javase/8/javafx/api/javafx/scene/paint/RadialGradient.html>

Table 2-2. *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.
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 this gradient fills.
cycleMethod	CycleMethod	Cycle method applied to the gradient.
Stops	List<Stop>	Gradient's color specification.

In our example the focus angle is set to zero, the distance is set to .1, center X and Y are set to (80, 45), the radius is set to 120 pixels, proportional is set to false, the 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).

Semitransparent Gradients

Next, you will see how to create the rectangle, which has a yellow semitransparent linear gradient. For our yellow rectangle you will use a linear gradient (LinearGradient) paint.

Table 2-3 presents the JavaFX 8 Javadoc definitions found for the LinearGradient class. You can find the definitions also at the following URL:

<http://docs.oracle.com/javase/8/javafx/api/javafx/scene/paint/LinearGradient.html>

Table 2-3. *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. When set to true use unit square coordinates, otherwise use scene/screen coordinate system.
cycleMethod	CycleMethod	Cycle method applied to the gradient.
stops	List<Stop>	Gradient's color specification.

To create a linear gradient paint, you specify 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 in Figure 2-10, the yellow rectangle, you set the start X and Y to (0.0, 0.0), end X and Y to (0.0, 1.0), proportional to true, the 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 (0.0, 0.0) (the top left of a unit square) that interpolates to the color black to an end point of (0.0, 1.0) (the bottom left of a unit square).

Reflective Cycle Gradients

Finally, at the bottom of Figure 2-10 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 end X, Y values 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 stop colors. The following code snippet implements the rounded rectangle having a cycle method of reflect (CycleMethod.REFLECT):

```
LinearGradient cycleGrad = new LinearGradient(  
    50, // start X  
    50, // start Y  
    70, // end X  
    70, // end Y  
    false, // proportional  
    CycleMethod.REFLECT, // cycleMethod  
    new Stop(0f, Color.rgb(0, 255, 0, .784)),  
    new Stop(1.0f, Color.rgb(0, 0, 0, .784))  
);
```

Drawing Text

Another basic JavaFX node is the Text node. Text nodes allow you to display a string of characters onto the scene graph. To create Text nodes on the JavaFX scene graph you will use the javafx.scene.text.Text class. Because all JavaFX scene nodes extend from javafx.scene.Node they will inherit many capabilities such as the ability to be scaled, translated, or rotated.

Based on the Java inheritance hierarchy, the Text node's direct parent is a javafx.scene.shape.Shape class which provides even more capabilities than the Node class. Because a Text node is both a Node and a Shape object it is able to perform geometric operation between two shape areas such as subtract, intersect, or union. You can also clip viewport areas with a shape similar to stenciled letters.

To demonstrate drawing text, in this section you will look at a basic example of how to draw text nodes on the scene graph. This example will touch on the following three capabilities:

- Positioning Text nodes using (x, y) coordinates
- Setting a Text node's stroke color
- Rotating a Text node about its pivot point

To make things a little interesting, we will create 100 Text nodes and produce random values for the three capabilities just mentioned. Figure 2-11 shows our drawing text example in action.

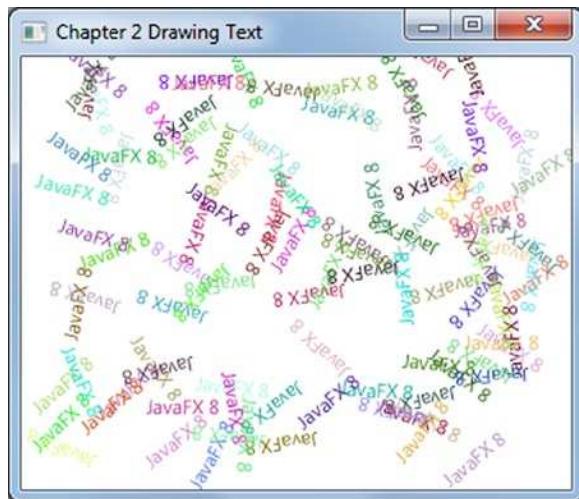


Figure 2-11. Drawing text

The `DrawingText.java` code shown in Listing 2-4 creates 100 Text nodes with randomly generated values for the following attributes:

- x, y coordinates
- RGB fill color
- Angle of rotation (degrees).

The code first creates a loop to generate random (x, y) coordinates to position Text nodes. In the loop, it creates random RGB color components between 0 and 255 to be applied to the Text nodes. Setting all components to zero produces black. Setting all three RGB values to 255 produces the color white.

Listing 2-4. DrawingText.java

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Chapter 2 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 textNode = new Text("JavaFX 8");
        textNode.setX(x);
        textNode.setY(y);
        textNode.setFill(Color.rgb(red, green, blue));
        textNode.setAngle(rand.nextInt(360));
        root.getChildren().add(textNode);
    }
}
```

```
Text text = new Text(x, y, "JavaFX 8");

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();
}
```

The rotation angle (in degrees) is a randomly generated value from 0–360 degrees, which causes the baseline of the text to be tilted. According to the API documentation the `setRotate()` method will rotate about the pivot point, which is the center of the untransformed layout bounds (`layoutBounds`) property. Basically, the pivot point is the center of a node that has no transforms (scaling, translating, shearing, rotating, and so on) applied.

Note If you need to use a combination of transforms, such as rotate, scale, and translate, take a look at the `getTransforms().add(...)` method. For more details on the difference between bounds in local, bounds in parent, and layout bounds, please see the Javadoc documentation.

The following code is used in `DrawingText.java` to create random values for a `Text` node's `x` and `y` position (baseline), 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 are applied to the `Text` nodes, which will be drawn onto the scene graph. Each `Text` node maintains a `textOrigin` property that contains the starting point of its baseline. In Latin-based alphabets, the baseline is an imaginary line underneath letters, similar to books on a bookshelf. However, some letters, such as the lowercase `j`, extend below the baseline. When specifying the `x` and `y` coordinate of the `Text` node you will be positioning the start of the baseline. In Figure 2-12, the `x` and `y` coordinate is located left of the lowercase `j` on the baseline in the text node “javafx 8.”



Figure 2-12. `Text` node's baseline (`x`, `y`) coordinate position

The following code snippet applies (x, y) coordinates, color (RGB) with an opacity .99 and a rotation (angle in degrees) to the Text node:

```
Text text = new Text(x, y, "JavaFX 8");
text.setFill(Color.rgb(red, green, blue, .99));
text.setRotate(rot);

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

You should be starting to see the power of the scene graph API in its ease of use, especially when working with Text nodes. Here you were able to position, colorize, and rotate text. To spruce things up a bit more, you will next see how to change a text's font and apply effects.

Changing Text Fonts

JavaFX's Font API enables you to change font styles and font sizes in the same way as word processing applications. To demonstrate it, I created a JavaFX application that displays four text nodes with the string value of "JavaFX 8 Intro by Example," each having a different font style. In addition to the font styles I also added effects such as a drop shadow (DropShadow) and reflection (Reflection).

Figure 2-13 shows the example's output, and Listing 2-5 shows the `ChangingTextFonts.java` source code.

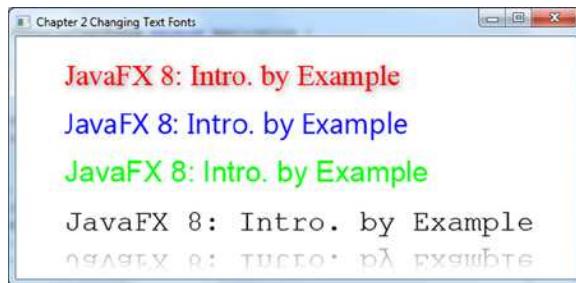


Figure 2-13. Changing text fonts

Listing 2-5. `ChangingTextFonts.java`

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Chapter 2 Changing Text Fonts");

    System.out.println("Font families: ");
    Font.getFamilies()
        .stream()
        .forEach( i -> System.out.println(i));
    System.out.println("Font names: ");
    Font.getFontNames()
        .stream()
        .forEach( i -> System.out.println(i));
```

```
Group root = new Group();
Scene scene = new Scene(root, 580, 250, Color.WHITE);

// Serif with drop shadow
Text text2 = new Text(50, 50, "JavaFX 8: 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 8: 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 8: 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 8: Intro. by Example");
Font monoFont = Font.font("Monospaced", 30);
text5.setFont(monoFont);
text5.setFill(Color.BLACK);
root.getChildren().add(text5);

// Reflection
Reflection refl = new Reflection();
refl.setFraction(0.8f);
refl.setTopOffset(5);
text5.setEffect(refl);

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

Note Listing 2-5, `ChangingTextFonts.java`, presents only the `start()` method, which is the heart of the example application. To see the full code listing please download the book's full example code from Apress.com.

With Text nodes JavaFX takes a retained-mode approach, in which nodes are using vector-based graphics rendering instead of an immediate mode rendering. The immediate mode uses bitmap graphic rendering strategies. By using vector-based graphics you will have nifty advantages over bitmap graphics. The major advantage is that it allows you to scale shapes and apply different effects without pixilation (the jaggies). For example, in an immediate mode rendering the image becomes grainy when scaled larger; however, in retained mode you will have smooth (anti-aliased) shapes. It's nice to be able to see beautiful type fonts (typography) that are smooth at all sizes.

ChangingTextFonts.java focuses on the following JavaFX classes to be applied to the Text node. The code uses the Serif, SanSerif, Dialog, and Monospaced fonts along with the drop shadow and reflection effects.

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

The code begins by setting the title on the stage. Next, you will notice the new Java 8 language lambdas feature at work, where the code lists the current system's available font families and font names. If you are new to lambdas, in Chapter 3 you will take a closer look at the concept, but for right now you can think of it as an elegant way to iterate over collections. The font family and font name list will be printed on the console output. This is very convenient for you to later experiment with different font styles. The following lines list the available fonts on the current system using Java 8's lambda syntax (as we'll discuss in Chapter 3):

```
System.out.println("Font families: ");
Font.getFamilies()
    .stream()
    .forEach( i -> System.out.println(i));
System.out.println("Font names: ");
Font.getFontNames()
    .stream()
    .forEach( i -> System.out.println(i));
```

Note In case the selected font is not installed on your system, Text will use a default font. If you can find this font, download and install it, to get the proper rendering.

After listing the available fonts, the code creates a root node with a background color for its scene. Before drawing the first Text node, let's quickly discuss how to obtain a font. To obtain a font the code invokes the static method `font()` from the `Font` class. When calling the `font()` method, you can specify the font name and the point size to return a suitable font to the caller. A font name is a string value that represents a system's supported font types. Secondly, is the point size or standard sizing value for fonts. Please refer to the Javadoc documentation to see the other ways to obtain system fonts. The following lines show the creation of a Text node instance and obtaining a 30 point Serif font. Once the font is obtained the Text node `setFont()` method is used to apply the font.

```
Text text = new Text(50, 50, "JavaFX 8: Intro. by Example");
Font serif = Font.font("Serif", 30);
text.setFont(serif);
```

Note Although, the example listing 2-5 uses absolute positioning of text nodes at some point you may want to display text nodes with the ability to wrap several text nodes while keeping their own font formatting in the same layout. To achieve this behavior refer to JavaFX 8's new TextFlow API. Refer to the Javadoc documentation at: <http://docs.oracle.com/javase/8/javafx/api/javafx/scene/text/TextFlow.html>.

Applying Text Effects

In the first Text node in the `ChangingTextFonts.java` example, the code adds a drop shadow effect. In JavaFX a drop shadow is an actual effect (`DropShadow`) object and is applied to a single Text node instance without the need of multiple images layered. The `DropShadow` object is set to be positioned based on an x, y offset in relation to the Text node. You also have the ability to set the color of the shadow; here the code will set the shadow color to gray with 0.588 opacity. Opacity is a range between 0 and 1 (double) where 0 is transparent and 1 is fully opaque. The 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 example is about setting text fonts, it also demonstrates applying a drop shadow effect to Text nodes. The `ChangingTextFonts.java` example includes yet another effect. While creating the last Text node using the monospaced font, I've applied the popular reflection effect. Calling the `setFraction()` method with 0.8f is essentially specifying that you want 80 percent of the reflection to be shown. The reflection values range from zero (0%) to one (100%). In addition to the fraction to be shown, the reflection's gap or top offset can be set. In other words the space between the opaque node portion and the reflection portion is adjusted by the `setTopOffset()` method. The top offset defaults to zero. The following code snippet implements a reflection of 80% with a float value of 0.8f and a top offset of five pixels:

```
Reflection refl = new Reflection();
refl.setFraction(0.8f);
refl.setTopOffset(5);
text5.setEffect(refl);
```

Summary

In this chapter you learned the fundamentals of drawing 2D shapes on the JavaFX scene graph. You gained insight into the difference between the Cartesian and screen coordinate systems. This furthered your knowledge of how to draw shape nodes onto the JavaFX scene graph. Our first example was the basic shape of a line using the JavaFX Line class. Here, you delved deeper into the common properties of the parent class (Shape). Some of the common properties discussed were setting the shape's stroke width, stroke color, and dash pattern. Next, you explored simple, complex, and custom shapes by drawing fun cartoon-like images. After creating shapes, you learned how to paint them using colors. You not only used standard RGB colors, you also used various built-in techniques such as a linear (LinearGradient) and radial (RadialGradient) gradient paint. Finally, you worked with JavaFX's Text node. Using the Text node you were able to obtain the available system fonts and also apply effects such as drop shadow (DropShadow) and reflection (Reflection).

In our last example which detailed changing text fonts, you learned how to iterate over a list of system fonts using Java 8's new stream API and lambda expressions. In chapter 3 you will be introduced with Java 8's lambdas and properties.

Lambdas and Properties

In this chapter you will learn about the new language feature introduced in Java 8 called *lambda expressions*, or *lambdas*, and I will also discuss JavaFX properties and binding APIs. The goal of this chapter is to demonstrate how lambdas and properties will be used in the context of JavaFX GUI applications. Having said this, I will mainly concentrate on the common features that will be used in most of the examples in this book, and will not detail every lambda and properties feature.

To get a better idea of Java's lambda roadmap, please visit the following article by Brian Goetz, who is Oracle's Java Language Architect:

<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>

The term *lambda* derives from what is known in mathematics as *lambda calculus*. The general concepts from that field are used to derive the specific behaviors that have been added to Java, which are described in this chapter. Figure 3-1 shows the Greek letter lambda, which you sometimes see in connection with the topic.



Figure 3-1. The Greek Symbol for Lambda

Lambda

Java Lambda is based on JSR (Java Specification Request) 335, “Lambda Expressions for the Java™ Programming Language.” The feature was appropriately named after lambda calculus, the formal system in mathematical logic and computer science to express computations. JSR 335, better known as Project Lambda, comprised many features, such as expressing parallel calculations on streams (the Stream API).

A primary goal of lambdas is to help address the lack in the Java language of a good way to express functional programming concepts. Languages that support functional programming concepts have the ability to create anonymous (unnamed) functions, similar to creating objects instead of methods in Java. These function objects are commonly known as *closures*. Some common languages that support closures or lambdas are Common Lisp, Clojure, Scheme, Scala, Groovy, Python, Ruby, and JavaScript.

Some will argue about what a closure is and what it is not; however the main idea is that languages that support functional programming with closure syntax are able to create anonymous functions as first-class citizens. In other words, functions or closures can be treated like objects, so that they can be assigned to variables and passed into other functions. In order to mimic closures in the previous Java world (Java 1.1 to Java 7), you would typically use anonymous inner classes to encapsulate functions (behavior) as objects. Listing 3-1 shows an example of an anonymous inner class that defines handler code for a JavaFX button when pressed.

Listing 3-1. A Button’s OnAction Set with an Anonymous Inner Class

```
Button btn = new Button();
btn.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Hello World");
    }
});
```

You will notice that this code looks very verbose just to wire up a button. Buried deep in an anonymous inner class is a single line to output text. Wouldn’t it be nice to be able to express a block of code containing the behavior you want without the need of so much boilerplate code? Look no further. Java 8 solves this issue with the use of lambda expressions (closures). To see how it looks with Java 8 lambda expressions, let’s rewrite the button handler code just shown. Listing 3-2 is our button’s EventHandler code rewritten as a Java 8 lambda expression.

Listing 3-2. A Button’s OnAction Set with a Lambda Expression

```
btn.setOnAction(event -> System.out.println("Hello World") );
```

Using lambda expressions not only makes code concise and easy to read, but the code is also likely to perform better. Actually, under the hood the compiler is capable of optimizing code and able to reduce its footprint.

Lambda Expressions

As discussed before, lambda expressions are Java 8's version of closures. In the same way other languages create closures syntactically, you will also be able to reference them and pass them into other methods. In order to achieve this ability you will later learn about *functional interfaces*. But for now, let's look at the basic syntax for lambda expressions.

Syntax

There are two ways to specify lambda expressions. The following simple examples illustrate the general forms:

```
(param1, param2, ...) -> expression;  
(param1, param2, ...) -> { /* code statements */ };
```

Lambda expressions begin with a list of parameters surrounded by parentheses, the arrow symbol `->` (a hyphen and a greater-than sign), and an expression body. Similar to Java methods, the parentheses are for parameters that are passed into an expression. When the expression doesn't accept parameters (and is said to be *empty*), the parentheses still are required. Separating the parameter list and the expression body is the arrow symbol. The expression body or code block may or may not be surrounded by curly braces. When the expression body doesn't have surrounding curly braces, it must consist of only one statement. When an expression is a one-line statement, it is evaluated and returned to the caller implicitly. Just remember that if the method requires a return type and your code block has curly braces, you must have a `return` statement. Listing 3-3 shows two equivalent lambda expressions, with and without curly braces.

Listing 3-3. One Statement with an Explicit Return, and the Other with an Implicit Return

```
x -> { return x * x; } // must explicitly return result  
x -> x * x; // evaluates and implicitly returns result
```

Another thing to note is that parameters can be optionally typed. The compiler will infer the type of the parameters depending on the context. For example, when you set a JavaFX button by using the `setOnAction()` method, the compiler can figure out the lambda parameters by matching the method's argument types (its *signature*) and its return type. Sometimes for readability you can optionally specify the type of each parameter. If you specify the type, you will be required to surround the list of parameters in parentheses. The following line specifies the type (`ActionEvent`) of the parameter for the lambda expression for a JavaFX button's `setOnAction()` method.

```
btn.setOnAction( ActionEvent event) -> System.out.println("Hello World") );
```

You will also notice that it is okay to leave off the semicolon for a one line statement that is inside methods, such as the `setOnAction()` method here. One last piece of syntactic sugar is that if you have one parameter to be passed into your lambda, the parentheses can be omitted. Following is a lambda expression that has one parameter (`event`) without parentheses.

```
btn.setOnAction( event -> System.out.println("Hello World") );
```

As a recap of the syntax variations just discussed, the statements in Listing 3-4 are all equivalent lambda expressions.

Listing 3-4. Three Syntactically Equivalent Lambda Expressions to Set Action Code on a JavaFX Button

```
btn.setOnAction( (ActionEvent event) -> {System.out.println("Hello World"); } );
btn.setOnAction( (event) -> System.out.println("Hello World") );
btn.setOnAction( event -> System.out.println("Hello World") );
```

Variable Capture

Typically in functional programming languages, the use of closures (lambda expressions) allows you to use variables outside the closure's scope. These variables are often described as *nonlocal* or *capture* variables because they aren't locally scoped to the lambda expression or method. When these nonlocal variables are used in lambda expressions, they need to be immutable. This is a key principle in the functional programming world. New to the Java 8 compiler is the ability to reference variables outside the scope of an anonymous inner class or lambda expression.

When using lambda expressions you can optionally reference variables scoped outside the closure function. Typically, prior to Java 8 these nonlocal variables needed to be *final* (immutable). This immutable state enables the compiler to better predict program behavior and thus capable of optimizing code. In Java Swing development, this occurred often when variables of the enclosing scope needed to be used inside the scope of an anonymous inner class method (local variable scope). In order to abide by the compiler rules you had to declare a variable as *final*.

The Java 8 compiler has a new capability that will infer the intent of nonlocal variables passed into anonymous inner class methods and lambda expressions to then be converted to *final* (immutable). To see how we old-school Java Swing developers (prior to Java 8) used to receive compile-time errors, please look at Listing 3-5. The scenario is a button that will modify the text of a Swing JLabel component after the button is pressed. Prior to Java 8, the code would not compile correctly, because the *label* variable needed to be declared *final*.

Listing 3-5. Code That Would Receive a Compiler Error Prior to JDK 8

```
// Prior to Java 8
JLabel label = new JLabel("Press Me"); // nonlocal variable to actionPerformed()
btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        label.setText("Pressed"); // !Compile Error! Variable label must be final.
    }
});
```

New to Java 8 is the ability to convert nonlocal variables to be *final*. In Listing 3-6 you'll notice that the code is identical except for the comments. Starting with Java 8, you can now make use of the variable capture capability with anonymous inner classes and lambda expressions.

Listing 3-6. Code That Compiles Successfully in JDK 8

```
// New in Java 8
JLabel label = new JLabel("Press Me"); // nonlocal variable
// compiler implicitly converts label as a final (immutable) variable
btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        label.setText("Pressed"); // Legal. Variable label is converted to be final
    }
});
```

Both of the code examples shown in Listing 3-6 are identical. The difference comes when using a JDK 8 compiler. When using the Java 8 compiler, the capture variable will implicitly become final (immutable), and that will occur without triggering any compilation error.

Functional Interfaces

Now that you know how to create lambda expressions, you may be wondering what exactly a lambda expression's type is. Another way to think about this is to ask questions such as the following:

- How can I assign a lambda expression to a variable?
- How can I create a method with parameters that accepts lambda expressions?

To answer these questions you will be looking at examples of Java 8's functional interfaces. Before we get into the details, however, I'll give you some context about functional programming languages and how they differ from imperative languages.

Functional vs. Imperative Languages

The advent of cloud computing has helped to popularize many functional programming languages. It became apparent there was a paradigm shift in problem-solving that involves extremely large datasets. A typical use case when applying functional programming techniques is the ability to iterate over datasets while performing computations in a distributed fashion so that load can be shared among nodes or CPU cores. In contrast, imperative programming languages gather data to then be passed into a tight for-loop to be processed. Because of how data and code are coupled, this puts a lot of the burden on one thread (core) to process so much data. The problem needs to be decomposed to allow other threads (cores) to participate in the computation, which then becomes distributed.

One of the advantages of functional programming is the ability to express functionality in a syntactically concise manner, but more important is the ability to pass functionality (lambda expressions) to methods. Being able to pass lambda expressions to methods often fosters the concept of lazy evaluation. This behavior is the same as function callback behavior (asynchronous message passing), where invocations are deferred (and thus “lazy”) until a later time. The opposite of lazy evaluation is eager evaluation. Using lazy evaluations will often increase performance by avoiding unnecessary calculations.

Another important topic to discuss is how functional interfaces are used as closures in the Java language. Instead of implementing functions as first-class types in Java, the designers/architects of the Java Language cleverly defined the notion of functional interfaces as closures. Simply put, a functional interface is basically a single abstract method (SAM). The idea of functional interfaces has been around for a very long time. For instance, those who have worked with Java threads will recall using the Runnable interface, where there is a single run() method with a void return type. The single abstract method pattern is an integral part of Java 8's lambda expressions. As an example of a functional interface, I created an interface called MyEquation with a single abstract compute() method. Once it is created, you can declare and assign variables with a lambda expression. Listing 3-7 is a functional interface that has a single abstract compute() method.

Listing 3-7. A Declared Functional Interface with a Single Abstract Method, compute()

```
// functional interface
interface MyEquation {
    double compute(double val1, double val2);
}
```

After creating a functional interface, you can declare a variable to be assigned with a lambda expression. Listing 3-8 demonstrates the assignment of lambda expressions to functional interface variables.

Listing 3-8. Two Variables of a Functional Interface, Assigned with Lambda Expressions

```
MyEquation area = (height, width) -> height * width;
MyEquation perimeter = (height, width) -> 2*height + 2*width;

System.out.println("Area = " + area.compute(3, 4));
System.out.println("Perimeter = " + perimeter.compute(3, 4));
```

The following is the output of Listing 3-8:

```
Area = 12.0
Perimeter = 14.0
```

Aggregate Operations

Another powerful use of Java 8's lambda expressions is working with collections. Java 8 introduces the new Stream API (`java.util.stream.*`), which allows you to process elements from a given source. A source can be a reference to data structures such as collections or IO channels. According to Oracle's Java 8 documentation (Javadoc) on the Stream API, “a stream is not a data structure that stores elements; instead, they carry values from a source (which could be a data structure, a generator, an IO channel, etc.) through a pipeline of computational operations.”

A pipeline is a sequence of operations (lambda expressions/functional interfaces) that can process or interrogate each element in a stream. Such operations allow you to perform aggregate tasks. Aggregate operations are similar to the way spreadsheets can execute some computation over a series of cells, such as formatting, averaging, or summing up values. To begin using aggregate operations on collections, you will first invoke the default `stream()` method on the `java.util.Collection` interface. You'll learn more about default methods shortly, but for now all collections (`Collection`) have a method `stream()` that returns a `java.util.stream.Stream` instance. Listing 3-9 is a snippet of code that obtains a stream of elements from a source collection (`List<Integer>`).

Listing 3-9. A Stream Obtained by Invoking the `stream()` Method from a Collection

```
List<Integer> values = Arrays.asList(23, 84, 74, 85, 54, 60);
Stream<Integer> stream = values.stream();
```

The common built-in aggregate operations are filter, map, and forEach. A filter allows you to pass in an expression to filter elements and returns a new Stream containing the selected items. The map operation converts (or maps) each element to another type and returns a new Stream containing items of the mapped type. For instance, you may want to map Integer values to String values of a stream. A forEach operation allows you to pass in a lambda expression to process each element in the stream.

A typical use case to exercise the three common aggregate operations would be this: given a collection of integers, filter on values greater than a threshold value (a nonlocal variable), convert filtered items to hex values, and print each hex value. Listing 3-10 implements the use case, which exercises various aggregate operations.

Listing 3-10. The Use of Aggregate Operations Over a Collection of Integers

```
// create a list of values
List<Integer> values = Arrays.asList(23, 84, 74, 85, 54, 60);
System.out.println("values: " + values.toString());

// nonlocal variable to be used in lambda expression.
int threshold = 54;
System.out.println("Values greater than " + threshold + " converted to hex:");
// using aggregate functions filter() and forEach()
values.stream()
    .filter(val -> val > threshold) // Predicate functional interface
    .sorted()
    .map(dec -> Integer.toHexString(dec).toUpperCase() ) // Consumer functional interface
    .forEach(val -> System.out.println(val)); // each output values.
```

Following is the output from Listing 3-10:

```
values: [23, 84, 74, 85, 54, 60]
Values greater than 54 converted to hex:
3C
4A
54
55
```

In Listing 3-10 the code begins by creating a list of integers as a collection. Next, it outputs the original list of elements (unmodified). Then the code declares a nonlocal variable to the lambda expression called threshold, to be used in a filter expression. Next, we obtain a stream from the values collection (via stream() method) to perform aggregate operations.

The first operation is a filter that receives a lambda expression to filter elements (integers) greater than 54 (`val > threshold`). The next operation is the `sorted()` method, which sorts elements that were returned from the filter method stream. Continuing the method chaining, notice the `map()` operation. The `map()` method operation is able to map each element in the stream from one data type to another.

In this scenario the stream elements are Integer objects, which are mapped to String objects. The map() method returns another stream, containing elements of type Stream<String>. In this example the string elements are hexadecimal values. Finally, the forEach() operation iterates over each element in the stream of string (hex) elements to be printed.

You'll notice that as each aggregate function is called, the return type is a Stream object, which allows you to method-chain operations in a way very similar to the builder pattern. To clarify, the idea of method chaining actually comes from the fluent interface pattern for API design.

Default Methods

Default methods are a new way to add default implementation methods to Java interfaces. You're probably scratching your head and thinking out loud, "Java interfaces can have implementations?!" Yes, Java 8 now has support for the concept, called *virtual extension methods*, better known as *defender methods*. Default methods have the effect of adding (extending) new behavior to interfaces without breaking compatibility. Default methods are not abstract methods but methods having implementation (code).

For example, adding new methods to Java interfaces can affect all implementation classes. Because of the strict contract of Java interfaces, the compiler forces the implementer to implement abstract methods, but classes that implement interfaces with default methods do not force the developer to implement those default methods. Rather, derived classes will acquire the behavior of the default method implementation.

An Example Case: Cats Large and Small

It might not be obvious, but the method stream() on the Collection interface used earlier for aggregate operations isn't an abstract method, but actually a Java 8 default method. Listing 3-11 is the default method stream() in the java.util.Collection interface from the Java 8 source code.

Listing 3-11. The Default Method stream() of Java 8's Collection Interface

```
default Stream<E> stream() {  
    return StreamSupport.stream(splitterator, false);  
}
```

For this example I created a domain model (consisting of classes and interfaces) about cats (grrr, meow). The cat kind or species can easily demonstrate default behavior that is found in some cat kinds and not others. For instance, the great cats such as lions, tigers, and jaguars are able to roar, and smaller cats can purr. Also, a house (domestic) cat is able to meow. To further describe these behavioral traits, let's look at Figure 3-2, a class diagram depicting default behavior between cat kinds.

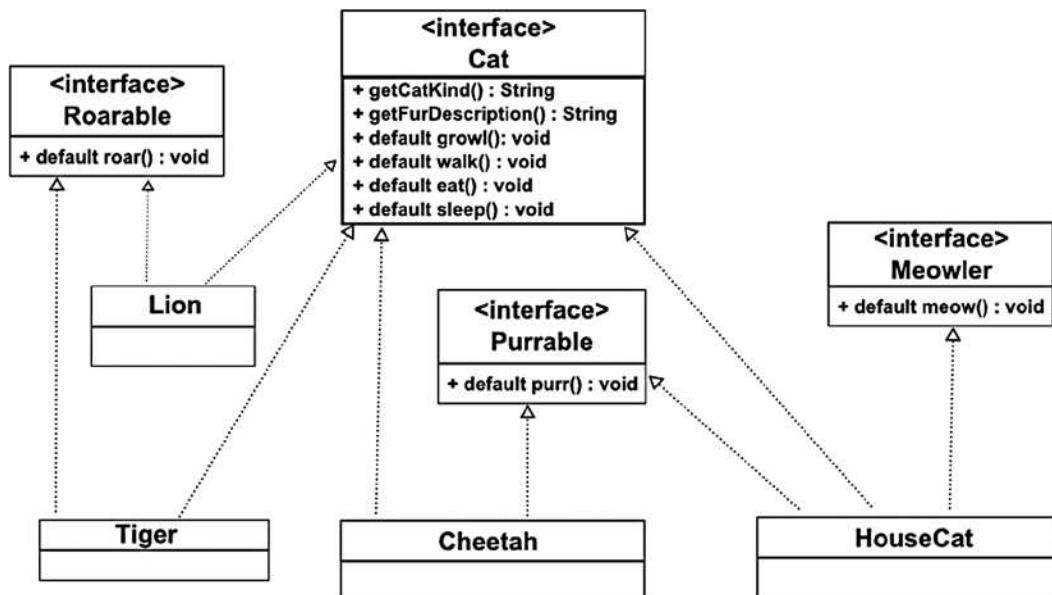


Figure 3-2. Cat domain model represented in a UML class diagram

Looking at the Tiger and Lion classes, you'll notice they each implement both Roarable and Cat interfaces. Both tigers and lions have the same behavior (can roar) via the default `roar()` method from the Roarable interface. Keep in mind that the Roarable interface doesn't have any abstract methods, but rather a default method `roar()`, which both Lion and Tiger acquire without implementing it.

In contrast, if the Roarable interface had `roar()` as an abstract method (a pure virtual function), you would have to implement `roar()` in both Tiger and Lion's concrete classes. With both having the same functionality in two places, later down the road things could become a major maintenance nightmare. That's why default methods allow you to add behavior without having to modify derived classes.

Code for the Example

To see the class diagram in Figure 3-2 implemented as code, refer to Listings 3-12 through 3-19. To implement the class diagram I will briefly outline the three parts of the code example. First is the set of cat interfaces with default method behaviors. Second are the concrete cat classes such as Lion, Cheetah and HouseCat. Last is the main example application, which demonstrates default methods in action.

In Listings 3-12 through 3-15 are the UML interfaces Roarable, Purrable, Meowler and Cat, each containing a default method.

Listing 3-12. The Roarable Interface with a Default Method roar()

```

public interface Roarable {
    default void roar() {
        System.out.println("Roar!!!");
    }
}
  
```

Listing 3-13. The Purrable Interface with a Default Method purr()

```
public interface Purrable {  
    default void purr() {  
        System.out.println("Purrrrrrrr...");  
    }  
}
```

Listing 3-14. The Meowler Interface with a Default Method meow()

```
public interface Meowler {  
    default void meow() {  
        System.out.println("MeeeeOww!");  
    }  
}
```

Listing 3-15. The Main Cat Interface with Both Abstract and Default Methods

```
/**  
 * This represents an abstract Cat containing default methods common to all cats.  
 * @author carldea  
 */  
public interface Cat {  
    String getCatKind();  
    String getFurDescription();  
  
    default void growl() {  
        System.out.println("Grrrrowl!!");  
    }  
    default void walk() {  
        System.out.println(getCatKind() + " is walking.");  
    }  
    default void eat() {  
        System.out.println(getCatKind() + " is eating.");  
    }  
    default void sleep() {  
        System.out.println(getCatKind() + " is sleeping.");  
    }  
}
```

Shown in Listings 3-16 through 3-18 are the concrete implementation classes Lion, Cheetah, and HouseCat, which utilize the interfaces just shown.

Listing 3-16. The Lion Class Implementing the Cat and Roarable Interfaces

```
public class Lion implements Cat, Roarable {  
    @Override  
    public String getCatKind() {  
        return getClass().getSimpleName();  
    }  
    @Override  
    public String getFurDescription() {  
        return "gold-brown";  
    }  
}
```

Listing 3-17. The Cheetah Class Implementing the Cat and Purrable Interfaces

```
public class Cheetah implements Cat, Purrable {  
    @Override  
    public String getCatKind() {  
        return getClass().getSimpleName();  
    }  
    @Override  
    public String getFurDescription() {  
        return "spotted";  
    }  
}
```

Listing 3-18. The HouseCat Class Implementing the Cat, Purrable, and Meowler Interfaces

```
public class HouseCat implements Cat, Purrable, Meowler {  
    @Override  
    public String getCatKind() {  
        return "Domestic Cat";  
    }  
    @Override  
    public String getFurDescription() {  
        return "mixed brown and white";  
    }  
}
```

Listing 3-19 is the main application, which invokes all of the methods of cat instances, demonstrating their behaviors.

Listing 3-19. The Main Application File (*Mixins.java*) That Executes the Code Example

```
/**Demonstrates default methods in Java 8.  
 * Mixins.java  
 *  
 * @author carldea  
 */  
public class Mixins {  
    public static void main(String[] args) {  
        Tiger bigCat = new Tiger();  
        Cheetah mediumCat = new Cheetah();  
        HouseCat smallCat = new HouseCat();  
  
        System.out.printf("%s with %s fur.\n", bigCat.getCatKind(),  
            bigCat.getFurDescription());  
        bigCat.eat();  
        bigCat.sleep();  
        bigCat.walk();  
        bigCat.roar();  
        bigCat.growl();  
        System.out.println("-----");  
        System.out.printf("%s with %s fur.\n", mediumCat.getCatKind(),  
            mediumCat.getFurDescription());  
    }  
}
```

```
mediumCat.eat();
mediumCat.sleep();
mediumCat.walk();
mediumCat.growl();
mediumCat.purr();
System.out.println("-----");
System.out.printf("%s with %s fur.\n", smallCat.getCatKind(),
    smallCat.getFurDescription());
smallCat.eat();
smallCat.sleep();
smallCat.walk();
smallCat.growl();
smallCat.purr();
smallCat.meow();
System.out.println("-----");

    }
}
```

Following is the output from Listing 3-19, which demonstrates default method invocations:

```
run:
Tiger with striped fur.
Tiger is eating.
Tiger is sleeping.
Tiger is walking.
Roar!!
Grrrrowl!!
-----
Cheetah with spotted fur.
Cheetah is eating.
Cheetah is sleeping.
Cheetah is walking.
Grrrrowl!!
Purrrrrrr...
-----
Domestic Cat with mixed brown and white fur.
Domestic Cat is eating.
Domestic Cat is sleeping.
Domestic Cat is walking.
Grrrrowl!!
Purrrrrrr...
MeeeeOww!
-----
BUILD SUCCESSFUL (total time: 0 seconds)
```

Explanation of the Code

Figure 3-2, shown earlier, depicts a UML class diagram of a cat domain model. In the top layer, you'll notice the interfaces Roarable, Cat, Purrable, and Meowler. These interfaces provide behavior for any derived class that wants to pick and choose (mixin) common behaviors.

The concrete implementation classes are Lion, Tiger, Cheetah, and HouseCat. You will notice that the main interface Cat contains abstract methods getCatKind(), getFurDescription(), and default methods walk(), eat(), and sleep(). As usual, the Java compiler will enforce classes implementing the Cat interface to implement the abstract methods getCatKind() and getFurDescription().

The default methods on the Cat interface have implementation code that serves as functionality for all cats. These default methods will provide behavior to the Cat interface while also centralizing functionality (implementation) for all derived classes. I created empty (marker) interfaces Roarable, Purrable, and Meowler, which contain respective default methods roar(), purr() and meow(). These interfaces are similar to the idea of a *mixin*, which in object-oriented languages allows developers to extend or add default behavior to any derived class or interface.

Listing 3-20 is the main class that begins by creating three cat objects: Tiger, Cheetah, and HouseCat.

Listing 3-20. Creating Three Types of Cat Instances

```
Tiger bigCat = new Tiger();
Cheetah mediumCat = new Cheetah();
HouseCat smallCat = new HouseCat();
```

After creating the cat objects, the code begins by exercising the default and abstract methods. It invokes all of the methods on each cat object. Listing 3-21 shows a cat of type Tiger with all of its methods being invoked.

Listing 3-21. All Methods Invoked on a Tiger Instance

```
System.out.printf("%s with %s fur.\n", bigCat.getCatKind(), bigCat.getFurDescription());
bigCat.eat();
bigCat.sleep();
bigCat.walk();
bigCat.roar();
bigCat.growl();
System.out.println("-----");
```

Since the rest of the cat kinds have similar tasks, I won't discuss them any further as I believe you have a handle on things. If you've gotten this far, you're a real trooper. Whenever I learn a new skill or concept, I'm always reminded of the programming proverb, "If all you have is a hammer, everything looks like a nail". Getting to learn new language features can often be difficult, especially when you've solved problems for so long without using these new and powerful concepts in Java.

I've only scratched the surface, so I encourage you to dig deeper into the features of Java 8. Now that you've been able to equip yourself with new-found powers of lambdas, let's get back on track with JavaFX.

Properties and Binding

Properties are basically wrapper objects for JavaFX-based object attributes such as String or Integer. Properties allow developers to add listener code to respond when the wrapped value of an object has changed or is flagged as invalid. Also, property objects can be bound to one another. Binding behavior allows properties to update or synchronize their values based on a changed value from another property.

UI Patterns

Before discussing JavaFX's properties and bindings APIs, I would like to share with you a little bit about UI patterns. When developing GUI applications you will inevitably encounter UI architectural framework concepts such as model view controller (MVC), model view presenter (MVP), or model view view-model (MVVM).

Depending on who you talk to, you might get different explanations; however, these concepts all address the issue of how best to handle synchronization between the model and view. What this means is that when a user interacts (input) with the UI (view) the underlying backend data store (the model) is automatically updated, and vice-versa.

Without trying to oversimplify the concepts, I will refer you to the actual UI patterns that are involved. The main UI patterns involved are the *Supervising Controller*, *Presentation Model*, and *Mediator*. If you are interested in understanding more about UI patterns, please read “GUI Architectures” by Martin Fowler at: <http://martinfowler.com/eaaDev/uiArchs.html>.

Because these patterns have been heavily discussed over the years, the JavaFX team has designed and implemented APIs to overcome problems that arose with these UI scenarios. In this section you will be learning how to use JavaFX properties and bindings to synchronize between your GUI and your data objects. Of course, I can't possibly cover all of the use-case scenarios involving the UI patterns just mentioned, but hopefully I can give you the basics to get you on the right path.

Properties

Before the JavaFX properties API, Java Swing developers adhered to the JavaBean convention (specification), which specifies that objects will contain privately scoped instance variables which have associated getter (accessor) and setter (mutator) methods. For instance, a User class might have a private instance variable password of type String. The associated getter and setter would be the getPassword() and setPassword() methods, respectively. Listing 3-22 shows a User class that follows the older JavaBean convention.

Listing 3-22. A Simple JavaBean Class with Getter and Setter Methods

```
public class User {  
    private String password;  
    public String getPassword() {  
        return password;  
    }
```

```
public void setPassword(String password) {  
    this.password = password;  
}  
}
```

As a developer who follows the JavaBean convention, you will quickly see the naming convention for the property, which is based on the getter and setter method names and not the private variable name. In other words, if the getter method were named `getPwd()`, the property would be named `pwd` and would have nothing to do with the attribute named `password`.

The JavaBeans specification also provided an API that has the concept of property change support (`java.beans.PropertyChangeSupport`), which allowed the developer to add handler (listener) code when the property changed. At the time, this solved only part of the issue of property change support. The JavaBeans API started showing its age when it came to binding variables and working with the Collections API. Although the JavaBeans API became a standard way to build domain objects, it still lacked robust ways to synchronize the domain models and views within a GUI application as described earlier.

Through the years the JavaBean specification and API gave rise to many third-party tools and frameworks to ease the developer experience. However, wiring up GUI components to JavaBeans using unidirectional or bidirectional binding still got pretty complicated. At times, developers would have resources that were not properly released, which led to object leaks. Developers quickly realized they needed a better way to bind and unbind GUI controls to be wired up to properties.

Types of JavaFX Properties

Let's fast forward to JavaFX's Properties API to see how it handles the common issues. We'll first discuss the different types of JavaFX-based properties. There are two types to be concerned about:

- Read/Writable
- Read-Only

In short, JavaFX's properties are wrapper objects holding actual values while providing change support, invalidation support, and binding capabilities. I will address binding later, but for now let's examine commonly used property classes.

Properties are wrapper objects that have the ability to make values accessible as read/writable or read-only. All wrapper property classes are located in the `javafx.beans.property.*` package namespace. Listed here are commonly used property classes. To see all of the property classes, please refer to the documentation in Javadoc.

- `javafx.beans.property.SimpleBooleanProperty`
- `javafx.beans.property.ReadOnlyBooleanWrapper`
- `javafx.beans.property.SimpleIntegerProperty`
- `javafx.beans.property.ReadOnlyIntegerWrapper`
- `javafx.beans.property.SimpleDoubleProperty`
- `javafx.beans.property.ReadOnlyDoubleWrapper`

- `javafx.beans.property.SimpleStringProperty`
- `javafx.beans.property.ReadOnlyStringWrapper`

The properties that have a prefix of Simple and a suffix of Property are the read/writable property classes, and the classes with a prefix of ReadOnly and a suffix of Wrapper are the read-only properties. Later, you will see how to create a JavaFX bean using these commonly used properties, but for now let's examine read/writable properties.

Read/Writable Properties

Read/writable properties are, as the name suggests, property values that can be both read and modified. As an example, let's look at JavaFX string properties. To create a string property that is capable of both readable and writable access to the wrapped value, you will use the `javafx.beans.property.SimpleStringProperty` class. Listing 3-23 is a code snippet that demonstrates an instance of a `SimpleStringProperty` class and modifies the property via the `set()` method.

Listing 3-23. Creating a StringProperty Instance

```
StringProperty password = new SimpleStringProperty("password1");
password.set("1234");
System.out.println("Modified StringProperty " + password.get() ); // 1234
```

Here a declared variable `password` of type `StringProperty` is assigned to an instance of a `SimpleStringProperty` class. It's always a good idea when declaring variables to be more abstract in object-oriented languages. Thus, referencing a `StringProperty` exposes fewer methods of the implementation class (`SimpleStringProperty`). Also notice that the actual value is the string “`password1`”, which is passed into the constructor of the `SimpleStringProperty` class. You will later discover other convenient constructor methods when working with JavaBeans and Property objects.

In the case of reading the value back, you would invoke the `get()` method (or `getValue()`), which returns the actual wrapped value(`String`) to the caller. To modify the value you simply call the `set()` method (or `setValue()`) by passing in a string.

Read-Only Properties

To make a property read-only you would use the wrapper classes that are prefixed with `ReadOnly` from the `javafx.beans.property.*` package. To create a property to be read-only you will need to take two steps. First is to instantiate a read-only wrapper class and invoke the method `getReadOnlyProperty()` to return a true read-only property object. Listing 3-24 creates a read-only string property.

Listing 3-24. Creating a Read-Only String Property

```
ReadOnlyStringWrapper userName = new ReadOnlyStringWrapper("jamestkirk");
ReadOnlyProperty readOnlyUserName = userName.getReadOnlyProperty();
```

This code snippet actually takes two steps to obtain a read-only string property. You will notice the call to `getReadOnlyProperty()`, which returns a read-only copy (synchronized) of the property. According to the Javadoc API, the `ReadOnlyStringWrapper` class “creates two properties that are synchronized. One property is read-only and can be passed to external users. The other property is read- and writable and should be used internally only.” Knowing that the other property can be read and written could allow a malicious developer to cast the object to type `StringProperty`, which then could be modified at will. From a security perspective you should be aware of the proper steps to create a true read-only property.

JavaFX JavaBean

Now that you’ve seen the JavaBean specification’s approach to creating domain objects, I will rewrite the earlier JavaBean User class to use JavaFX properties instead. In the process of rewriting the bean I also wanted to add an additional read-only property called `userName` to demonstrate the read-only property behavior. Listing 3-25 shows the User class rewritten to use JavaFX properties.

Listing 3-25. The File User.java Rewritten as a JavaFX Bean

```
import javafx.beans.property.*;
public class User {
    private final static String USERNAME_PROP_NAME = "userName";
    private final ReadOnlyStringWrapper userName;
    private final static String PASSWORD_PROP_NAME = "password";
    private StringProperty password;
    public User() {
        userName = new ReadOnlyStringWrapper(this,
USERNAME_PROP_NAME, System.getProperty("user.name"));
        password = new SimpleStringProperty(this, PASSWORD_PROP_NAME, "");
    }

    public final String getUserName() {
        return userName.get();
    }
    public ReadOnlyStringProperty userNameProperty() {
        return userName.getReadOnlyProperty();
    }

    public final String getPassword() {
        return password.get();
    }
    public final void setPassword(String password) {
        this.password.set(password);
    }
    public StringProperty passwordProperty() {
        return password;
    }
}
```

You may notice some obvious differences in the way I instantiated the `ReadOnlyStringWrapper` and `SimpleStringProperty` classes. Similar to the JavaBean property change support, JavaFX properties have constructors that will allow you to specify the bean itself, its property name, and its value. As a simple example, Listing 3-26 shows the instantiations of a read-only and a read-writable property using the JavaFX property change support-based constructors. The `userName` variable is assigned a read-only property, and the `password` variable is assigned a read/writable property.

Listing 3-26. Instantiating a Read-Only and a Read-Writable Property

```
userName = new ReadOnlyStringWrapper(this, USERNAME_PROP_NAME, System.getProperty("user.name"));
password = new SimpleStringProperty(this, PASSWORD_PROP_NAME, "");
```

It is good practice to use the property change support-based constructors for many reasons, such as third-party tooling and testing. But most of all, when dealing with property change support, you will need access to the bean and its other properties.

One last thing to mention is that in Listing 3-25 you'll also notice the getter and setter methods are final. Making the getter and setter final prevents any derived classes from overriding and possibly changing the underlying property.

Property Change Support

Property change support is the ability to add handler code that will respond when a property changes. JavaFX property objects contain an `addListener()` method. This method will accept two types of functional interfaces, `ChangeListener` and `InvalidationListener`. Recall that functional interfaces are single abstract methods which are expressed using the Java lambda syntax. All JavaFX properties are descendants of the `ObservableValue` and `Observable` interfaces (method overloading), which provide the `addListener()` methods for `ChangeListener` and `InvalidationListener`, respectively. One last thing to point out is that it is important to clean up listeners by removing them. To remove them you will invoke the `removeListener()` method by passing into it a referenced (named) listener as opposed to an anonymous inner class or anonymous lambda expression.

Listing 3-27 shows how to create a `ChangeListener` to be registered with a property. As the property's value changes, the `change()` method will be invoked. I've provided the implementations for both the anonymous inner class and lambda expression-style syntax for you to compare.

Listing 3-27. Adding a ChangeListener Instance to Respond When the Property Value Changes

```
SimpleIntegerProperty xProperty = new SimpleIntegerProperty(0);

// Adding a change listener (anonymous inner class)
xProperty.addListener(new ChangeListener<Number>(){
    @Override
    public void changed(ObservableValue<? extends Number> ov, Number oldVal, Number newVal) {
        // code goes here
    }
});

// Adding a change listener (lambda expression)
xProperty.addListener((ObservableValue<? extends Number> ov, Number oldVal, Number newVal) -> {
    // code goes here
});
```

Listing 3-28 shows how to create an `InvalidationListener` to be registered with a property. As the property's value changes, the `invalidated()` method will be invoked. I've provided the implementations for both the anonymous inner class and lambda expression-style syntax for you to compare.

Listing 3-28. Adding an `InvalidationListener` Instance to Respond When the Property Value Changes

```
// Adding a invalidation listener (anonymous inner class)
xProperty.addListener(new InvalidationListener() {
    @Override
    public void invalidated(Observable o) {
        // code goes here
    }
});

// Adding a invalidation listener (lambda expression)
xProperty.addListener((Observable o) -> {
    // code goes here
});
```

Often developers become confused about the difference between a `ChangeListener` and an `InvalidationListener`. In Listing 3-28 you'll notice that using a `ChangeListener` you will get the `Observable` (`ObservableValue`), the old value, and the new value, while using the `InvalidationListener` only gets the `Observable` object (property).

In the Javadoc documentation, the difference between a `ChangeListener` and `InvalidationListener` is described as follows:

“A change event indicates that the value has changed. An invalidation event is generated, if the current value is not valid anymore. This distinction becomes important, if the `ObservableValue` supports lazy evaluation, because for a lazily evaluated value one does not know if an invalid value really has changed until it is recomputed. For this reason, generating change events requires eager evaluation while invalidation events can be generated for eager and lazy implementations.”

The `InvalidationListener` provides a way to mark values as invalid but does not recompute the value until it is needed. This is often used in UI layouts or custom controls, where you can avoid unnecessary computations when nodes don't need to be redrawn/repositioned during a layout request or draw cycle. When using the `ChangeListener`, you normally want eager evaluation such as the validation of properties on a form-type application. That doesn't mean you can't use `InvalidationListeners` for validation of properties; it just depends on your performance requirements and exactly when you need the new value to be recomputed (evaluated). When you access the observable value, it causes the `InvalidationListener` to be eager.

Binding

Simply put, binding has the idea of at least two values (properties) being synchronized. This means that when a dependent variable changes, the other variable changes. JavaFX provides many binding options that enable the developer to synchronize between properties in domain objects and GUI controls. In this section you will learn about three binding strategies when binding property objects.

In this section you will be focusing on the following JavaFX API packages:

- `javafx.beans.binding.*`
- `javafx.beans.property.*`

Binding of properties is quite easy to do. The only requirement is that the property invoking the bind() method must be a read/writeable property. To bind a property to another property, you will invoke the bind() method. This method will bind in one direction (unidirectional). For instance, when property A binds to property B the change in property B will update property A, but not the other way. If A is bound to B you can't update A, as you'll get a RuntimeException: A bound value cannot be set.

The following are three additional binding strategies to consider using in JavaFX's Properties API:

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

Bidirectional Binding

Bidirectional binding allows you to bind properties with the same type allowing changes on either end while keeping a value synchronized. When binding bi-directionally, it's required that both properties must be read/writable. It's pretty easy to bind two properties bidirectionally through the use of the `bindBidirectional()` method. To demonstrate, I've created a simple example.

Listing 3-29 is a Contact bean having a property `firstName` that is bound bidirectionally to a local variable `fname` of type `StringProperty`.

Listing 3-29. Bidirectional Binding between the `firstName` Property and a Regular String Property Variable

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

contact.firstNameProperty().set("Play");
fname.set("Jane");

System.out.println("contact.firstNameProperty = " + contact.firstNameProperty().get() );
System.out.println("fname = " + fname.get() );
```

The output of this code snippet is as follows:

```
contact.firstNameProperty = Jane
fname = Jane
```

High-level Binding

Introduced in JavaFX 2.0 is a fluent interface API to bind properties. The fluent APIs are methods that allow you to perform operations on properties using English-like method names. For example, if you have a numeric property, there would be methods like `multiply()`, `divide()`, `subtract()`,

and so on. Another example would be a string property having methods like `isEqualTo()`, `isNotEqualTo()`, `concat()`, and similar. As an example, Listing 3-30 shows how to create a property that represents the formula for the area of a rectangle.

Listing 3-30. Creating a Property (area) Using the High-Level Binding Strategy for Calculating the Area of a Rectangle

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

NumberBinding area = width.multiply(height);
```

This code demonstrates high-level binding by using the fluent API from the `javafx.beans.binding.IntegerExpression` parent interface. This example binds by using the `multiply()` method, which returns a `NumberBinding` containing the computed value. What's nice is that the binding is lazy-evaluated, which means the computation (multiplying) doesn't occur unless you invoke the property's `(area)` value via the `get()` (or `getValue()`) method. For all available fluent interfaces, please see the Javadoc for `javafx.beans.binding.*` packages.

Low-Level Binding

When using low-level binding, you would use a derived `NumberBinding` class, such as a `DoubleBinding` class for values of type `Double`. 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. The difference between high-and low-level binding is that high-level uses methods such as `multiply()`, `subtract()` instead of the operators `*` and `-`. Listing 3-31 shows how to create a low-level binding for the formula for the volume of a sphere.

Listing 3-31. A Property Binding (volumeOfSphere) Using the Low-Level Binding Strategy

```
DoubleProperty radius = new SimpleDoubleProperty(2);
DoubleBinding volumeOfSphere = new DoubleBinding() {
    {
        super.bind(radius); // initial bind
    }

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

If you've gotten this far, you are definitely ready to begin building GUIs. So far you've had a chance to learn about lambda expressions, properties, and bindings. Next, I want to close out this chapter with an example showing how to build a snazzy-looking logon dialog window that makes use of shapes, lambda expressions, properties, and bindings (with a hint of UI controls).

A Logon Dialog Example

Logon dialogs are typical of any application when authenticating users. In this example I created a simple form application that will involve all of the skills you've learned from Chapter 2 and in this chapter thus far. Before getting deep into the code, which you'll see in Listing 3-32, let's look at the logon dialog. Shown in Figure 3-3 is the initial display of the logon dialog window the code will create.



Figure 3-3. Initial display of the logon dialog window

Following are the instructions for interacting with the application.

1. The user will have three attempts to enter the correct password. Figure 3-4 shows the user making the first attempt to log on. (Note: The password is "password1.")



Figure 3-4. The user is entering the password into the logon dialog window

2. As the user enters the password, they have an opportunity to hit the Enter key. If the password is incorrect, a red X appears to the right of the password text field. Figure 3-5 shows an invalid logon.



Figure 3-5. The user has pressed Enter with an invalid password

3. If the user types the correct password in real time a green check appears. After the green check appears, the user may hit the Enter key to be granted access. Figure 3-6 shows the user entering a valid password.



Figure 3-6. The user has entered a valid password

Login Dialog Source Code

Listing 3-32 is the source code for the JavaFX example logon dialog application.

Listing 3-32. The JavaFX Example Logon Dialog Application

```
package jfx8ibe;

import javafx.application.*;
import javafx.beans.property.*;
import javafx.geometry.Pos;
import javafx.scene.*;
import javafx.scene.control.PasswordField;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;
import javafx.scene.text.*;
import javafx.stage.*;

/**
 * A login form to demonstrate lambdas, properties, and bindings.
 * @author cdea
 */
public class FormValidation extends Application {

    private final static String MY_PASS = "password1";
    private final static BooleanProperty GRANTED_ACCESS = new SimpleBooleanProperty(false);
    private final static int MAX_ATTEMPTS = 3;
    private final IntegerProperty ATTEMPTS = new SimpleIntegerProperty(0);

    @Override
    public void start(Stage primaryStage) {
        // create a model representing a user
        User user = new User();

        // create a transparent stage
        primaryStage.initStyle(StageStyle.TRANSPARENT);

        Group root = new Group();
        Scene scene = new Scene(root, 320, 112, Color.rgb(0, 0, 0));
        primaryStage.setScene(scene);
    }
}
```

```
// all text, borders, svg paths will use white
Color foregroundColor = Color.rgb(255, 255, 255, .9);

// rounded rectangular background
Rectangle background = new Rectangle(320, 112);
background.setX(0);
background.setY(0);
background.setArcHeight(15);
background.setArcWidth(15);
background.setFill(Color.rgb(0, 0, 0, .55));
background.setStrokeWidth(1.5);
background.setStroke(foregroundColor);

// a read only field holding the user name.
Text userName = new Text();
userName.setFont(Font.font("SanSerif", FontWeight.BOLD, 30));
userName.setFill(foregroundColor);
userName.setSmooth(true);
userName.textProperty()
    .bind(user.userNameProperty());

// wrap text node
HBox userNameCell = new HBox();
userNameCell.prefWidthProperty()
    .bind(primaryStage.widthProperty()
        .subtract(45));
userNameCell.getChildren().add(userName);

// pad lock
SVGPath padLock = new SVGPath();
padLock.setFill(foregroundColor);
padLock.setContent("M24.875,15.334v-4.876c0-4.894-3.981-8.875-8.875s-8.875,3.981-
8.875,8.875v4.876H5.042v15.083h21.916V15.334H24.875zM10.625,10.458c0-2.964,2.411-5.375,5.375-
5.375s5.375,2.411,5.375,5.375v4.876h-10.75V10.458zM18.272,26.956h-4.545l1.222-3.667c-0.782-0.389-
1.324-1.188-1.324-2.119c0-1.312,1.063-2.375,2.375-2.375s2.375,1.062,2.375,2.375c0,0.932-0.542,1.73-
1.324,2.119L18.272,26.956z");

// first row
HBox row1 = new HBox();
row1.getChildren()
    .addAll(userNameCell, padLock);

// password text field
PasswordField passwordField = new PasswordField();
passwordField.setFont(Font.font("SanSerif", 20));
passwordField.setPromptText("Password");
passwordField.setStyle("-fx-text-fill:black; "
    + "-fx-prompt-text-fill:gray; "
    + "-fx-highlight-text-fill:black; "
    + "-fx-highlight-fill: gray; "
    + "-fx-background-color: rgba(255, 255, 255, .80); "');
```

```
passwordField.prefWidthProperty()
    .bind(primaryStage.widthProperty()
        .subtract(55));
user.passwordProperty()
    .bind(passwordField.textProperty());

// error icon
SVGPath deniedIcon = new SVGPath();
deniedIcon.setFill(Color.rgb(255, 0, 0, .9));
deniedIcon.setStroke(Color.WHITE);//
deniedIcon.setContent("M24.778,21.419 19.276,15.917 24.777,10.415 21.949,7.585 16.447,13.087
10.945,7.585 8.117,10.415 13.618,15.917 8.116,21.419 10.946,24.248 16.447,18.746 21.948,24.248z");
deniedIcon.setVisible(false);

SVGPath grantedIcon = new SVGPath();
grantedIcon.setFill(Color.rgb(0, 255, 0, .9));
grantedIcon.setStroke(Color.WHITE);//
grantedIcon.setContent("M2.379,14.729 5.208,11.899 12.958,19.648 25.877,6.733 28.707,9.561
12.958,25.308z");
grantedIcon.setVisible(false);

StackPane accessIndicator = new StackPane();
accessIndicator.getChildren()
    .addAll(deniedIcon, grantedIcon);
accessIndicator.setAlignment(Pos.CENTER_RIGHT);

grantedIcon.visibleProperty().bind(GRANTED_ACCESS);

// second row
HBox row2 = new HBox(3);
row2.getChildren()
    .addAll(passwordField, accessIndicator);
HBox.setHgrow(accessIndicator, Priority.ALWAYS);

// user hits the enter key
passwordField.setOnAction(actionEvent -> {
    if (GRANTED_ACCESS.get()) {
        System.out.printf("User %s is granted access.\n",
            user.getUserName());
        System.out.printf("User %s entered the password: %s\n",
            user.getUserName(), user.getPassword());
        Platform.exit();
    } else {
        deniedIcon.setVisible(true);
    }
    ATTEMPTS.set(ATTEMPTS.add(1).get());
    System.out.println("Attempts: " + ATTEMPTS.get());
});
```

```
// listener when the user types into the password field
passwordField.textProperty().addListener((obs, ov, nv) -> {
    boolean granted = passwordField.getText()
        .equals(MY_PASS);
    GRANTED_ACCESS.set(granted);
    if (granted) {
        deniedIcon.setVisible(false);
    }
});

// listener on number of attempts
ATTEMPTS.addListener((obs, ov, nv) -> {
    if (MAX_ATTEMPTS == nv.intValue()) {
        // failed attempts
        System.out.printf("User %s is denied access.\n",
            user.getUserName());
        Platform.exit();
    }
});

VBox formLayout = new VBox(4);
formLayout.getChildren().addAll(row1, row2);
formLayout.setLayoutX(12);
formLayout.setLayoutY(12);

root.getChildren().addAll(background, formLayout);

primaryStage.show();
}

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

Explanation of the Code

The JavaFX application in Listing 3-32 simulates a logon dialog window. The code mainly demonstrates binding JavaFX JavaBeans properties with UI controls by reusing the User class that was shown earlier, in Listing 3-25.

Let's begin by describing the class variables. The variable MY_PASS contains a hardcoded string of the password of "password1". Next declared is the GRANTED_ACCESS variable, of type SimpleBooleanProperty class. This variable will be later bound from the green check (grantedIcon) node's visible property. Basically, as the user enters the correct keystrokes for the password, the GRANTED_ACCESS become true, which will internally set the green check (SVGPath) node's visible property to true, thus displaying the icon in the scene graph. When the password is incorrect, the visible property becomes false, which hides the green check node (SVGPath). The constant variable MAX_ATTEMPTS is the maximum number of attempts the user can try to log in. Last, the ATTEMPTS variable of type IntegerProperty holds the current number of attempts the user has tried to log into the dialog by hitting the Enter key.

In the start() method, a User object is created to be used as the domain object (model) to be synchronized with the GUI form. Next, the stage is set to be transparent by invoking the Stage.initStyle(StageStyle.TRANSPARENT); method. This will allow us to create translucent and irregularly shaped windows without the decoration of the native OS title bar. After initializing the stage, I also needed to create a Scene with a root node of type Group that will allow me to add child nodes. Even though the root node is transparent, it will need to be the same size as our logon screen. The logon screen is actually just a rounded rectangle having an arc height and width of 15 with a fill color of black with an opacity level of 55%. To give things a consistent look I've created a foregroundColor variable, containing a color of white with an opacity of 90 percent. The rectangle's stroke color will use the variable foregroundColor to give it a white outline.

After setting up the initial rectangular background, the code creates a Text node that will represent the user's user name as a read-only field (per the User class). To obtain the user's real username, you can call the System.getProperty("user.name") method. The text node is set up with a 30 point SanSerif font having a fill color of the foregroundColor (white) variable. The text node's text property is bound to the User object's read-only userName property. Next, I created an HBox layout that will hold the Text node userName. The HBox (userNameCell) will be binding its preferred width property by using the fluent API to ensure that it doesn't hog up all the horizontal space and allow the padlock icon to have room to be placed beside it. Shown in Figure 3-3 in the upper-right corner of the logon dialog window is the padlock icon. The following code snippet shows an HBox containing the TextNode (username) from Listing 3-32.

```
// wrap text node
HBox userNameCell = new HBox();
userNameCell.prefWidthProperty().bind(primaryStage.widthProperty().subtract(45));
userNameCell.getChildren().add(userName);
```

I have said very little about the HBox layout control here, because I will discuss it further in Chapter 4, on Layout and UI Controls. Another new node I have not mentioned is JavaFX's SVGPath node to create the padlock icon. Recall that in Chapter 2 we discussed generating complex shapes using path elements to create an ice cream cone, as depicted in Figure 2-4. Having said this, the SVGPath class is very similar to the Path node, except that the child path elements are expressed in W3C SVG path notation as a string. To render a path the string representing the path notation is passed into the setContent() method on the SVGPath node. To see how to use SVG paths, visit <http://www.w3.org/Graphics/SVG>. To create the padlock icon, I headed over to the famous website of the JavaScript library Raphaël by Dmitry Baranovskiy. There I was able to obtain very nice SVG icons, at <http://raphaeljs.com/icons>. Because an SVGPath node is also a JavaFX shape, I simply filled it with the color white (foregroundColor) by using the setFill() method. Next, the code creates an HBox as a container for the first row (row1) to hold the userNameCell and padLock nodes.

After creating the first row, the code proceeds by creating the PasswordField UI control. Like the Text node, the PasswordField UI control's font can be set. I added a 20 point SanSerif font to the password field. Next, the code will set the prompt text to let the user know what to type into the field. The prompt text is shown in Figure 3-3 of the initial logon dialog with the prompt text as "Password."

For some final decorating of the password field I used JavaFX CSS styling, which you will get a chance to see in later chapters. The CSS style I applied to the password field was a white background with opacity of 80 percent. Still working with the password field, I used the fluent interface subtract() to bind the preferred width to subtract 55px to allow room for the stack pane that contains the granted and denied icons. Last, the user object's passwordProperty is bound to the PasswordField's textProperty. This is a unidirectional bind, which means that as the user enters text into the password field the user object's password property changes and not the other way around.

Next, I create two more icons using the SVGPath node that represents an X and a check mark, denoting denied (deniedIcon) and granted (grantedIcon) access icons, respectively. These icons will be placed in a StackPane layout node. The StackPane layout allows child nodes to be stacked, hence the name. In this scenario the deniedIcon and grantedIcon nodes are stacked on top of each other. This trick will allow us to flip between icons using the setVisible() method. As a whole (stack pane) node, it will be positioned to the right of the password field. As the user types into the password field the Boolean property GRANTED_ACCESS is updated, which updates the grantedIcon's visible property. The following code statement shows the visible property bound to the GRANTED_ACCESS property.

```
grantedIcon.visibleProperty().bind(GRANTED_ACCESS);
```

Completing the second row, I create another HBox similar to our first row, this time holding the password field and the stack pane containing the icons (denied and granted icons).

Next, the code continues by wiring up the UI controls and adding property change support (ChangeListeners) to various properties. The code proceeds with the passwordField by adding an action as a functional interface for the method setOnAction() when the user hits the Enter key. The following handler code is set on the password field when the user hits the Enter key.

```
passwordField.setOnAction(actionEvent -> {
    if (GRANTED_ACCESS.get()) {
        System.out.printf("User %s is granted access.\n", user.getUserName());
        Platform.exit();
    } else {
        deniedIcon.setVisible(true);
    }
    ATTEMPTS.set(ATTEMPTS.add(1).get());
    System.out.println("Attempts: " + ATTEMPTS.get());
});
```

Essentially, if the password matches and the user hits Enter, the program exits and outputs the following:

```
User cdea is granted access.
```

You'll also notice that the code increments the ATTEMPTS property by invoking the fluent interface method add().

Still working with the password field, I added a change listener to listen for keystrokes from the user. As the user types into the password field the password is compared against the stored (hardcoded) password. If the password is valid, the code sets the GRANTED_ACCESS property to true and hides the deniedIcon icon via the setVisible(false) method.

Finally, I added the last bit of wiring for when a user has too many failed attempts. I basically added a change listener onto the ATTEMPTS property. The change listener will compare MAX_ATTEMPTS against ATTEMPTS to see if they equal. If so, it will invoke the JavaFX's Platform.exit() method to gracefully exit the application. The rest is assembling the rows into a VBox layout and adding it to the root node for the scene to be shown.

Summary

In this chapter we've covered a lot of ground with properties and bindings, especially relating to new Java 8 language features. You were introduced to the new features in Java 8 such as lambda expressions, stream APIs, and default methods. Starting with Java lambdas, you were able to learn the syntax and make use of functional interfaces. Next, you were able to use the stream API and common aggregate functions. After an example of using the stream APIs to manipulate elements in a collection, you then learned how to extend behavior on Java interfaces with default methods. Then we explored UI patterns, the Properties API, and the idea of change support. With a solid understanding of working with properties, you then learned about various binding strategies. To wrap up the chapter, you saw an example of how to create a logon dialog window, which employed all the concepts you've learned so far. Next, in Chapter 4 you'll explore JavaFX UI layout and UI controls.

Layouts and UI Controls

As more and more graphically rich applications find their way into our lives such as desktops, smart phones and tablets devices, it is very important to design GUI applications that provide better usability across the different form factors. With the increase of the number different screen sizes a UI developer will need to learn how to create applications with UI layout management in mind.

In this chapter we will be discussing the most common JavaFX layouts used in GUI applications of today. After layouts are discussed in the first half of this chapter an example will be presented showing a form-based application. The form-based application demonstrates the resizing of child UI controls as the parent window is being resized. The last half of the chapter will be about commonly used JavaFX UI controls. JavaFX UI controls include menus, menu items, list views, table views and progress indicators.

Layouts

One of the greatest challenges in building user interfaces is the laying out of UI controls onto the display area. In GUI applications it is ideal to allow the user to resize a viewable area while UI controls also resize in order to provide a pleasant user experience. Similar to Java's Swing API, the JavaFX API has stock layouts that provide the most common ways to display UI controls onto the scene graph. Following are the JavaFX layouts discussed in this section:

- `javafx.scene.layout.HBox`
- `javafx.scene.layout.VBox`
- `javafx.scene.layout.FlowPane`
- `javafx.scene.layout.BorderPane`
- `javafx.scene.layout.GridPane`

HBox

The HBox layout's job is to place JavaFX child nodes in a horizontal row. As child nodes are added consecutively, each is appended to the end (right side). By default, the HBox layout honors the child's preferred width and height. When the parent node is not resizable (a Group node, for example), the row height of the HBox takes on the height of the child node with the greatest preferred height. Also, by default each child node aligns to the top-left (Pos.TOP_LEFT) position.

The HBox can be configured to have resizing ranges to allow the parent to manage and calculate space available upon layout requests. There are many ways you can programmatically alter the HBox's layout constraints, such as border, padding, margin, spacing, and alignment. Although in this chapter you will learn how to set constraints using a programmatic approach, there are also alternative strategies. One well known and preferred strategy is to style or set constraints using JavaFX CSS attributes; you will catch a glimpse of that approach later in this chapter, and more details in Chapter 6, on custom UIs.

When dealing with nonresizable child nodes such as Shape nodes, the parent takes into account the Shape's rectangular bounds (ParentInBounds), its width and height. In contrast, when using resizable nodes such as a TextField control, the parent has the ability to manage and compute the available space for the TextField to grow horizontally. To grow UI controls horizontally within an HBox, use the static HBox.setHgrow() method. The following code snippet sets a TextField control to grow horizontally when the parent HBox's width is resized:

```
TextField myTextField = new TextField()  
HBox.setHgrow(myTextField, Priority.ALWAYS);
```

As an example of an HBox layout in action, the code shown in Listing 4-1 uses four rectangles as child nodes with varying HBox constraints. Nonresizable nodes such as rectangle shapes are used in the example to demonstrate the many spacing attributes for the HBox layout control. For brevity, only the relevant code is listed here. To see the full listing, please visit the Source Code/Download tab of the book's page at www.apress.com to download the source code.

Listing 4-1. The HBoxExample.java file demonstrates an HBox layout with shapes as child nodes

```
HBox hbox = new HBox(5);           // space between child nodes only  
hbox.setPadding(new Insets(1));    // padding between hbox border and row.  
  
Rectangle r1 = new Rectangle(10, 10); // Square  
Rectangle r2 = new Rectangle(20, 20); // Big Square  
Rectangle r3 = new Rectangle(5, 20); // Vertical rectangle  
Rectangle r4 = new Rectangle(20, 5); // Horizontal rectangle  
  
HBox.setMargin(r1, new Insets(2,2,2,2)); // space between border & child node  
hbox.getChildren().addAll(r1, r2, r3, r4);
```

Figure 4-1 is the output of Listing 4-1.

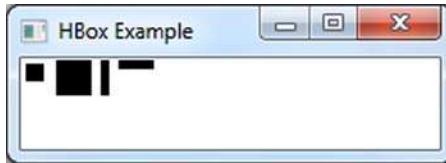


Figure 4-1. The output of the file `HBoxExample.java`, demonstrating the `HBox` layout

In Listing 4-1 the code begins by instantiating an `HBox` using the single-argument constructor. This constructor takes a value of type `double`, which specifies the horizontal space (in pixels) between child nodes. This is also the equivalent of invoking the `setSpacing()` method. Because spacing, padding, margin, and border can often be confusing I've illustrated an `HBox` in Figure 4-2. The figure displays three gray rectangular areas, each five pixels wide, that represent the spacing between child nodes (`r1` to `r4`). Setting the spacing uses an `Insets` object that allows you to create space between the `HBox`'s child nodes. Notice that the inner part of the `HBox` border has yellow padding one pixel wide surrounding the row.

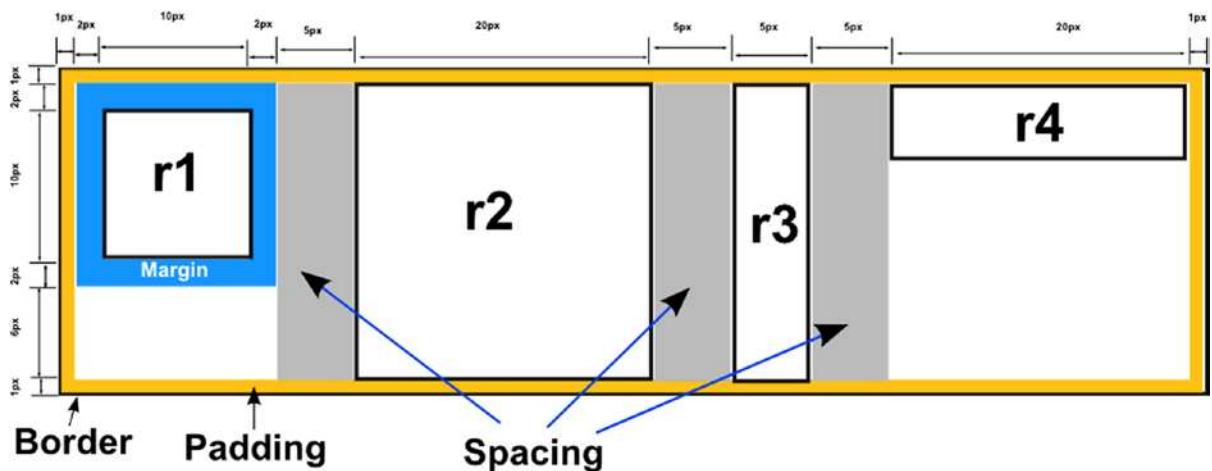


Figure 4-2. An example of an `HBox` layout containing four rectangles

After creating all four rectangles, the code sets the margin on rectangle `r1`. To set the margin you'll notice the static method `HBox.setMargin(Node, Insets)`. The code creates an `Insets` object with a margin of two pixels for the top, right, bottom, and left. As shown in Figure 4-2, this is the surrounding space around rectangle `r1` (blue). To calculate the size of the `HBox`'s dimensions, you basically add up the spacing, margin, padding, and the child nodes' bounds (`ParentInBounds`). When the parent of the `HBox` is a `Group` node (not resizable), the `HBox` width will total 76 pixels and the height 22 pixels.

VBox

The VBox layout is similar to an HBox, except that it places child nodes stacked in a vertical column. As children are added, each is placed beneath the previous child node. By default the VBox honors the children's preferred width and height. When the parent node is not resizable (for example, a Group node), the maximum vertical column's width is based on the node with the greatest preferred width. Also, by default each child node aligns to the top-left (Pos.TOP_LEFT) position. VBox layouts can also be configured to have resizing ranges to allow the parent node to manage and calculate space available upon layout requests. To demonstrate the use of a VBox, the example in Listing 4-2 contains the same four rectangles from the previous example, with the same layout constraints.

Listing 4-2. The VBoxExample.java file, demonstrating VBox layout

```
VBox vbox = new VBox(5);           // spacing between child nodes
vbox.setPadding(new Insets(1));     // padding between border and child nodes

Rectangle r1 = new Rectangle(10, 10); // little square
Rectangle r2 = new Rectangle(20, 20); // big square
Rectangle r3 = new Rectangle(5, 20);  // vertical rectangle
Rectangle r4 = new Rectangle(20, 5);  // horizontal rectangle
VBox.setMargin(r1, new Insets(2,2,2,2)); // margin around r1
vbox.getChildren().addAll(r1, r2, r3, r4);
```

Figure 4-3 shows the output of the `VBoxExample.java` application in Listing 4-2, demonstrating the VBox layout with rectangle shapes as child nodes.

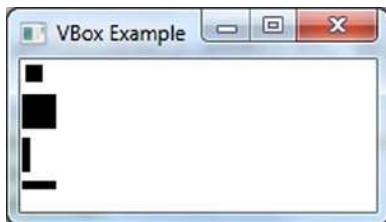


Figure 4-3. The output of `VBoxExample.java`

In Listing 4-2, the code begins by instantiating a VBox with a five-pixel space between child nodes. Figure 4-4 depicts horizontal gray bars that represent the spacing between child nodes (r1 to r4). After constructing the VBox, you set the padding via `setPadding()` method. Setting the padding by using an Insets object allows you to create padding between the border and the row. Notice in Figure 4-4 the VBox padding of one pixel surrounding the VBox shown surrounded by a yellow line.

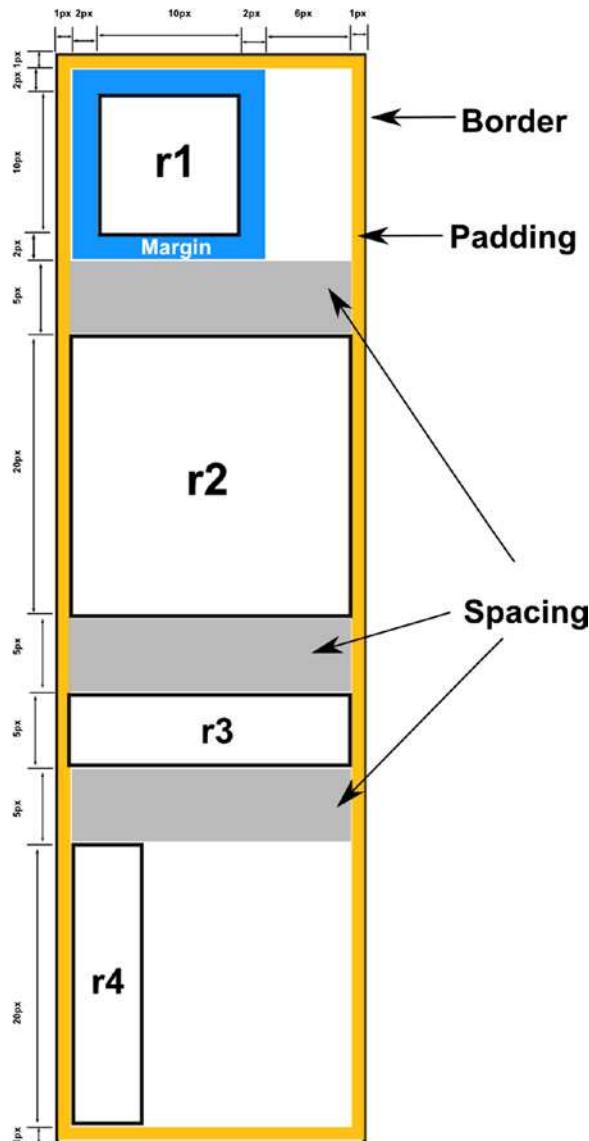


Figure 4-4. `VBox` containing four rectangle child nodes

Once the four rectangles are created, the code sets the margin on the rectangle `r1`. To set the margin we use the static method `VBox.setMargin(Node, Insets)`. The code then creates an `Insets` object with a margin of two pixels for the top, right, bottom, and left. Surrounding rectangle `r1` is a two-pixel margin shown in blue. To find the total size of a `VBox` instance, you simply add up the spacing, margin, padding, and child nodes' dimensions (bounds). Having a `Group` as a parent, the `VBox` node's calculated width would total 22 pixels, and the height would total 76 pixels.

FlowPane

The FlowPane layout node allows child nodes in a row to flow based on the available horizontal spacing and wraps nodes to the next line when horizontal space is less than the total of all the nodes' widths. Depicted in Figure 4-5 are four child nodes flowing from left to right and then wrapping.

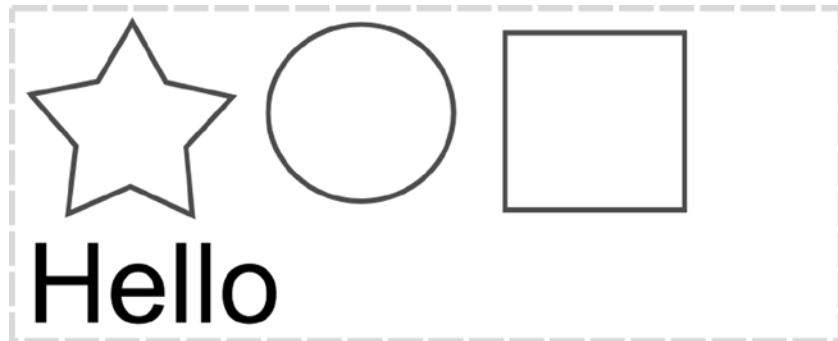


Figure 4-5. A FlowPane layout with nodes flowing left to right with the ability to wrap nodes based on horizontal space. The Hello text node is wrapped to the next line because the available horizontal space, denoted by the dashed line, is too small

By default a FlowPane layout flows child nodes from left to right (`Pos.TOP_LEFT`). To change the flow alignment, you can simply invoke the `setAlignment()` method by passing in an enumerated value of type `Pos`. Listing 4-3 creates a FlowPane layout to flow child nodes from right to left (`Pos.TOP_RIGHT`).

Listing 4-3. The creation of a FlowPane layout to flow from right to left

```
FlowPane flowPane = new FlowPane();
flowPane.setAlignment(Pos.TOP_RIGHT);
flowPane.getChildren().addAll(...); // child nodes to add.
```

BorderPane

The BorderPane layout node allows child nodes to be placed in a top, bottom, left, right, or center region. Because each region can only have one node, developers typically nest layouts. An example would be to create an HBox with children to then be set as the top region via the `setTop()` method. Figure 4-6 shows the BorderPane's regions.

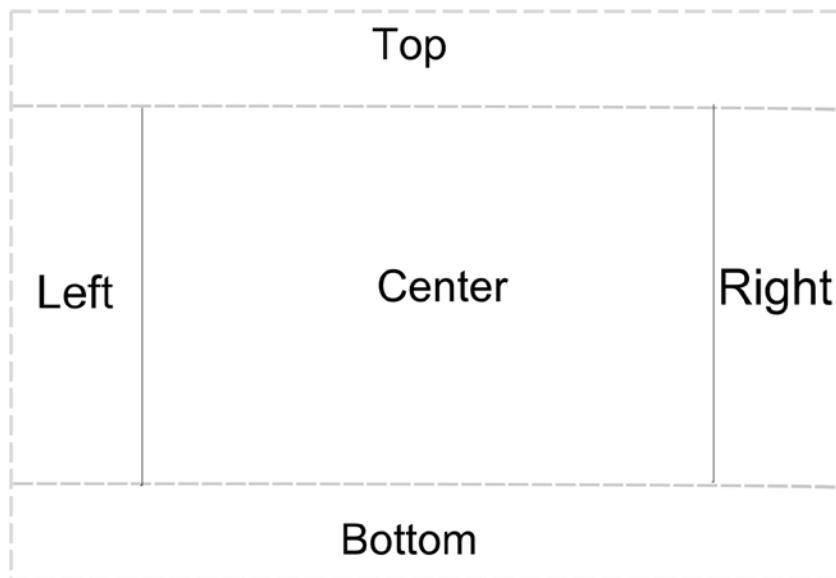


Figure 4-6. The `BorderPane` layout

This layout is similar to many websites we see every day, where navigational links are placed in either the top, bottom, left, or right regions of a page, while the main content is in the center area. A `BorderPane`'s region of top and bottom borders allows a resizable node to take up all of the available width. The left and right border regions take up the available vertical space between the top and bottom borders. All of the bordering regions honor the children's sizes (preferred width and height) by default. According to the Javadocs, the default alignment of nodes when placed in the top, bottom, left, right, and center regions is as follows:

- Top: `Pos.TOP_LEFT`
- Bottom: `Pos.BOTTOM_LEFT`
- Left: `Pos.TOP_LEFT`
- Right: `Pos.TOP_RIGHT`
- Center: `Pos.CENTER`

GridPane

So far you have learned about simple layouts; now let's take a look at a more advanced layout: the `GridPane`, commonly used in business applications. For instance, business applications typically have data entry form screens. Forms usually have read-only labels on the first column and input fields on the second column, resembling a grid pattern.

When using the `GridPane` you can specify constraints at the row, column, or cell level. For example, if the second column contains input text fields, you may want them to resize as the window is being resized. To lay out components in a grid-like manner, the example in Listing 4-4 uses JavaFX's `GridPane` layout. The most interesting part of the example is the use of column constraints on the `GridPane`. Column constraints apply a minimum and maximum horizontal width for the child nodes to grow or shrink as the window is resized.

Listing 4-4. The `GridPaneForm.java` file demonstrates the `GridPane` layout

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("GridPaneForm ");
    BorderPane root = new BorderPane();
    Scene scene = new Scene(root, 380, 150, Color.WHITE);

    GridPane gridpane = new GridPane();
    gridpane.setPadding(new Insets(5));
    gridpane.setHgap(5);
    gridpane.setVgap(5);
    ColumnConstraints column1 = new ColumnConstraints(100);
    ColumnConstraints column2 = new ColumnConstraints(50, 150, 300);
    column2.setHgrow(Priority.ALWAYS);
    gridpane.getColumnConstraints().addAll(column1, column2);

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

    Button saveButton = new Button("Save");

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

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

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

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

    // Save button
    GridPane.setAlignment(saveButton, HPos.RIGHT);
    gridpane.add(saveButton, 1, 2);
```

```
FlowPane topBanner = new FlowPane();
topBanner.setPrefHeight(40);
String backgroundStyle =
    "-fx-background-color: lightblue;" +
    "-fx-background-radius: 30%;" +
    "-fx-background-inset: 5px;";
topBanner.setStyle(backgroundStyle);
SVGPath svgIcon = new SVGPath();
// icon from http://raphaeljs.com/icons/#people
svgIcon.setContent(/* svg path code as a string */);
svgIcon.setStroke(Color.LIGHTGRAY);
svgIcon.setFill(Color.WHITE);

Text contactText = new Text("Contacts");
contactText.setFill(Color.WHITE);

Font serif = Font.font("Dialog", 30);
contactText.setFont(serif);
topBanner.getChildren().addAll(svgIcon, contactText);

root.setTop(topBanner);
root.setCenter(gridpane);

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

After executing Listing 4-4 you will see the output in Figure 4-7, which displays a simple form application allowing the user to enter a first and last name as contact information. When resizing the window, notice that the input text field controls grow or shrink based on the column constraints. Also notice that the Save button is horizontally aligned to the right (`HPos.RIGHT`), just beneath the Last Name text field.



Figure 4-7. Output of the file `GridPaneForm.java`

Listing 4-4 begins by creating an instance of a BorderPane object as the root node for our scene. By using a BorderPane as the root node and then placing the GridPane node object into the center region, we enable the parent (BorderPane) to resize the GridPane by giving it all the available horizontal and vertical space. After creating a BorderPane instance as our root node, the code instantiates a GridPane layout with padding along with horizontal and vertical gaps. The code in Listing 4-5 then creates a GridPane layout with padding, horizontal, and vertical gaps set to 5 (pixels).

Listing 4-5. Setting padding, horizontal, and vertical gap spacing between UI controls for each cell

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

Back in Listing 4-4, the code implements column constraints for the first and second columns. To set a column's constraint, use the `ColumnConstraints` constructor, which allows the developer to specify minimum, preferred, and maximum widths for a column on the `GridPane` node. This is very nice; it means that when the parent container node (BorderPane) is being resized, the `GridPane` will cause its child nodes to grow or shrink horizontally based on its row or column constraints. Listing 4-6 demonstrates the ability to constrain the first column to 100 pixels wide for the read-only labels and the second column with a minimum width of 50, a preferred width of 150 and a maximum width of 300.

Listing 4-6. Setting up column constraints for a JavaFX application

```
ColumnConstraints column1 = new ColumnConstraints(100);           // fixed for labels
ColumnConstraints column2 = new ColumnConstraints(50, 150, 300); // min,pref,max
column2.setHgrow(Priority.ALWAYS);
gridpane.getColumnConstraints().addAll(column1, column2);
```

After creating the column constraint for column 2, the code invokes the `setHgrow()` method by passing in the enum value `Priority.ALWAYS`, which lets the UI controls take up the available horizontal space within cells when the window is widened. I set the maximum width to 300 for the second column so you can see that the child `TextField` does not grow beyond the maximum width (300). In other words, when the window is resized wider than 300 pixels, the text fields within the grid pane column stop stretching.

The next step is simply putting each UI control into its respective cell location. All cells are relative to zero; to place a node in the cell in the first column and first row you would specify the cell as (0, 0). Thus, the following code snippet adds the Save button into the second column, third row of the `GridPane` layout at cell (1, 2):

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

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

```
GridPane.setAlignment(saveButton, HPos.RIGHT);
```

After completing the `GridPane` setup, the code implements the top banner area of the input form shown in Figure 4-7. To create a title banner, we use a flow layout with a background color of light blue, an SVG node icon (people) and a text node (contacts). The important code that relates to the `FlowPane` layout displaying the banner contains the settings for the preferred height and JavaFX CSS styles. Because the banner (`FlowPane`) is placed in the top region of the `BorderPane` layout, the preferred height of 40 pixels will be honored and the width will take up the available horizontal space.

With this knowledge that the `FlowPane` takes up the available horizontal space, notice the area being filled with the light blue background color. By using JavaFX CSS styling, the code sets a background color of light blue along with corner radii and insets specified using a string. Listing 4-7 is the code section that styles the banner (`FlowPane`) to be placed in the top region of the `BorderPane` layout. Don't worry if JavaFX CSS styling doesn't make sense yet; a more detailed discussion of it will follow in Chapter 6.

Listing 4-7. Styling a FlowPane layout for the top banner

```
FlowPane topBanner = new FlowPane();
topBanner.setPrefHeight(40);
String backgroundStyle = "-fx-background-color: lightblue;" +
    "-fx-background-radius: 30%;"
    "-fx-background-inset: 5px;";
topBanner.setStyle(backgroundStyle);
```

We've only scratched the surface of the `GridPane` layout. There are other ways to provide constraints and alignment of child nodes, so I trust you will delve deeper into the `GridPane` layout. Please see the Javadocs documentation for more details. Now that you've learned how to use the most common layouts, you will be able to create well-behaved user interfaces. Next, you will explore common JavaFX UI controls.

UI Controls

UI controls are essentially advanced JavaFX nodes that typically allow user input such as text fields, sliders, list views, and table views. Unlike HTML5, JavaFX allows UI control nodes and shape nodes to coexist on the scene graph. This means you can treat any UI control like any other JavaFX node; for example you can scale, rotate, style, and add effects. The remainder of the chapter presents many common JavaFX UI controls that are commonly used in data entry form applications. For starters, let's talk about application menus. Later sections will show how to work with table views and list views.

Menus

Menus are a standard way for windowed desktop applications to allow users to select options. For example, applications typically have a File menu offering options (menu items) to save or open a file. In a windowed environment the user will use their mouse to navigate and select menu items. Menus and menu items typically also have the functionality of key combinations to select options, also known as keyboard shortcuts. In other words, key combinations allow quick menu selections without the need of a mouse.

Creating Menus and Menu Items

Before exploring how to invoke code triggered by selecting menu options, let's look at how to create menus. For starters, you must create a menu bar (`javafx.scene.control.MenuBar`) object to hold many `javafx.scene.control.Menu` objects. Each `Menu` object is similar to a tree hierarchical structure where `Menu` objects can contain `Menu` and `javafx.scene.control.MenuItem` objects. Thus, a menu may contain other menus as submenus or menus within menus. `MenuItem`s are child options within a `Menu` object. You can think of `MenuItem`s as leaf nodes (containing no children) in a tree structure. Listing 4-8 shows the creation of a menu bar with a File menu that has a Save option as a menu item.

Listing 4-8. The creation of `MenuBar`, `Menu`, and `MenuItem` instances

```
MenuBar menuBar = new MenuBar();
Menu fileMenu = new Menu("File");
fileMenu.getItems().add(new MenuItem("Save"));
menuBar.getMenus().add(fileMenu);
```

Figure 4-8 shows the output of Listing 4-8, a simple File menu containing a Save menu item.

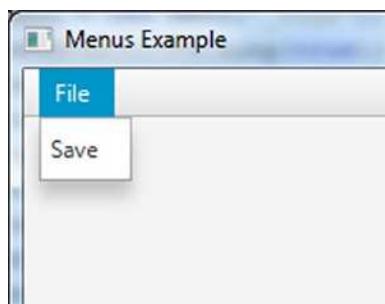


Figure 4-8. A File menu containing a Save menu item

Although you can create simple menu items, you may want more advanced options such as checked options or radio buttons. Based on the inheritance hierarchy, the following are subclasses of the `MenuItem` class. The following listing shows the available `MenuItem` subclasses to use as menu options. A brief description of each subclass will follow.

- `javafx.scene.control.CheckMenuItem`
- `javafx.scene.control.RadioMenuItem`
- `javafx.scene.control.CustomMenuItem`
- `javafx.scene.control.SeparatorMenuItem`
- `javafx.scene.control.Menu`

A `CheckMenuItem` menu item is similar to a check box UI control, allowing the user to select items optionally. The `RadioMenuItem` menu item is similar to the radio button UI control, allowing the user to select only one item from an item group. When you want to create a custom menu item you can use the `CustomMenuItem` class. For instance, you may want to have a menu option that behaves like a toggle button. Next is a `SeparatorMenuItem`, which is really a derived class of type `CustomMenuItem`.

A `SeparatorMenuItem` is a menu item that displays as a visual line to separate menu items. Last in the list is the `Menu` class. Because a `Menu` class is a subclass of `MenuItem`, it has a `getItems().add()` method that's capable of adding children such as other `Menu` and `MenuItem` instances.

Invoking a Selected MenuItem

Now, that you know how to construct a menus and menu items, let's see how to invoke code that is attached to a menu item. You'll be happy to know that you wire-up handler code to a menu item in exactly the same way that you would wire-up JavaFX buttons, which means that menu items also have a `setOnAction()` method. The `setOnAction()` method receives a functional interface of type `EventHandler<ActionEvent>`, which is the handler code that is invoked when the menu item is selected. Listing 4-9 shows two equivalent implementations of action code to be invoked when the Exit menu item is triggered. The first implementation uses an anonymous inner class, and the second uses Java 8's lambda expressions.

Listing 4-9. Adding handler code via the setOnAction() method

```
// Implementation that uses an anonymous inner class
exitMenuItem.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent t) {
        Platform.exit();
    }
});

// Implementation that uses lambda expressions
exitMenuItem.setOnAction(ae -> Platform.exit());
```

Example: Working with Menus

To demonstrate various menu selection scenarios, the next example mimics a security alarm application. Listing 4-10 is from the file `MenuExample.java` and shows the relevant code that demonstrates menu selections. Before executing the code you may want to know how to use this fictitious security alarm application. As illustrated in Figure 4-9, the menu bar displays three main menus: File, Cameras, and Alarm. When the File menu is selected, it displays three options—New, Save, and Exit—as child menu items. Selecting the Cameras menu displays two `CheckMenuItem` menu items, for the optional selection of Show Camera 1 or Show Camera 2. Lastly, the Alarm menu displays two `RadioMenuItem` options and a submenu Contingent Plans. The submenu displays three child `CheckMenuItem` options.

Listing 4-10. The MenuExample.java file is a fictional alarm system application

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Menus Example");
    BorderPane root = new BorderPane();
    Scene scene = new Scene(root, 300, 250, Color.WHITE);

    MenuBar menuBar = new MenuBar();
    root.setTop(menuBar);

    // File menu - new, save, exit
    Menu fileMenu = new Menu("File");
    MenuItem newItem = new MenuItem("New");
    MenuItem saveMenuItem = new MenuItem("Save");
    MenuItem exitMenuItem = new MenuItem("Exit");
    exitMenuItem.setOnAction(actionEvent -> Platform.exit() );

    fileMenu.getItems().addAll(newItem,
        saveMenuItem,
        new SeparatorMenuItem(),
        exitMenuItem
    );
    menuBar.getMenus().add(fileMenu);

    // Cameras menu - camera 1, camera 2
    Menu cameraMenu = new Menu("Cameras");
    CheckMenuItem cam1MenuItem = new CheckMenuItem("Show Camera 1");
    cam1MenuItem.setSelected(true);
    cameraMenu.getItems().add(cam1MenuItem);

    CheckMenuItem cam2MenuItem = new CheckMenuItem("Show Camera 2");
    cam2MenuItem.setSelected(true);
    cameraMenu.getItems().add(cam2MenuItem);

    // Alarm menu
    Menu alarmMenu = new Menu("Alarm");

    // sound or turn alarm off
    ToggleGroup tGroup = new ToggleGroup();
    RadioMenuItem soundAlarmItem = new RadioMenuItem("Sound Alarm");
    soundAlarmItem.setToggleGroup(tGroup);

    RadioMenuItem stopAlarmItem = new RadioMenuItem("Alarm Off");
    stopAlarmItem.setToggleGroup(tGroup);
    stopAlarmItem.setSelected(true);

    alarmMenu.getItems().addAll(soundAlarmItem,
        stopAlarmItem,
        new SeparatorMenuItem());

    Menu contingencyPlans = new Menu("Contingent Plans");
    contingencyPlans.getItems().addAll(
        new CheckMenuItem("Self Destruct in T minus 50"),
        new CheckMenuItem("Turn off the coffee machine "),
        www.it-ebooks.info
```

```

        new CheckMenuItem("Run for your lives! "));

    alarmMenu.getItems().add(contingencyPlans);

    menuBar.getMenus().addAll(fileMenu, cameraMenu, alarmMenu);

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

```

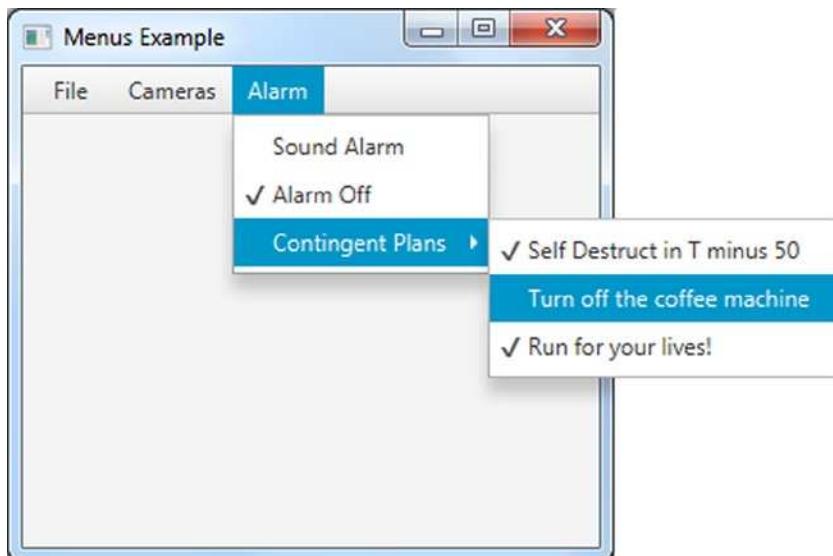


Figure 4-9. The output of Listing 4-10, which simulates an alarm system application

The code begins by creating a `MenuBar` control, which may contain one to many menu (`MenuItem`) objects. After the `MenuBar` is created, it is set as the top region on the `BorderPane` layout. Since the top region allows a resizable node to take up the available horizontal space, the `MenuBar` control will stretch to the width of the `BorderPane` (the root scene). Listing 4-11 sets the top region of the `BorderPane` to a menu bar.

Listing 4-11. A menu bar is set as the top region node of the root `BorderPane`

```

MenuBar menuBar = new MenuBar();
root.setTop(menuBar);

```

Next, the code instantiates `Menu` objects that contain one or more menu item (`MenuItem`) objects and other `Menu` objects, making submenus.

After creating the `File`, menu the code creates the `Cameras` menu. This menu will contain `CheckMenuItem` objects. These menu items allow the user to optionally select or unselect `Show Camera 1` and `Show Camera 2` as menu items.

Last is the implementation of our Alarm menu, which contains two radio menu items (RadioMenuItem), a separator (SeparatorMenuItem), and a submenu (Menu). To create the radio menu items for the Alarm menu, the code creates an instance of a ToggleGroup class. The ToggleGroup class is also used on regular radio buttons (RadioButtons) to allow one selected option only. Listing 4-12 creates radio menu items (RadioMenuItem) to be added to the alarmMenu Menu object.

Listing 4-12. The creation of menu items that contain radio and separator menu items

```
// Alarm menu
Menu alarmMenu = new Menu("Alarm");

// sound or turn alarm off
ToggleGroup tGroup = new ToggleGroup();
RadioMenuItem soundAlarmItem = new RadioMenuItem("Sound Alarm");
soundAlarmItem.setToggleGroup(tGroup);

RadioMenuItem stopAlarmItem = new RadioMenuItem("Alarm Off");
stopAlarmItem.setToggleGroup(tGroup);
stopAlarmItem.setSelected(true);

alarmMenu.getItems().addAll(soundAlarmItem,
    stopAlarmItem,
    new SeparatorMenuItem());
```

The code also adds a visual separator, by instantiating a SeparatorMenuItem class that is added to a menu via the `getItems().addAll()` method. The method `getItems()` returns an observable list of MenuItem objects (`ObservableList<MenuItem>`). You'll learn about ObservableList collections later, but briefly, they are collections that have the ability to notify and update UI controls as items are added or removed. For a full description please see the section "The ObservableList Collection Class" later in this chapter.

The last child menu item added to the Alarm menu is a submenu, Contingent Plans. To add some levity I've created some humorous menu options for contingency plans when an emergency occurs.

Additional Ways to Select Menus and Menu Items

In the previous example you learned how to create and invoke menu items using your mouse, but did you know that there are other ways to invoke menu items? In this section you will learn three additional ways to invoke menu items: by constructing key mnemonics, key combinations, and context menus.

Key Mnemonics

Standard menus generally have keyboard mnemonics to select menu items without using a mouse. Typically, applications that use a menu allow the user to hit the Alt key, which puts an underscore (_) beneath a letter of the menu text label. After the user hits the letter, the menu will drop down to display its child menu items and the user will be allowed to navigate with arrow keys.

To perform this behavior in code, you instantiate a Menu by invoking the constructor that receives a String value as before, but this time you place an underscore character preceding the chosen letter in the menu or menu item's text. Also, to let the system recognize the mnemonic you will simply pass true to the `setMnemonicParsing(true)` method. Listing 4-13 creates a File menu that uses the letter "F" as the mnemonic.

Listing 4-13. The creation of a File menu with a mnemonic for the letter “F”

```
Menu fileMenu = new Menu("_File");
fileMenu.setMnemonicParsing(true);
```

Key Combinations

A key combination is a combination of keystrokes to select a menu option. For instance, most applications on the Windows platform have the key combination of Ctrl + S to save a file. On the Mac OS platform, the key combination would be Command + S. Keys such as Ctrl, Command, Alt, Shift, and Meta are called modifier keys. Usually the modifiers are pressed in combination with a single letter. Sometimes key combinations are referred as keyboard shortcuts. To create a key combination, you need an instance of the KeyCodeCombination object that will contain the keystroke and the modifiers. Listing 4-14 provides a key code combination of (Ctrl or Meta) + S. You will notice that the code uses the KeyCombination.SHORTCUT_DOWN value as the key modifier instead of CONTROL_DOWN or META_DOWN. The reason is simple; the value of SHORTCUT_DOWN will enable the application to be cross-platform. The values CONTROL_DOWN and META_DOWN are system dependent on the Windows and MacOS platforms respectively, but SHORTCUT_DOWN works on all platforms.

Listing 4-14. A menu item mapped to a keyboard shortcut of Ctrl + S (Windows) or Command + S (Mac)

```
MenuItem saveItem = new MenuItem("_Save");
saveItem.setMnemonicParsing(true);
saveItem.setAccelerator(new KeyCodeCombination(KeyCode.S, KeyCombination.SHORTCUT_DOWN));
```

Context Menus

Context menus are popup menus displayed when a user right-clicks the mouse button on a JavaFX UI control or a stage surface. To create a context menu, you instantiate a ContextMenu class. Exactly like a regular menu, the ContextMenu menu has a getItems().add() method to add menu items. The following code snippet shows a context menu instantiated with a menu item (exitItem):

```
ContextMenu contextFileMenu = new ContextMenu(exitItem);
```

To respond to a right-click of the mouse on the scene in an application, you can basically add an event handler to listen for a right-click event. Once that is detected, the context menu’s show() method is invoked. The code in Listing 4-15 sets up an event handler to show and hide a context menu based on a right or left mouse click, respectively. Notice the hide() method invoked by the primary mouse click (left-click) to remove the context menu.

Listing 4-15. Attaching an event handler that displays a popup menu upon encountering a right-click mouse event on a JavaFX stage

```
primaryStage.addEventHandler(MouseEvent.MOUSE_CLICKED, (MouseEvent me) -> {
    if (me.getButton() == MouseButton.SECONDARY || me.isControlDown()) {
        contextFileMenu.show(root, me.getScreenX(), me.getScreenY());
    } else {
        contextFileMenu.hide();
    }
});
```

To see the full code listing showing the use of these additional strategies to invoke menus, please visit the book’s website to download the project KeyCombinationsAndContextMenu.

The ObservableList Collection Class

When using the Java Collections API you'll notice the many useful container classes that represent data structures such as sets, maps, and lists. One of the commonly used concrete list classes is `java.util.ArrayList`, which implements the `List` interface. For a long time Java Swing developers built applications using an `ArrayList` to represent a list of objects that will be displayed in a list-like UI control. The issues that caused a lot of grief were the ability to synchronize list models and view components. To solve this problem, you can use JavaFX's `ObservableList` class. The `ObservableList` is a collection that is capable of notifying UI controls when objects are added, updated, and removed. JavaFX `ObservableLists` are typically used in list UI controls such as `ListView` and `TableView`. Let's look at two examples that will use the `ObservableList` collection class.

Working with ListViews

The Hero Picker example application was created to demonstrate the `ObservableList`'s capabilities; it displays two JavaFX `ListView` UI controls and allows the user to transfer items between the two `ListView` controls. One list contains candidates for hero status and the other contains those who have already achieved it. As shown in Listing 4-16, this application allows you to choose a candidate to be transferred to the other `ListView` control, which contains the chosen heroes.

Listing 4-16. The Hero Picker application, which demonstrates the ObservableList and ListView classes

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Hero Picker: Chapter 4 Creating and Working with ObservableLists");
    BorderPane root = new BorderPane();
    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);
    ColumnConstraints column1 = new ColumnConstraints(150, 150, Double.MAX_VALUE);
    ColumnConstraints column2 = new ColumnConstraints(50);
    ColumnConstraints column3 = new ColumnConstraints(150, 150, Double.MAX_VALUE);
    column1.setHgrow(Priority.ALWAYS);
    column3.setHgrow(Priority.ALWAYS);
    gridpane.getColumnConstraints().addAll(column1, column2, column3);

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

```
// Heroes label
Label heroesLbl = new Label("Heroes");
gridpane.add(heroesLbl, 2, 0);
GridPane.setAlignment(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<>(candidates);
gridpane.add(candidatesListView, 0, 1);

// heroes
final ObservableList<String> heroes = FXCollections.observableArrayList();
final ListView<String> heroListView = new ListView<>(heroes);

gridpane.add(heroListView, 2, 1);

// select heroes
Button sendRightButton = new Button(" > ");
sendRightButton.setOnAction((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((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);
root.setCenter(gridpane);

GridPane.setVgrow(root, Priority.ALWAYS);
primaryStage.setScene(scene);
primaryStage.show();
}

```

Figure 4-10 shows the output from Listing 4-16, the Hero Picker application.

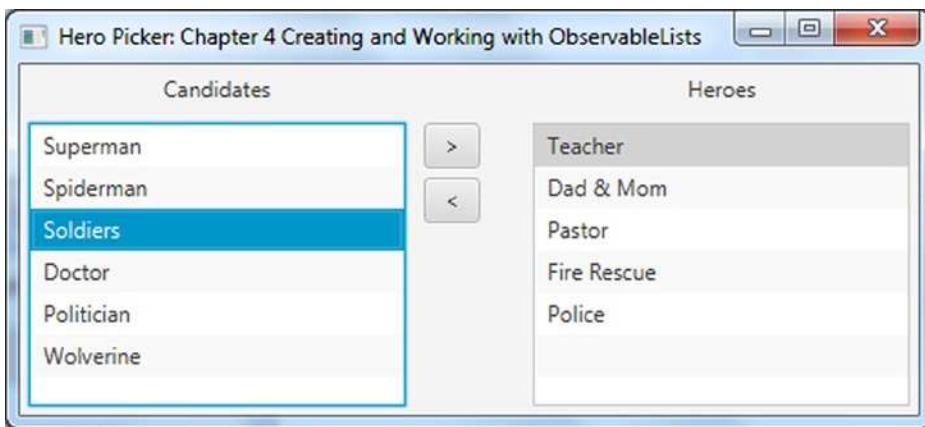


Figure 4-10. The output of the Hero Picker application, demonstrating the use of the `ObservableList` and `ListView` classes

Created in this example is a GUI application that allows a user 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 the code example uses two `ListView` controls to hold `String` objects. For the sake of brevity we will not discuss the use of the layouts, but jump right into the code relating to the button actions and `ListView` controls.

Before creating an instance of a `ListView`, the code creates an `ObservableList` containing our candidates. Here you'll notice the use of a factory class called `FXCollections`, into which you can pass common collection types to be wrapped and returned to the caller as an `ObservableList`. In the example an array of `Strings` is passed into the `FXCollections.observableArrayList()` method instead of an `ArrayList`. Using the `ObservableList` allows us to update the list of items in the array and in the display on the screen concurrently with a single method call.

Hopefully you get the idea about how to use the `FXCollections` class. I trust you will use it wisely; as Peter Parker's Uncle Ben says, "With great power, there must also come great responsibility." The following code line calls the `FXCollections.observableArrayList()` method class to return an observable list (`ObservableList`):

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

After the code creates an ObservableList, a ListView class is instantiated using a constructor that receives the observable list. The following code creates and populates a ListView object:

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

The last item of business is where the code manipulates ObservableLists as if they were java.util.ArrayLists. As they are manipulated, the ListView is notified and automatically updated to reflect the changes to the ObservableList. Listing 4-17 implements the action event code when the user presses the Send (>) button.

Listing 4-17. Action code to process selected items

```
// select heroes
Button sendRightButton = new Button(">");
sendRightButton.setOnAction( actionEvent -> {
    String potential = candidatesListView.getSelectionModel().getSelectedItem();
    if (potential != null) {
        candidatesListView.getSelectionModel().clearSelection();
        candidates.remove(potential);
        heroes.add(potential);
    }
});
```

To set action code for the button, the code uses a lambda expression with an actionEvent (ActionEvent) as a parameter, which is invoked when the button is clicked. When a button-press event arrives, the code determines which item in the ListView was selected via the getSelectionModel().getSelectedItem() on the ListView. Once the item is determined, the code clears the selection (clearSelection()), removes the item, and adds the item to the Heroes ObservableList. What's great is that there is no need to refresh the ListView UI control when the collection changes.

Working with TableViews

We can also use ObservableLists in another list-like UI control, the TableView. JavaFX's javafx.scene.control.TableView control is analogous to Swing's JTable component, which contains rows, columns, and cells very similar to spreadsheet applications.

As an example of the TableView control I've created an application called Bosses and Employees that demonstrates how to populate a TableView to contain a collection of employees (the Person JavaFX bean) based on the selection of a boss from a ListView control.

Before delving into the GUI code, we will examine the use of a domain object to represent a person (Person). Listing 4-18 shows a JavaFX bean of a Person class that represents a boss or an employee that will be displayed in a ListView and/or TableView control. The GUI code to handle list view selection and populating the table view appears in Listing 4-19.

Listing 4-18. The Person class as a JavaFX bean to be used in ObservableLists

```
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);  
    }  
}
```

Listing 4-19. The main GUI code that creates a grid pane containing a ListView and a TableView control, for bosses and employees, respectively

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Bosses and Employees: Chapter 4 Working with Tables");
    BorderPane root = new BorderPane();
    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);
    root.setCenter(gridpane);

    // candidates label
    Label candidatesLbl = new Label("Boss");
    GridPane.setAlignment(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.setMaxWidth(Double.MAX_VALUE);
    leaderListView.setPrefHeight(150);

    // display first and last name with tooltip using alias
    leaderListView.setCellFactory(new Callback<ListView<Person>, ListCell<Person>>() {
        @Override
        public ListCell<Person> call(ListView<Person> param) {
            Label leadLbl = new Label();
            Tooltip tooltip = new Tooltip();
            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.setAlignment(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().addAll(aliasNameCol, firstNameCol, lastNameCol);
gridpane.add(employeeTableView, 2, 1);

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

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

```

Listing 4-20 shows the `getPeople()` method contained in the `BossesAndEmployees` main application class to help populate the `TableView` control.

Listing 4-20. A private method `getPeople()` that returns an `ObservableList` containing `Person` instances. A boss may contain zero to many employees

```

private ObservableList<Person> getPeople() {
    ObservableList<Person> people = FXCollections.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");
// ...code to add employees
Person biker = new Person("Mountain Biker", "Jonathan", "Gennick");
// ...code to add employees

people.addAll(docX, magneto, biker);
return people;
}

```

Figure 4-11 displays the output of the Bosses and Employees application, which demonstrates a JavaFX TableView control.

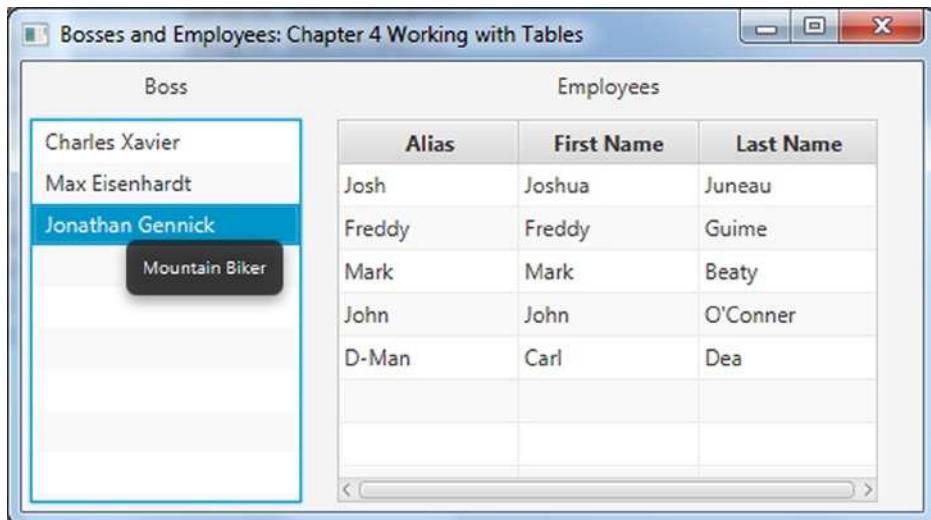


Figure 4-11. The output of the Bosses and Employees application

For the fun of it, I've created a simple GUI application that displays bosses and their employees. Notice on the left side of Figure 4-11 a list of important people (bosses). When you click to select a boss, their employees will be shown in the TableView control to the right. You'll also notice the tooltip popup when the mouse cursor is hovered over the selected boss.

Before we discuss the TableView control, let's detail the ListView control's responsibility of updating the TableView. In model view fashion, the code in Listing 4-20 first creates an ObservableList containing all the bosses via the private getPeople() method to be passed to the ListView control's constructor. Listing 4-21 populates the ListView control with the list of bosses.

Listing 4-21. Creation of a ListView populated with an ObservableList containing Person objects

```

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

```

Next, the code instantiates a cell factory via the `setCellFactory()` method in Listing 4-19 to properly display the person's name in the `ListView` control. Because each item isn't a string but a `Person` object, the `ListView` does not know how to render each row (cell) in the `ListView` control. To inform the `ListView` which properties to use from the `Person` object, the code simply creates a `javafx.util.Callback` generic object by specifying the `ListView<Person>` and a `ListCell<Person>` data type. With your trusty NetBeans IDE, it can pregenerate or stub code for an empty implementation of the method `call()`.

You just need to type:

```
leaderListView.setCellFactory(new );
```

As the cursor is positioned after the `new` keyword hit `Ctrl + Space` and when you see the `Callback` interface at the top of the popup hit `Enter` or select it using your mouse pointer. The following code block will be generated.

```
leaderListView.setCellFactory(new Callback<ListView<Person>, ListCell<Person>>() {  
    @Override  
    public ListCell<Person> call(ListView<Person> p) {  
        throw new UnsupportedOperationException("Not supported yet.");  
    }  
});
```

After generating the code above using the NetBeans code completion feature the throws `UnsupportedOperationException` statement will need to be replaced with a code implementation to respond when a row item (`ListCell`) is selected. Next, are the details on the implementation of the `call()` method.

Within the `call()` method shown in Listing 4-22 is the variable `cell` of type `ListCell<Person>`, in which the code creates an anonymous inner class. The inner class must implement the `updateItem()` method. The job of the `updateItem()` method is to obtain each person's information, which then updates a `Label` control (`leadLbl`). The label is what is displayed for each cell row in the `ListView` control. The last thing the `updateItem()` method does is to add a tooltip popup (`Tooltip`) to each cell row. The tooltip will pop up when the cursor hovers over the cell (boss) within the `ListView`. After updating the cell the `cell` variable is returned.

Listing 4-22. The `call` method to display the boss's name in a `ListView` control along with a tooltip popup showing the alias name

```
@Override  
public ListCell<Person> call(ListView<Person> param) {  
    Label leadLbl = new Label();  
    Tooltip tooltip = new Tooltip();  
    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;
}
```

Finally, the code creates a TableView control to display the employees based on the selected boss from the ListView control. When creating a TableView control the code first creates the column headers as shown in Listing 4-23.

Listing 4-23. Creating the First Name table column by setting the property with the JavaFX bean property name convention

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

Once you have created a column, you'll notice the `setCellValueFactory()` 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 column in the table. Based on the JavaBean convention a new PropertyValueFactory instance is created with the name of the bean attribute (`firstName`).

Last is the implementation of the selection listener on the ListView. In the code you'll notice the `getSelectionModel().selectedItemProperty()` method (shown at the end of Listing 4-19) that allows a listener to be added. The code simply creates and adds a ChangeListener for selection events. When a user selects a boss, the TableView is cleared and populated with the employees of the selected boss. Actually, it is the magic of the ObservableList that notifies the TableView of changes. Listing 4-24 clears and populates the TableView via the teamMembers (ObservableList) variable.

Listing 4-24. Clearing the ObservableList owned by the TableView and repopulating it with employees

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

Generating a Background Process

Typically in desktop applications that have long running processes, visual feedback should indicate that something is happening or suggest that the user be patient. One of the main pitfalls of GUI development is the difficulty of knowing when and how to delegate work to worker threads. Trying to avoid these pitfalls, GUI developers are constantly reminded of thread safety, especially when it comes to blocking the UI thread during heavy load. JavaFX provides UI controls that help solve this issue by off-loading work to a different thread (Task) while providing feedback or progress indicators of the work that is being done.

To demonstrate a GUI application generating a background process while also providing feedback to the user, I've created an example application that simulates a dialog window visualizing the progress of many files being copied. Listing 4-25 is a code example of using JavaFX's progress bar and indicator UI controls to provide feedback to the user as files are being copied.

The following are the main classes used in this example:

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

Listing 4-25. An example application that demonstrates background processes

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("BackgroundProcesses: Chapter 4 Background Processes");

    // ... Layout and UI controls code here
    final Label label = new Label("Files Transfer:");
    final ProgressBar progressBar = new ProgressBar(0);
    final ProgressIndicator progressIndicator = new ProgressIndicator(0);

    // ... Layout and UI controls code here
    final Button startButton = new Button("Start");
    final Button cancelButton = new Button("Cancel");
    final TextArea textArea = new TextArea();
    // ... Layout and UI controls code here

    // wire up Start button
    startButton.setOnAction((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(
            (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((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();
}

private 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;
        }
    };
}

private 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);
}

```

The code begins by creating not one but two progress controls to show the user the work being done. The first control is a progress bar, and the second is a progress indicator. Figure 4-12 depicts from left to right a progress bar and a progress indicator (pie chart). The progress indicator shows a percentage below the indicator icon.

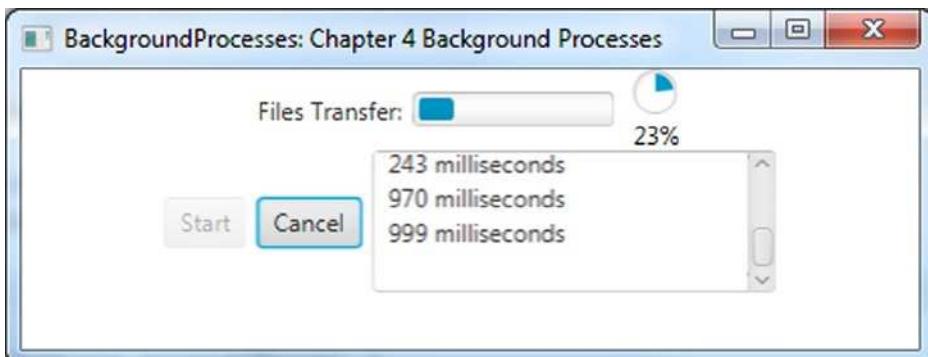


Figure 4-12. The output of Listing 4-25, simulating files being copied

Listing 4-26 is the creation of a progress bar and a progress indicator using the constructor that initializes the progress value to zero.

Listing 4-26. Instantiating the ProgressBar and ProgressIndicator UI controls

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

Next, the code creates a worker thread via the private `createWorker()` method. The `createWorker()` convenience method instantiates and returns 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, thanks to its event-driven design. An example is a `Task` object's ability to update progress asynchronously.

When creating an instance of a `Task` object, the code implements the `call()` method to do work in the background. As the work is being done, you will often want to queue up intermediate results such as progress or text information to provide feedback to the user. To queue information, the API allows you to call the `updateProgress()` or `updateMessage()` method. These methods update information in a thread-safe way so that the observer of the progress properties can update the GUI safely without blocking the GUI thread. Listing 4-27 is code that queues up messages and progress.

Listing 4-27. The text messages and progress to be updated without blocking the progress UI controls

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

After creating a worker `Task`, the code unbinds any old tasks bound to the progress controls. Once the progress controls are unbound, the code then binds the progress controls to a newly created `Task` object named `copyWorker`. Listing 4-28 rebinds a new `Task` object to the progress UI controls.

Listing 4-28. After a task is used, the progress controls will need to rebind to a new task

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

Moving on, notice the code implementing 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.

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

We finish off our code by wiring up the Cancel button. This button simply calls 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 reusable solution, look at the javafx.concurrent.Service class.

Summary

You began this chapter by learning the simple layouts such as the HBox, VBox, FlowPane, and BorderPane. You saw that the HBox layout facilitated a way to string along child nodes horizontally in a row. Similar to the HBox is the VBox layout, which places child nodes vertically in a column. Next, you learned about the FlowPane layout, which is very useful when laying out child nodes to flow horizontally and wrapping nodes to the next line depending on the available horizontal space. The fourth of the simple layouts is the BorderPane layout, which provides regions such as top, bottom, right, left and center. By learning these simple layouts you were able to understand the fundamentals of the many spacing-related attributes. Learning about these attributes helped you differentiate between what spacing, padding, and margins each contribute to. The last layout you were introduced was the advanced GridPane layout, which provides a way to manage child nodes in a tabular grid containing columns and rows, where each cell can contain a JavaFX node. You also ran an example to show off a form-based application using FlowPane, BorderPane, and GridPane. In particular, the example demonstrated column constraints, which enabled child nodes to be resized.

After learning about common layouts you delved into UI controls. You began by learning how to create standard menus and menu bars. By creating menus you were able to learn to create handler code to respond to the selection of menu items. Next, you were able to map key combinations or keyboard shortcuts to menu options. You learned how to create context menus invoked by right-clicking the mouse on the scene graph. With the many ways to create menus and menu items, you got a chance to run an example program that simulated a security alarm application.

Finally, you were exposed to ObservableList collections, which support the display of UI controls such as the ListView and TableView nodes. By learning about the UI controls ListView and TableView, you saw fun examples enabling a user to move items between lists. Lastly, you learned about the ProgressBar and ProgressIndicator for background worker threads. With ProgressBars and ProgressIndicators you were able to run an example that simulates background worker threads copying files.

Chapter 5

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 building GUI applications needing graphics or animations, we are often on a collision course between business and gaming worlds.

In the ever-changing world of rich Internet applications (RIA), you probably have noticed an increased use of animations such as pulsing buttons, transitions, moving backgrounds, and so on. GUI applications can use animations to provide visual cues to let the user know what to do next. With JavaFX, you will be able to have the best of both worlds, creating appealing graphics as well as handling business transactions.

In this chapter you will learn about displaying images and playing animations. Fasten your seatbelts; you'll discover solutions to integrate cool game-like interfaces into everyday applications.

Working with Images

Yet another powerful capability in JavaFX is the ability to display standard image file formats on the scene graph. In this section you will learn how to do the following:

- Load images (`javafx.scene.image.Image`) from your local file system or a remote webserver.
- Display images using the `javafx.scene.image.ImageView` node.

After learning about the `Image` and `ImageView` APIs, you will also see an example of a photo viewer application that demonstrates the ability to load and display images.

Loading Images

You are probably aware of the many standard image file formats, such as .jpg, .png, .gif, and .bmp. To load standard image file formats JavaFX provides the `javafx.scene.image.Image` API. The `Image` class has many convenient constructors that facilitate different loading strategies, as shown in the following list:

- `Image(java.io.InputStream inputStream)`
- `Image(java.io.InputStream is, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth)`
- `Image(java.lang.String url)`
- `Image(java.lang.String url, boolean backgroundLoading)`
- `Image(java.lang.String url, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth)`
- `Image(java.lang.String url, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth, boolean backgroundLoading)`

You can see that there are only a handful of parameters, common to all of the constructors that are used in various combinations for different loading scenarios. Table 5-1 briefly describes each parameter.

Table 5-1. The `javafx.scene.image.Image` constructor's parameters

Parameter	Data Type	Description
<code>inputStream</code>	<code>java.io.InputStream</code>	An input stream such as a file or network.
<code>url</code>	<code>String</code>	An image's URL location such as a file on the local file system or a web server hosting an image file.
<code>backgroundLoading</code>	<code>boolean</code>	Loads the image in the background (off the JavaFX application thread).
<code>requestWidth</code>	<code>double</code>	Specifies an image's bounding box width.
<code>requestedHeight</code>	<code>double</code>	Specifies an image's bounding box height.
<code>preserveRatio</code>	<code>boolean</code>	Used to keep an image's aspect ratio within the bounding box.
<code>smooth</code>	<code>boolean</code>	True indicates the use of a better algorithm to smooth the image, which will be slower; otherwise it will use the lower quality, which will render faster.

To demonstrate loading an image, Listing 5-1 shows code that loads an image first from a local file system and then from a remote host on a web server. The constructor accepts a string representing a URL and the `backgroundLoading` parameter (`boolean`).

Listing 5-1. Two images are loaded based on the protocol file and HTTP for URLs

```

try {
    File file = new File("C:\\Users\\jdoe\\Pictures\\myphoto.jpg");
    String localUrl = file.toURI().toURL().toString();
    Image localImage = new Image(localUrl, false); // don't load in the background

    String remoteUrl = "http://mycompany.com/myphoto.jpg";
    Image remoteImage = new Image(remoteUrl, true); // load in the background

    System.out.println(localUrl); // file:/C:/Users/jdoe/Pictures/myphoto.jpg
    System.out.println(remoteUrl); // http://mycompany.com/myphoto.jpg

    // ... rest of the code

} catch (MalformedURLException ex) {
    // error
}

```

After running Listing 5-1, you'll see the following output:

```

file:/C:/Users/jdoe/Pictures/myphoto.jpg
http://mycompany.com/myphoto.jpg

```

To load an image from the local file system, the code creates a `File` object based on a string with the path location to the image. In this scenario I've specified a Microsoft Windows-based absolute file path for the image, but don't worry about system-dependent code; you will see that the `toURL()` method converts the file path into a standard URL format. After creating the file object, the code obtains a string formatted to the standard URL specification, via the `toURI().toURL().toString()` method call. What's nice is that the method call converts absolute paths into the standard Uniform Resource Locator specification format. After the method call, the returned string will contain the `file:` protocol prefixed to the file path information.

Also implemented in the example is the ability to load an image hosted on a remote web server. You can see that the code also uses `http:` as a protocol prefixed to the URL path to demonstrate loading images on a host web server.

Now that you know how to load images, you will learn how to display them using the JavaFX `ImageView` node.

Viewing Images

A `javafx.scene.image.ImageView` node is a wrapper object that references an `Image` object, which we discussed earlier. Because the `ImageView` object is a JavaFX Node object you will be able to apply effects, perform transforms, and scale images.

When an `ImageView` node is used to apply special effects such as blurring, the image's pixel data is not actually manipulated, but copied, to be calculated and displayed onto the `ImageView` node. This is quite a powerful technique when there are many `ImageView` objects that all point to a single `Image` object. Listing 5-2 loads an image asynchronously (a technique known as background loading) to be passed into an `ImageView` constructor. By default, the width and height will be the actual dimensions

in pixels of the image. In the next section you will see an example of a photo viewer application, which will allow images to be displayed and scaled.

Listing 5-2. The creation of an ImageView node

```
Image image = new Image(url, true);
ImageView imageView = new ImageView(image);
```

A Photo Viewer Example

A practical way to display images in JavaFX is demonstrated by an example photo viewer application. This application will allow you to drag and drop image files to be viewed on the scene graph. Figure 5-1 illustrates the JavaFX photo viewer application displaying an image. In this section I will first describe the photo viewer's features, then summarize how to use it, and finally list and then walk through the source code.

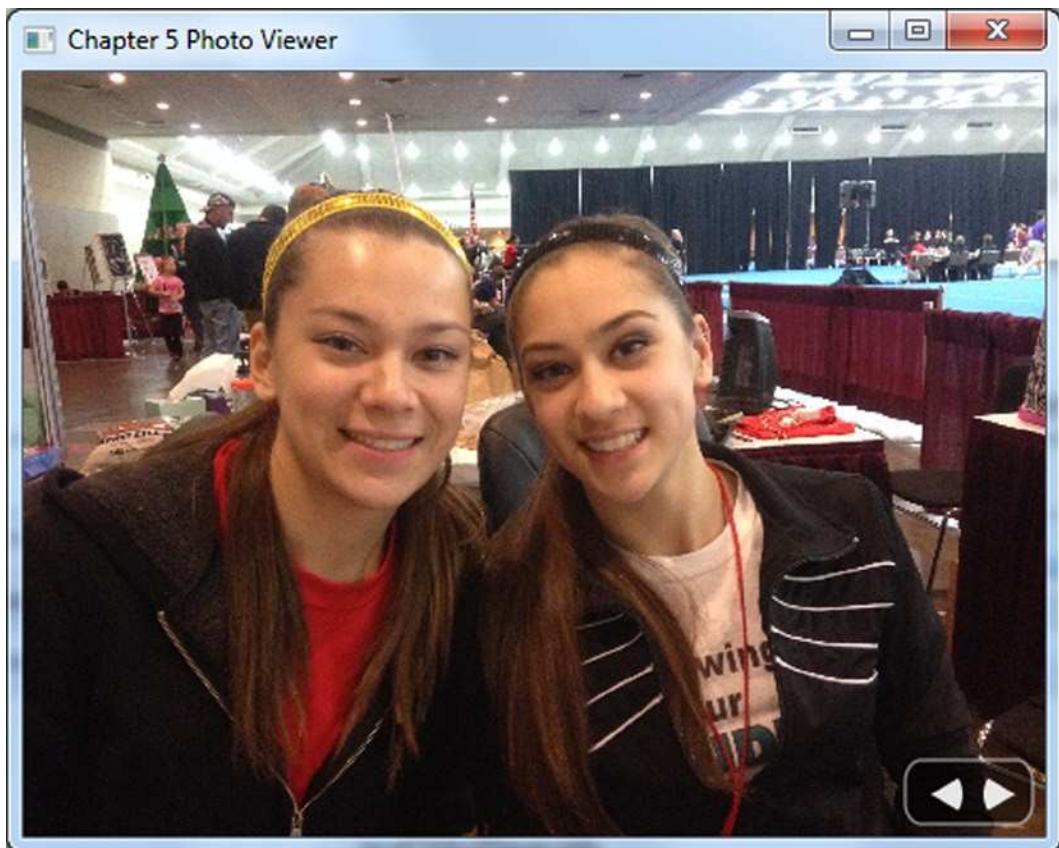


Figure 5-1. A JavaFX Photo Viewer application displaying an image file

Before looking at the source code of our JavaFX photo viewer application, I will briefly explain the features of the photo viewer application and how to use it.

Photo Viewer Features

The photo viewer application lets users do the following:

- Load images with the following file formats: .jpg, .gif, .png, .bmp.
- Drag and drop a local image file to be viewed.
- Drag and drop a remote image file hosted on a web server to be viewed.
- Display a progress indicator showing the loading of an image.
- Resize the currently displayed image by resizing the window while maintaining the aspect ratio.
- Page through images using button controls to see the previous and next image.

Photo Viewer User Instructions

Take the following steps to run the photo viewer application:

1. Locate an image on the file system:

/home/my_username/fun-pics.jpg

or webserver (browser's address line):

<http://www.mycompany.com/images/fun-pics.jpg>

2. Use your mouse to drag and drop the file onto the window depicted in Figure 5-2.

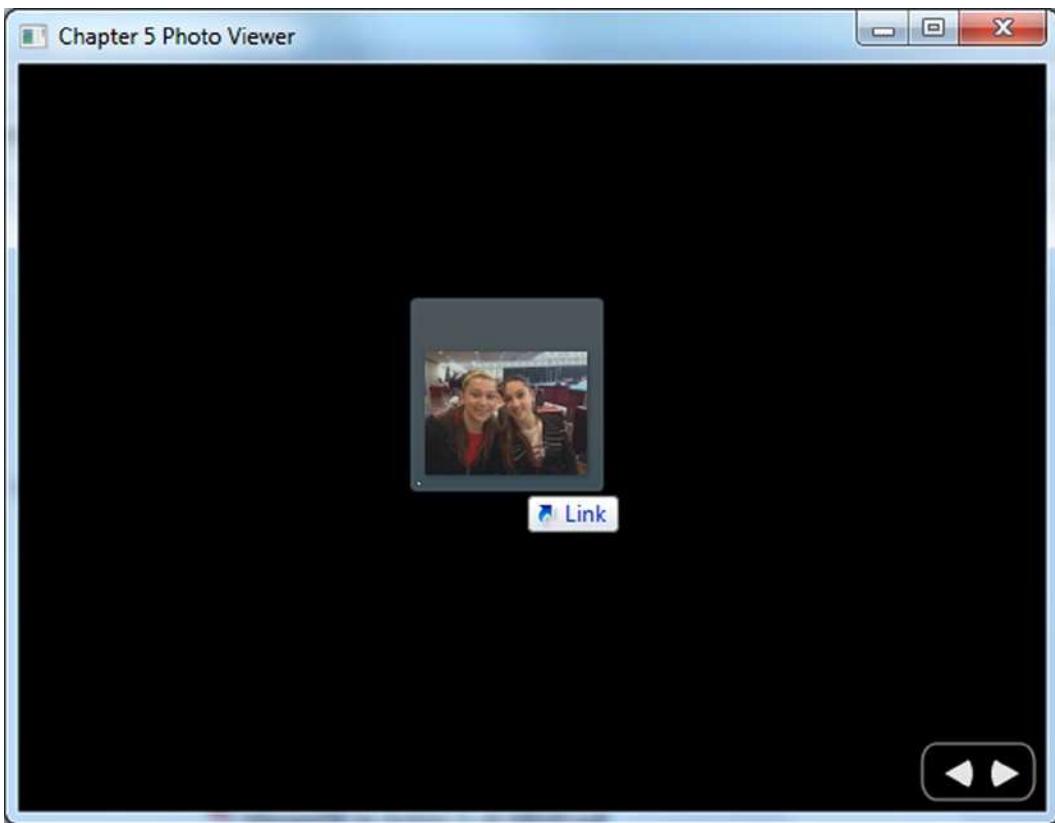


Figure 5-2. Dragging a file into the photo view application

3. Repeat steps 1 and 2 until you have more than one image.
4. Click the left-arrow button to view the previous image.
5. Click the right-arrow button to view the next image.
6. Resize the window to resize the image.

Photo Viewer Code Listing

Now that you know what the photo viewer is required to do and how it should operate, let's look at the source code, shown in Listing 5-3.

Listing 5-3. The code listing for a photo viewer application

```
package jfx8ibe;

// imports omitted...

/**
 * A photo viewer application to demonstrate the JavaFX ImageView node.
 * <p>
 * Instructions:
 *   1. Drag and drop an image file onto the application window.
 *   2. Repeat step 1 so more than 2 images are loaded.
 *   3. Click the left and right arrow controls to advance.
 * </p>
 * @author cdea
 */

public class PhotoViewer extends Application {
    // List of URL strings
    private final List<String> imageFiles = new ArrayList<>();
    // The current index into the imageFile
    private int currentIndex = -1;
    // Enumeration of next and previous button directions
    public enum ButtonMove {NEXT, PREV};
    // Current image view display
    private ImageView currentImageView;
    // Loading progress indicator
    private ProgressIndicator progressIndicator;
    // mutex */
    private AtomicBoolean loading = new AtomicBoolean();

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chapter 5 Photo Viewer");
        Group root = new Group();
        Scene scene = new Scene(root, 551, 400, Color.BLACK);
        scene.getStylesheets()
            .add(getClass()
                .getResource("photo-viewer.css")
                .toExternalForm());

        primaryStage.setScene(scene);

        // set up the current image view area
        currentImageView = createImageView(scene.widthProperty());

        // set up drag & drop file abilities
        setupDragNDrop(scene);

        // create button panel controls (left & right arrows)
        Group buttonGroup = createButtonPanel(scene);
```

```

        // create a progress indicator
        progressIndicator = createProgressIndicator(scene);

        root.getChildren().addAll(currentImageView,
                                buttonGroup,
                                progressIndicator);

        primaryStage.show();
    }
}

/*
 * A factory function returning an ImageView instance to
 * preserve the aspect ratio and bind the instance to the width
 * of the scene for resizing the image.
 *
 * @param widthProperty is the Scene's read only width property.
 * @return ImageView newly created image view for current display.
 */
private ImageView createImageView(ReadOnlyDoubleProperty widthProperty) {
    // maintain aspect ratio
    ImageView imageView = new ImageView();
    // set aspect ratio
    imageView.setPreserveRatio(true);
    // resize based on the scene
    imageView.fitWidthProperty().bind(widthProperty);
    return imageView;
}

/*
 * Sets up the drag and drop capability for files and URLs to be
 * dragged and dropped onto the scene. This will load the image into
 * the current image view area.
 * @param scene The primary application scene.
 */
private void setupDragNDrop(Scene scene) {
    // Dragging over surface
    scene.setOnDragOver((DragEvent event) -> {
        Dragboard db = event.getDragboard();
        if ( db.hasFiles()
            || (db.hasUrl()
                && isValidImageFile(db.getUrl())))
            event.acceptTransferModes(TransferMode.LINK);
        } else {
            event.consume();
        }
    });

    // Dropping over surface
    scene.setOnDragDropped((DragEvent event) -> {
        Dragboard db = event.getDragboard();
        // image from the local file system.
    });
}

```

```

        if (db.hasFiles() && !db.hasUrl()) {
            db.getFiles()
                .stream()
                .forEach( file -> {
                    try {
                        addImage(file.toURI().toURL().toString());
                    } catch (MalformedURLException ex) {
                        ex.printStackTrace();
                    }
                });
        } else {
            // image from some host
            addImage(db.getUrl());
        }
        if (currentIndex > -1) {
            loadImage(imageFiles.get(currentIndex));
        }

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

}

/*
 * Returns a custom created button panel
 * containing left and right buttons to
 * see previous and next images.
 * @param scene The main application scene
 * @return Group A custom button panel with
 * previous and next buttons
 */
private Group createButtonPanel(Scene scene){
    // create button panel
    Group buttonGroup = new Group();

    Rectangle buttonArea = new Rectangle(0, 0, 60, 30);
    buttonArea.getStyleClass().add("button-panel");
    buttonGroup.getChildren().add(buttonArea);

    // left arrow button
    Arc leftButton = new Arc(12,16, 15, 15, -30, 60);
    leftButton.setType(ArcType.ROUND);
    leftButton.getStyleClass().add("left-arrow");

    // return to previous image
    leftButton.addEventHandler(MouseEvent.MOUSE_PRESSED,
        (mouseEvent) -> {
            System.out.println("busy loading? " + loading.get());
            // if no previous image or currently loading.
            if (currentIndex == 0 || loading.get()) return;
            int indx = gotoImageIndex(ButtonMove.PREV);
    });
}

```

```

        if (indx > -1) {
            loadImage(imageFiles.get(indx));
        }
    });

    // right arrow button
    Arc rightButton = new Arc(12,16, 15, 15, 180-30, 60);
    rightButton.setType(ArcType.ROUND);
    rightButton.getStyleClass().add("right-arrow");

    // advance to next image
    rightButton.addEventHandler(MouseEvent.MOUSE_PRESSED,
        (mouseEvent) -> {
            System.out.println("busy loading? " + loading.get());
            // if no next image or currently loading.
            if (currentIndex == imageFiles.size()-1
                || loading.get()) return;

            int indx = gotoImageIndex(ButtonMove.NEXT);
            if (indx > -1) {
                loadImage(imageFiles.get(indx));
            }
        });
    // add buttons to button group
    buttonGroup.getChildren().addAll(leftButton, rightButton);

    // 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));
    return buttonGroup;
}

/*
 * Create a progress indicator control to be centered.
 * @param scene The primary application scene.
 * @return ProgressIndicator a new progress indicator centered.
 */
private ProgressIndicator createProgressIndicator(Scene scene) {
    ProgressIndicator progress = new ProgressIndicator(0);
    progress.setVisible(false);
    progress.layoutXProperty()
        .bind(scene.widthProperty()
            .subtract(progress.widthProperty())
            .divide(2));
}

```

```
progress.layoutYProperty()
    .bind(scene.heightProperty()
        .subtract(progress.heightProperty())
        .divide(2));
return progress;
}

/*
 * Returns true if URL's file extensions match jpg, jpeg, png and gif.
 * @param url standard URL path to image file.
 * @return boolean returns true if URL's extension matches jpg, jpeg,
 * png and gif.
 */
private boolean isValidImageFile(String url) {
    List<String> imgTypes = Arrays.asList(".jpg", ".jpeg", ".png", ".gif", ".bmp");
    return imgTypes.stream()
        .anyMatch(t -> url.endsWith(t));
}

/**Adds the URL string representation of the path to the image file.
 * Based on a URL the method will check if it matches supported
 * image format.
 * @param url string representation of the path to the image file.
 */
private void addImage(String url) {
    if (isValidImageFile(url)) {
        currentIndex +=1;
        imageFiles.add(currentIndex, url);
    }
}

/**
 * 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.
 */
private 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;
}

/*
 * Returns a worker task (Task) which will off-load the image
 * on a separate thread when finished; the current image will
 * be displayed on the JavaFX application thread.
 * @param url string representation of the path to the image file.
 * @return
 */
private Task createWorker(final String url) {
    return new Task() {
        @Override
        protected Object call() throws Exception {
            // on the worker thread...
            Image image = new Image(url, false);
            Platform.runLater(() -> {
                // on the JavaFX Application Thread....
                System.out.println("done loading image " + url);
                currentImageView.setImage(image);
                progressIndicator.setVisible(false);
                loading.set(false); // free lock
            });
            return true;
        }
    };
}

/*
 * This method does the following loads an image,
 * updates a progress bar and spawns a new thread.
 * If another process is already loading
 * the method will return without loading.
 * @param url string representation of the path to the image file.
 */
private void loadImage(String url) {
    if (!loading.getAndSet(true)) {
        Task loadImage = createWorker(url);
        progressIndicator.setVisible(true);
        progressIndicator.progressProperty().unbind();
        progressIndicator.progressProperty()
            .bind(loadImage.progressProperty());
        new Thread(loadImage).start();
    }
}

```

```

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

```

Walking Through the Code

To begin our code walk-through, let's discuss the application's variables. Table 5-2 describes the variables that maintain state information for our photo viewer application.

Table 5-2. The PhotoViewer Instance Variables

Variable	Data Type	Example	Description
imageFiles	List<String>	/home/cdea/fun.jpg	A list of Strings, each containing an image's URL file path.
currentIndex	int	0	A zero-relative index into the imageFiles list; -1 means no images to view.
NEXT	enum	-	The NEXT enum value when the user clicks the right-arrow button.
PREV	enum	-	The PREV enum value when the user clicks the left-arrow button.
currentImageView	ImageView	-	The current image will be displayed onto the image view node.
progressIndicator	ProgressIndicator	-	The progress indicator control displayed during image loading.
loading	AtomicBoolean	-	A flag to prevent loading of an image until a prior image is finished loading. True means it is currently busy loading, otherwise the value is False.

Now that you are familiar with the variables that provide bookkeeping for our photo viewer application, I will explain the code, beginning with the usual start() method's body in Listing 5-3.

The start() method first sets the title, creates the root node, and then creates the scene. After instantiating a scene referencing the root node (Group), the code will style scene graph nodes via the method getStyledSheets().add(). In this scenario I created a JavaFX CSS file (photo-viewer.css) to style various nodes. This technique allows a developer to customize nodes by using an external CSS file instead of the programmatic approach with setter methods on node attributes. For now you can ignore the code statements relating to applying CSS styles; we'll discuss them in more detail in Chapter 6, on custom UIs.

The start() method next calls various private methods to create and assemble the parts that make up our photo viewer application, such as the current photo view area and a custom button panel.

Current Photo View Area: ImageView

After setting up the scene, the `start()` method creates the current image display area by invoking a private `createImageView()` method, which creates and returns a JavaFX `ImageView` object that has its width property bound to the root (parent) node's width. Binding the width means that resizing the application window will also resize the `ImageView`. And by invoking the `setPreserveRatio()` method set to true on the `ImageView` object, we enable the resizing ability to maintain the image's aspect ratio.

Drag and Drop: OnDragOver and OnDragDropped

Next the `start()` method invokes the private `setupDragNDrop()` method, which I created to attach handler code to respond to the drag-and-drop capability of the native desktop for loading and displaying images in our photo viewer application. A user can do any of the following: drag a file from their host file explorer (finder), drag a URL pointing to a file from a browser's address bar, or drag the image from the browser, assuming the address or URL path is a valid image file hosted on a web server. The icon to the left of the URL is dragged and the address line (filename) must have one of the image file extensions mentioned earlier.

The OnDragOver Event

The `setupDragNDrop()` method starts by setting handler code (using a lambda expression) for the `OnDragOver` event via the `setOnDragOver()` method. The `OnDragOver` event is triggered when the mouse is dragged over the scene's surface—in other words, holding down the mouse button over selected files that are dragged over the surface triggers the `OnDragOver` event. The following code snippet repeated from Listing 5-3 shows the `OnDragOver` event's handler code:

```
// Dragging over surface
scene.setOnDragOver((DragEvent event) -> {
    Dragboard db = event.getDragboard();
    if ( db.hasFiles()
        || (db.hasUrl()
            && isValidImageFile(db.getUrl())))
        event.acceptTransferModes(TransferMode.LINK);
    } else {
        event.consume();
    }
});
```

Notice here that the lambda expression has an event (`DragEvent`) parameter. The lambda expression code body obtains the `Dragboard` object from the `DragEvent` to determine if there are valid image files being dragged over the scene's surface. If so, the code sets the transfer mode (`TransferMode.LINK`); otherwise, we ignore or consume the event using the `consume()` method. Figure 5-2 earlier in the chapter shows the transfer mode (`TransferMode.LINK`) icon as the user is dragging the image onto the scene, thus providing visual feedback to the user.

The OnDragDropped Event

Next, we discuss the handler code to respond to an `OnDragDropped` event. The `OnDragDropped` event occurs when the user has released the mouse button after dragging over the surface of the scene the event handler code is shown in the following code snippet (see this code in context in Listing 5-3):

```
// Dropping over surface
scene.setOnDragDropped((DragEvent event) -> {
    Dragboard db = event.getDragboard();
    //String currentFile = null;
    // image from the local file system.
    if (db.hasFiles()) {
        db.getFiles()
            .stream()
            .forEach( file -> {
                try {
                    addImage(file.toURI().toURL().toString());
                } catch (MalformedURLException ex) {
                    ex.printStackTrace();
                }
            });
    } else {
        // image from some host
        addImage(db.getUrl());
    }
    if (currentIndex > -1) {
        loadImage(imageFiles.get(currentIndex));
    }

    event.setDropCompleted(true);
    event.consume();
}); // setOnDragDropped
```

Once again you see a lambda expression that has a parameter of type `DragEvent`. The code obtains the `Dragboard` object to detect whether there are one or more files, via the `hasFiles()` method. If so, the code iterates through each file to be added to the `imageFiles` (`List<String>`) variable. Before adding a URL-formatted string into the `imageFiles` list, the private method `addImage()` in Listing 5-3 validates each file using the private `isValidImageFile()` method. I created the `isValidImageFile()` method that uses the Stream API to determine whether the file extensions of the URL string represents a standard image file format, such as `.jpg`, `.gif`, or `.png`. By using the `anyMatch()` method with a functional interface of type `java.util.function.Predicte` you can easily evaluate whether the URL string ends with any of the image file extensions. The following code snippet iterates over the list of file formats to match the URL.

```
List<String> imgTypes = Arrays.asList(".jpg", ".jpeg", ".png", ".gif", ".bmp");
return imgTypes.stream()
    .anyMatch(t -> url.endsWith(t));
```

Next in the `OnDragDropped` event's handler, notice the code to make sure the current index is greater than `-1`, which indicates that a valid image can be loaded and displayed. Once an image is determined to be valid, the private method `loadImage()` creates a worker task (`Task`) to begin loading an image file.

The loadImage() method is repeated here from Listing 5-3 in the following code snippet . It begins by checking the loading (AtomicBoolean) variable before actually loading the image. This bit of code prevents the user from rapidly clicking on the buttons and spawning many images, loading a task for each one. Also, if the loading variable is not set to true, the code will set it to true before beginning the image-loading process via the getAndSet(true) method.

```
private void loadImage(String url) {  
    // do not begin task until current  
    // task is finished loading (atomic)  
    if (!loading.getAndSet(true)) {  
        Task loadImage = createWorker(url);  
        progressIndicator.setVisible(true);  
        progressIndicator.progressProperty().unbind();  
        progressIndicator.progressProperty()  
            .bind(loadImage.progressProperty());  
        new Thread(loadImage).start();  
    }  
}
```

This snippet shows the code related to creating a worker thread by invoking a private method. Here, `createWorker()` is a convenience method to create worker tasks for asynchronous callbacks. After it returns a worker (Task) instance, the load progress will be bound to the progress indicator, described in Table 5-2 earlier. Essentially, the progress indicator will provide an animation as visual feedback to the user that an image is in the process of loading. If there was a previously bound task, the code will unbind the progress indicator and re-bind it to the newly created task. Finally, the worker thread is simply started. Keep in mind that the image is loaded in a separate thread and not on the JavaFX application GUI thread.

Finishing off the `OnDragDropped` event's handler code, the event will complete the drop operation by invoking the `setDropCompleted(true)` and `consume()` methods, respectively.

The Custom Button Panel

After setting up the drag-and-drop capabilities the `start()` method, Listing 5-3 invokes the private method `createButtonPanel()` to create a custom button panel to contain the Previous and Next buttons. These buttons allow the user to advance through images to load and display.

To create custom buttons, I've used ordinary JavaFX shapes, which are drawn onto a `Group` node. To create the Previous and Next buttons I used JavaFX's `javafx.scene.shape.Arc` shape to build the left and right arrows on top of a small transparent rounded rectangle, `javafx.scene.shape.Rectangle`.

Next in the `createButtonPanel()` method in Listing 5-3 you'll find `EventHandler` code that listens to mouse-pressed events to load and display the appropriate image based on the enums `ButtonMove.PREV` and `ButtonMove.NEXT`. Attaching handler code for a mouse-press event for the Previous and Next buttons is quite easy. The code simply invokes the `addEventHandler()` by passing in the event that it is interested in (`MouseEvent.MOUSE_PRESSED`) and a closure that represents the handler code that will respond when the button (the arc shape) is pressed. Shown in Listing 5-4, the code determines whether the application is currently busy loading before proceeding by checking the loading (AtomicBoolean) flag.

Listing 5-4. The Attached mouse press event handler code for the custom Previous button (leftButton)

```
Arc leftButton = //... create an Arc
leftButton.addEventHandler(MouseEvent.MOUSE_PRESSED,
    (MouseEvent) -> {
    System.out.println("busy loading? " + loading.get());
    // if no previous image or currently loading.
    if (currentIndex == 0 || loading.get()) return;
    int indx = gotoImageIndex(ButtonMove.PREV);
    if (indx > -1) {
        loadImage(imageFiles.get(indx));
    }
});
```

Assuming the loading flag is false, meaning not busy loading, the code will proceed by returning the index into the `imageFiles` list of the next image to display.

The right button's (`rightButton`) event handler is similar. The only thing different is determining whether the Previous or Next button was pressed via the `ButtonMove` enum. This is passed to the `gotoImageIndex()` method in Listing 5-3 to determine whether an image is available in that direction.

Finishing up the assembling of our button panel (`buttonGroup`), the code binds the button panel to the scene's width and height, which will dynamically reposition it relative to the bottom-right corner of the scene as the user resizes the window. Here, we bind the `translateXProperty()` to the scene's width property by subtracting the `buttonArea`'s width shown in Listing 5-5 and bind the `translateYProperty()` based on the scene's height property.

Listing 5-5. The bindings for the button panel to float relative to the lower right corner of the window(scene)

```
// move button group when the scene is resized
buttonGroup.translateXProperty()
    .bind(scene.widthProperty()
        .subtract(buttonArea.getWidth() + 6));
buttonGroup.translateYProperty()
    .bind(scene.heightProperty()
        .subtract(buttonArea.getHeight() + 6));
```

Finally in Listing 5-3, the call to the private method `createProgressIndicator()` in the `start()` method returns a progress indicator bound to the scene's center area. You'll notice the `setVisible()` method (in `loadImage()` and in the Task created by `createWorker()`) to show or hide the progress indicator before and after loading an image.

Animation

An animation is an illusion of motion when images change over time. Quite similar to a cartoon flip book, each page represents a frame or picture that will be displayed on a timeline for a period of time. In JavaFX you will be working with the animation API (`javafx.animation.*`). In this section you will learn about key values, key frames, timelines, and transitions. Finally, to demonstrate the animation APIs, we will explore a slightly modified photo viewer application with additional enhancements.

What are Key Values?

The JavaFX animation API allows you to assemble timed events that can interpolate over property-based values (Properties) in order to produce animated effects. For example, to produce a fade-out effect, you would target a node's opacity property to interpolate its value starting from 1 (fully opaque) to 0 (transparent) over a period of time. Listing 5-6 defines a KeyValue instance that targets a rectangle node's opacity property starting at 1 and ending with the value 0. By default, a KeyValue object will have a linear interpolator.

Listing 5-6. A KeyValue object to fade out by interpolating the opacity property from 1 to 0

```
Rectangle rectangle = new Rectangle(0, 0, 50, 50);
KeyValue keyValue = new KeyValue(rectangle.opacityProperty(), 0);
```

The KeyValue object doesn't actually interpolate the value, but simply defines the start and end values of a property to interpolate between. Also, the KeyValue can be defined with different types of interpolators such as linear, ease in, or ease out. For example, shown in Listing 5-7 is code that defines a key value that will animate a rectangle from left to right by 100 pixels having an Interpolator.EASE_OUT interpolator. Easing out will slow down the fade before the end key value.

Listing 5-7. A KeyValue object to move a rectangle from left to right 100 pixels with an interpolator that starts to slow down when it reaches the end value

```
Rectangle rectangle = new Rectangle(0, 0, 50, 50);
KeyValue keyValue = new KeyValue(rectangle.xProperty(), 100, Interpolator.EASE_OUT);
```

Please visit the Javadocs documentation for available interpolators (`javafx.animation.Interpolator`). By default, KeyValue constructors that don't specify an interpolator will use a linear interpolation (`Interpolator.LINEAR`), which evenly distributes the values.

What are Key Frames?

When an animation (Timeline) occurs, each timed event, called a keyframe (a KeyFrame object), is responsible for interpolating key values (KeyValue objects) over a period of time (`javafx.util.Duration`). When creating a KeyFrame object, the constructor requires a timed duration (Duration) to interpolate over key values. KeyFrame constructors all accept one or more key values by using a variable containing an argument list or a collection of KeyValue objects.

To demonstrate moving a rectangle in a diagonal direction from upper left to lower right, Listing 5-8 defines a keyframe that has a duration of 1000 milliseconds and two key values that represent the rectangle's x and y properties. The example defines a keyframe to move a rectangle's upper-left corner (0, 0) to point (100, 100) in one second or 1000 milliseconds.

Listing 5-8. Keyframe defined

```
Rectangle rectangle = new Rectangle(0, 0, 50, 50);
KeyValue xValue = new KeyValue(rectangle.xProperty(), 100);
KeyValue yValue = new KeyValue(rectangle.yProperty(), 100);

KeyFrame keyFrame = new KeyFrame(Duration.millis(1000), xValue, yValue);
```

A KeyFrame object is defined to interpolate over the x and y properties. In this example a keyframe is defined that will animate the rectangle to move in a southeast direction for one second.

What is a Timeline?

A Timeline is one animation sequence consisting of many KeyFrame objects. Each KeyFrame object is run sequentially. Because a Timeline is a subclass of the abstract javafx.animation.Animation class, it has standard attributes such as its cycle count and auto-reverse that you can 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 property is a Boolean flag to indicate that the animation can play the timeline backward (the keyframes in reverse order). By default, the cycle count is set to 1, and auto-reverse is set to false.

To add key frames to a Timeline object, use the getKeyFrames().addAll() method. Listing 5-9 demonstrates a timeline playing indefinitely (looping) with auto-reverse set to true to enable the animation sequence to go backwards. A Timeline instance is created with a cycle count set to run indefinitely and auto-reverse set to true to allow the animation sequence to go backwards.

Listing 5-9. A Timeline instance

```
Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
timeline.getKeyFrames().addAll(keyFrame1, keyFrame2);
timeline.play();
```

With this knowledge of timelines, you can now animate any scene graph node in JavaFX. Although you have the ability to create timelines in a low-level way, this technique can become very cumbersome for simple animations. You are probably wondering whether there are easier ways to express common animation effects such as fading, scaling, and translating.

Good news! JavaFX has stock (built-in) transition animations (the Transition class), which are convenience classes to perform common animated effects. Here are some of the most common animation effects classes, followed by a brief description of each:

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

The fade transition (FadeTransition) targets the node's opacity property for a fading animation effect. The path transition (PathTransition) enables a node to follow a generated path (Path). Scale transition (ScaleTransition) targets a node's scaleX, scaleY, and scaleZ properties to resize a node. Translate transition (TranslateTransition) targets a node's translateX, translateY, and translateZ properties to move a node across the screen. I've only listed a few common transitions; however, there are many more transitions I haven't discussed, so I encourage you to look at the Javadocs documentation for subclasses of the javafx.animation.Transition class.

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. To demonstrate these built-in transitions let's look at an example.

A News Ticker Example

To demonstrate some of the common transitions mentioned earlier, I decided to add some of them to the prior photo viewer application, and also add a news ticker-like control. Because the new ticker code is similar to the previous photo viewer example, I will describe only the relevant code that performs the animation transitions. To see the full source code, visit the book's website to download the code for this chapter. The example project is called `NewsTickerPhotoViewer`.

Before discussing any code, let's go over the suggested enhancements of our news-ticker-photo-viewer application. Figure 5-3 is the News Ticker photo viewer application containing the suggested enhancements.



Figure 5-3. The News Ticker Photo Viewer application

Following are the enhancements and their descriptions:

- **Fade in/out photo:** Apply a fade effect to fade out the current image (`ImageView`) and fade in the newly loaded image when using the Previous and Next buttons. Use the `FadeTransition` and `SequentialTransition` classes.
- **Fade in/out button panel:** Apply a fade effect to fade-out the button panel control when the mouse cursor is not hovering inside the application window (`Scene`), otherwise fade-in (cursor is hovering inside application window). Use the `FadeTransition` class.
- **Add a news ticker-like control:** Scroll text content from right to left in a rectangular region positioned near the bottom and just left of the button panel control. Use the `TranslateTransition` class.

Adding a Fade in/out Transition to Photos

To add a nice touch to the photo viewer application, I decided to add a fade transition effect between image views. Assuming you have more than one image loaded, you can click the Next or Previous buttons to see the transition effect in action. Because these are actually two separate actions (fade-out and fade-in), they need to be performed in consecutive order (sequentially).

In JavaFX the animation API also provides two types of higher-level transition classes that are capable of performing multiple animations (transitions). These two types are sequential (`SequentialTransition`) and parallel (`ParallelTransition`). As the name implies, the sequential transition contains one or more animations that run in sequence. An example would be to animate a rectangle by scaling it first and then rotating it. Parallel transitions run child animations simultaneously (in parallel). Using the same example, the rectangle would scale and rotate at the same time.

In our news ticker photo viewer application, you will use the `SequentialTransition` class to contain two `FadeTransition` animations. To augment the existing photo viewer application I added the private method `transitionByFading()`, shown in Listing 5-10. This method creates and returns a `SequentialTransition` instance that contains two `FadeTransition` child animation transitions.

Listing 5-10. The method `transitionByFading()` to fade-out the current image and fade-in the next image

```
private SequentialTransition transitionByFading(Image nextImage,
                                              ImageView imageView) {

    // fade out image view node
    FadeTransition fadeOut =
        new FadeTransition(Duration.millis(500), imageView);
    fadeOut.setFromValue(1.0);
    fadeOut.setToValue(0.0);
    fadeOut.setOnFinished(actionEvent -> imageView.setImage(nextImage));

    // fade in image view node
    FadeTransition fadeIn =
        new FadeTransition(Duration.millis(500), imageView);
    fadeIn.setFromValue(0.0);
    fadeIn.setToValue(1.0);
```

```

// fade out image view, swap image and fade in image view
SequentialTransition seqTransition =
    new SequentialTransition(fadeOut, fadeIn);

return seqTransition;
}

```

The method begins by receiving the `nextImage` argument with the preloaded next image and the argument `imageView`, which references the `ImageView` node pointing to the current image. The code first creates a fade-out transition that targets the `ImageView` object's `opacity` property by setting the methods `setFromValue(1.0)` and `setToValue(0.0)` on the associated `FadeTransition` instance. Once the start and end values are defined, handler code (lambda) is added to the `OnFinished` event via the `setOnFinished()` method to switch out the image to be set as the new image. In other words, when the image view node is finished fading out, set the image displayed to the next image. Because the image view's opacity is at zero, the user cannot see the image until the fade-in transition occurs.

After creating a fade-out transition, the code creates the fade-in. This transition is similar to the fade-out except that the From and To values are reversed; the `opacity` property on the image view will start fading with a value of zero (invisible) and end with a value of one (fully opaque). Finally, the code completes the method by creating a `SequentialTransition` instance that will contain the two fade transitions just mentioned.

After adding the `transitionByFading()` method to the main application class (`NewsTickerPhotoViewer.java`), you will need to wire it up to the code that loads the next image (within the `createWorker()` method). To perform the sequential fade transition effect, you can now appropriately invoke the private `transitionByFading()` method. Shown in Listing 5-11, within the `call()` method of our returned `Task` is the call to the `transitionByFading()` method. Also notice the code that invokes the `play()` method on the `SequentialTransition` object to kick off the animation sequence.

Listing 5-11. Updated `createWorker()` method to invoke the new `transitionByFading()` method instead of setting the next image right away

```

private Task createWorker(final String url) {
    return new Task() {
        @Override
        protected Object call() throws Exception {
            Image image = new Image(url, false);
            Platform.runLater(() -> {
                // New code:
                SequentialTransition seqTransition =
                    transitionByFading(image, currentImageView);
                seqTransition.play();

                // Old code: currentImageView.setImage(image);

                progressIndicator.setVisible(false);
                loading.set(false); // free lock
            });
            return true;
        }
    };
}

```

Adding a Fade in/out Transition to the Button Panel

Another enhancement to our photo viewer application is a fade transition effect on the custom button panel control. When the mouse cursor is hovering over the application area (Scene), the button panel control will fade in to become fully opaque, and when it is not hovering, the button panel control will fade out to become invisible.

Listing 5-12 adds this new behavior to the `createButtonPanel()` method of the original photo viewer application. For brevity I've omitted most of the button panel creation code right before the code for fading the button panel. The additional code is at the end of the method `createButtonPanel()`; it simply attaches handler code to the `OnMouseEntered` and `OnMouseExited` events on the main application scene object. You will notice that each fade transition uses a half-second duration, specified using the `Duration.millis(500)` method to perform the animation.

Listing 5-12. The fade in and out transition effect added to the button panel method `createButtonPanel()`

```
/**  
 * Returns a custom created button panel containing left and right  
 * buttons to see the previous and next image.  
 * @param scene  
 * @return Group node.  
 */  
private Group createButtonPanel(Scene scene){  
  
    // Code omitted...  
    // ... The button panel code goes here.  
  
    // Fade in button controls  
    scene.setOnMouseEntered((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((MouseEvent me) -> {  
        FadeTransition fadeButtons =  
            new FadeTransition(Duration.millis(500), buttonGroup);  
        fadeButtons.setFromValue(1);  
        fadeButtons.setToValue(0);  
        fadeButtons.play();  
    });  
  
    return buttonGroup;  
}
```

Creating a News Ticker Control

The last core functionality of this example is to add a news ticker control to our photo viewer application. Listing 5-13 shows a method called `createTickerControl()`, which returns a custom news ticker control that is added to the scene graph. After you've had a chance to look over the implementation in Listing 5-13, I will explain how the custom news ticker operates.

Listing 5-13. A method that returns a custom news ticker control to be added to the main scene graph

```
/**
 * Returns a news ticker control
 * @param scene
 * @param buttonGroup
 * @return
 */
private Group createTickerControl(Stage stage, double rightPadding) {
    Scene scene = stage.getScene();

    // create ticker area
    Group tickerArea = new Group();
    Rectangle tickerRect = new Rectangle(scene.getWidth(), 30);
    tickerRect.getStyleClass().add("ticker-border");

    Rectangle clipRegion = new Rectangle(scene.getWidth(), 30);
    clipRegion.getStyleClass().add("ticker-clip-region");
    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(rightPadding));
    clipRegion.widthProperty()
        .bind(scene.widthProperty()
            .subtract(rightPadding));
    tickerArea.getChildren().add(tickerRect);

    // news feed container
    FlowPane tickerContent = new FlowPane();

    // add some news
    Text news = new Text();
    news.setText("JavaFX 8.0 News! | 85 and sunny | :)");
    news.setFill(Color.WHITE);
    tickerContent.getChildren().add(news);

    DoubleProperty centerContentY = new SimpleDoubleProperty();
    centerContentY.bind(
        clipRegion.heightProperty())
}
```

```

        .divide(2)
        .subtract(tickerContent.heightProperty()
            .divide(2)));
    }

    tickerContent.translateYProperty().bind(centerContentY);

    tickerArea.getChildren().add(tickerContent);

    // scroll news feed
    TranslateTransition tickerScroller = new TranslateTransition();
    tickerScroller.setNode(tickerContent);
    tickerScroller.setDuration(
        Duration.millis(scene.getWidth() * 40));
    tickerScroller.fromXProperty()
        .bind(scene.widthProperty());
    tickerScroller.toXProperty()
        .bind(tickerContent.widthProperty()
            .negate()));

    // when ticker has finished, reset and replay ticker animation
    tickerScroller.setOnFinished((ActionEvent ae) -> {
        tickerScroller.stop();
        tickerScroller.setDuration(
            Duration.millis(scene.getWidth() * 40));
        tickerScroller.playFromStart();
    });
    // start ticker after nodes are shown
    stage.setOnShown( windowEvent -> {
        tickerScroller.play();
    });
}

return tickerArea;
}

```

The news ticker control created here will resemble a stock ticker, as news headlines scroll from right to left inside a rounded rectangular region. As the news content scrolls by, it will enter the rectangular region from the right edge and exit the left edge. To have the news content appear only in the rectangular area, you will want the news content to be clipped. Clipping the area creates a viewport that shows only the pixels inside the rounded rectangular region.

The method `createTickerControl()` begins by accepting `stage` and `rightPadding` parameters. These parameters are primarily used to help determine size and dimensions to position the news ticker control dynamically when the window is resized. The `stage` parameter is the application's main `Stage` object, and the `rightPadding` is spacing to reduce the width of the ticker control to make room for the button panel (`Group`) node.

The method body starts off by creating a `Group` node as the parent (`tickerArea`) to hold all the components a ticker control comprises. Next, the code creates a stylized rectangle with rounded corners and a thin white border. This rectangle's styling is applied by adding the `ticker-border` style class; this class is defined in the JavaFX CSS file `news-ticker-photo-viewer.css`, which is set earlier in the application's `start()` method. The CSS file shown in Listing 5-14 is a code snippet from the

`start()` method that initializes the scene's nodes with the `news-ticker-photo-viewer.css` file. We will discuss how to style nodes using CSS in detail in Chapter 6.

Listing 5-14. Initializing the scene's nodes with the JavaFX CSS based file

```
scene.getStylesheets()
    .add(getClass().getResource("news-ticker-photo-viewer.css")
        .toExternalForm())
);
```

After creating the ticker's styled (white) border, the code also creates an identical rectangle only to be used as a clipped region. Having a clipped region creates a viewable area in the shape of a rounded rectangle. This clipped rectangle represents a region that will be set on the parent Group node (`tickerArea`) by using the `setClip(someRectangle)` method. Figure 5-4 shows a rounded rectangular area as a clip region.



Figure 5-4. Setting the rounded rectangular clip region on the Group node object

With the ticker area (Group) created, the code will begin binding the ticker area's `translateX` and `translateY` properties, which are based on the scene's dimensions (width and height). The code will also bind the ticker area's children nodes' `translateX` and `translateY` properties.

By binding these properties, the ticker control's nodes 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 beside the button panel control. Listing 5-15 implements the property bindings for the ticker control's `textArea`, `tickerRect`, and `clipRegion`.

Listing 5-15. News Ticker bindings for repositioning control when the window is resized

```
// 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(rightPadding));
clipRegion.widthProperty()
    .bind(scene.widthProperty()
        .subtract(rightPadding));
```

Now that the main ticker control is finished, we need to create news to feed into it. Listing 5-16 begins by creating a `FlowPane` instance assigned to the variable `tickerContent`.

Listing 5-16. Creating a news feed container

```
// news feed container
FlowPane tickerContent = new FlowPane();

// add some news
Text news = new Text();
news.setText("JavaFX 8.0 News! | 85 and sunny | :)");
news.setFill(Color.WHITE);

tickerContent.getChildren().add(news);

// center the ticker content vertically
DoubleProperty centerContentY = new SimpleDoubleProperty();
centerContentY.bind(clipRegion.heightProperty()
    .divide(2)
    .subtract(
        tickerContent.heightProperty()
            .divide(2)));
tickerContent.translateYProperty()
    .bind(centerContentY);

tickerArea.getChildren().add(tickerContent);
```

The `tickerContent` is a container node to hold news headlines (child nodes) to be scrolled in the ticker control. The `tickerContent` node is nice because instead of animating each child node (`Text`), the code will simply scroll the `tickerContent` (`FlowPane`). The idea of using a container node (`FlowPane`) is analogous to moving a cargo train with box cars instead of moving each box car individually.

For demonstration purposes I merely created one `Text` node with a hard-coded news headline. As an exercise, you may want to refactor the code to expose the `ticker content` node to allow users of the API to add real news headlines.

Finally, the last bit of code from our `createTickerControl()` method is the scrolling of the ticker content. Listing 5-17 shows the code that scrolls the ticker content from right to left. To animate a node to appear to scroll, I used the `TranslateTransition` class. You'll notice I target the `ticker content` (`FlowPane`) node via the `setNode()` method. Next, the code sets the duration using the method `Duration.millis()`. It takes the scene's width and multiplies it by 40 (milliseconds) to equal the total time for the node to travel. Finally, the code will interpolate the node's `translateX` property from the right edge of the screen to the negative width of the `ticker content` node.

Listing 5-17. Scroll ticker content using the `TranslateTransition` class.

```
// scroll news feed
TranslateTransition tickerScroller = new TranslateTransition();
tickerScroller.setNode(tickerContent);
tickerScroller.setDuration(
    Duration.millis(scene.getWidth() * 40));
tickerScroller.fromXProperty()
    .bind(scene.widthProperty());
```

```
tickerScroller.toXProperty()
    .bind(tickerContent.widthProperty()
        .negate());  
  
// when ticker has finished reset and replay ticker animation
tickerScroller.setOnFinished((ActionEvent ae) -> {
    tickerScroller.stop();
    tickerScroller.setDuration(
        Duration.millis(scene.getWidth() * 40));
    tickerScroller.playFromStart();
});
```

When the ticker's news has scrolled completely off 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). Here, the method adds handler code (a lambda expression) for the OnFinished event to respond when the tickerScroller (TranslateTransition) transition has completed one cycle of the animation.

Once the animation is defined and the Stage is shown, the code will invoke the play() method, based on the OnShown window event (see Listing 5-18). To animate the ticker's content, certain widths and heights of nodes and their properties aren't realized until nodes have been displayed on the scene graph. That is why the tickerScroller's play() method is invoked after the stage has been shown.

Listing 5-18. Start the news ticker scroll animation

```
// start ticker after nodes are shown.
stage.setOnShown( windowEvent -> {
    tickerScroller.play();
});
```

Summary

In this chapter you learned how to load images using the `javafx.scene.image.Image` class' constructor, which accepts the standard URL specification format of files located locally or on a remote web server. Then you got a chance to implement a photo viewer application with drag-and-drop capability to load and display images. Next, you looked at the fundamentals of JavaFX's animation API to create key values, key frames, and timelines. After learning about the fundamental concepts of animating any node using timelines, you looked at convenient animation classes called *transitions*. You learned that Transition classes have simplified the implementation of common animation effects such as fading, scaling, and translating. Finally, you ended the chapter by exploring an example that augmented the photo viewer application with animated enhancements, including a custom news ticker control.

Chapter 6

Custom UIs

Wouldn't it be nice if you could change the appearance of your application without altering its functionality? Many companies spend lots of money and time to design user interfaces (UIs). Often companies will mock up several designs before deciding how their UI should look and behave. Depending on the scope, many companies also build UIs that support different screen dimensions. Typically, tablets, smart phones, TVs, and desktops are among the main challenges UI designers face.

In this chapter you will learn about theming and how to customize applications by applying various themes (look and feels). Next, you will learn about the fundamentals of JavaFX CSS styling, which will enable you to change attributes of any Scene graph node, such as colors, backgrounds, margins, and so on. After learning about CSS styling, you will apply these same techniques while using the amazing Scene Builder application tool. Scene Builder is a free tool that enables you to create and define UI screens graphically and save them as a JavaFX FXML-formatted file. Last, you will learn about creating custom controls. Here you will learn how to build UI controls that aren't part of the JavaFX 8 built-in controls. Some custom controls that might come to mind are number pads, gauges, clocks or futuristic context menus.

In this chapter you will be learning the following tools and concepts:

- Theming
- JavaFX CSS styling
- Scene Builder
- Custom controls

Theming

You have probably heard the terms *skinning* and *look and feel* in the context of designing user interfaces. These terms are often used interchangeably, and both reflect the basic concept of *theming*. Theming has the idea of transforming an entire application's appearance without changing its underlying functionality. An example of theming would be to change an application's UI layout and controls from a stylized business theme to a sports-like theme. Usually, companies will try to

skin an application to resemble their logo. They'll also use beautiful typefaces (typography) selected to personify their corporate culture or target audience.

In JavaFX you have the ability to create, modify, or use existing themes to skin your applications. Later, we will discuss how to create themes, but for now I'll show you how to apply existing themes (skins) in JavaFX applications. To demonstrate applying themes to an application's UI, I created the JavaFX 8 application depicted in Figure 6-1, which allows a user to choose and apply various UI look-and-feel themes.

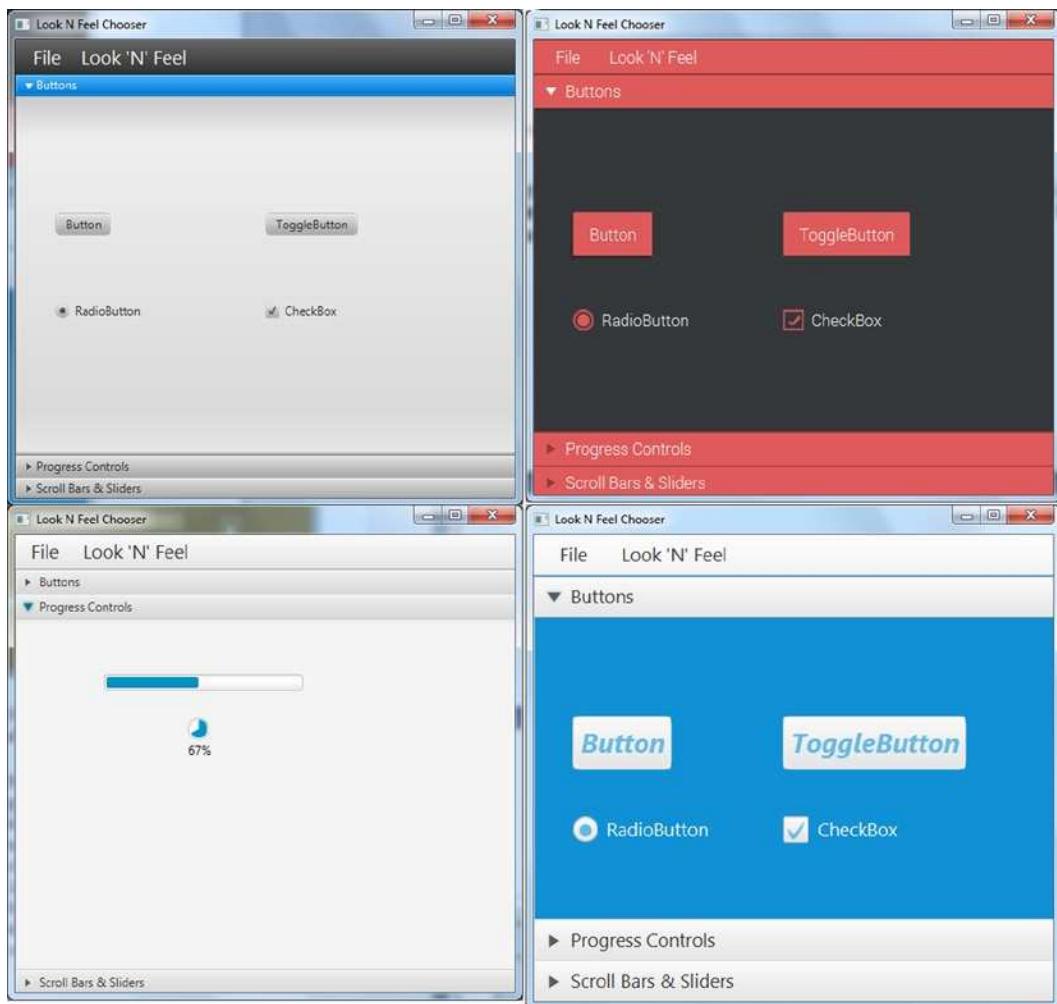


Figure 6-1. Setting an application's look and feel using JavaFX CSS styling. Moving counterclockwise from the upper-left are the Caspian, Modena, Sky and Flat Red look-and-feel themes

As you can see, there are four themes applied to the same application. At the upper left is the familiar Caspian theme from Java FX 2.X. Caspian has served well over the years and still has quite the professional look. At the lower left you see an accordion UI control expanded to show progress

controls using JavaFX 8's new default theme, Modena. The Modena theme seems brighter, cleaner, and more modern.

At the lower right, our next theme, called the Sky look and feel, was created by yours truly (Carl Dea). My intent was to create a theme for touch displays such that the controls would be slightly enlarged for adult-sized fingers. I wanted a friendly feel and so the cool blue sky came to mind.

Last but not least at the upper right is the Flat Red look and feel, created by Gerrit Grunwald. The Flat Red uses fonts to give text on controls a crisp and clean look. Another nice feature of the Flat Red theme is that it uses CSS effects based on pseudo-class state changes of UI controls, such as a mouse press on a slider control. Don't worry if you don't know yet what pseudo-classes are, because later in this chapter we will discuss pseudo-class selectors in detail. Mr. Grunwald's look and feel is also an excellent candidate for touch displays.

Before jumping into code I want to give a shout-out to individuals in the JavaFX community who have crafted beautiful skins and themes that you will find most inspiring. For those interested in the OS X (Mac desktop) native look and feel, Claudine Zillmann (@etteClaudette) has created AquaFX (with Elements). AquaFX has an API that allows developers to style applications easily with the native OS X look, but also to theme the UI with colors other than the usual aqua blue.

If you prefer the Windows 8 Metro native look and feel, Pedro Duque Vieira (@P_Duke) has created JMetro. The JMetro look and feel can style applications that resemble many applications you'll find on Microsoft Surface devices or Windows 8 desktops. JMetro also has the ability to switch from a dark or light theme.

The last skin I want to mention is a really nice look and feel for touch displays that has a Roku-like or Apple TV-like interface; it's called Flatter, by Hendrick Ebbers (@hendrikEbbers). Flatter was designed for the BoxFX project. BoxFX is another project Mr. Ebbers has created by using a Raspberry Pi device running JavaFX 8 with the Flatter look and feel. The following are their respective websites:

- AquaFX (with Elements): <http://aquafx-project.com/project.html>
- Jmetro: <http://pixelduke.wordpress.com/category/metro> or <https://github.com/JFXtras/jfxtras-styles>
- Flatter: <http://www.guigarage.com/2013/09/flatter>
- BoxFX: <http://www.guigarage.com/2013/08/boxfx-javaone-preview-1>

Applying JavaFX CSS Themes

Now that you have seen a glimpse of some great-looking skins and themes, let's see how to apply JavaFX CSS-based themes. JavaFX has the capability to apply Cascading Style Sheets to the Scene graph and its nodes in very much the way Web developers use CSS style sheets against HTML5 elements. These CSS style sheets are external files containing attributes and values to style JavaFX nodes. An example JavaFX CSS file would look something like Listing 6-1.

Listing 6-1. A JavaFX CSS File Containing Styling Attributes

```
/* sample.css */  
.button {  
    -fx-text-fill: brighter-sky-blue;  
    -fx-border-color: rgba(255, 255, 255, .80);  
    -fx-border-radius: 8;  
    -fx-padding: 6 6 6 6;  
    -fx-font: bold italic 20pt "LucidaBrightDemiBold";  
}  
/* ...more styling */
```

This code contains a CSS class style definition for JavaFX button controls. Button UI controls by default have a class style `button`, which maps to `-fx-` prefixed properties. You will learn about JavaFX CSS styling later in this chapter, but for now let's discuss how to swap between the CSS style sheets that will dynamically skin an application.

In this section you will learn two ways to apply CSS style sheets as look-and-feel themes onto JavaFX applications. The first way to apply CSS style sheets is by invoking the static `setUserAgentStylesheet()` method on the JavaFX Application (`javafx.application.Application`) class. This static method styles every scene and all child nodes in a JavaFX application.

The second way of applying a look and feel is by invoking a Scene object's `getStylesheets().add()` method. Unlike the first method of applying a look and feel, the `getStylesheets().add()` method is used to style a single scene and its child nodes.

Using the `setUserAgentStylesheet(String URL)` Method

The `setUserAgentStylesheet(String URL)` method accepts a valid URL string representing the JavaFX CSS file. Typically, CSS files are bundled inside a jar application; however, they can reside on the local file system or a remote web server. When the CSS file is in the classpath, the call to the `getClass().getResource("path/some_file.css").toExternalForm()` method will find the CSS file and produce a URL String for accessing the file.

The following code snippet loads the `sample.css` file as the current look and feel. The `sample.css` file in the code snippet is co-located where the current class is located. In other words, your Java class and your CSS file are in the same directory, so there is no need for a path in front of the file name.

```
Application.setUserAgentStylesheet(getClass().getResource("sample.css")  
    .toExternalForm());
```

The `setUserAgentStylesheet()` method applies styling globally to *all* scenes owned by an application, such as context menus and child popup dialog windows. JavaFX 8 currently contains two style sheets, Caspian and Modena, which serve as default cross-platform look and feel skins. Because the two style sheets are predefined, you can easily switch between them using the `setUserAgentStylesheet()` method. The following code shows how to switch between the Caspian and Modena look and feel style sheets.

```
// Switch to JavaFX 2.x's CASPIAN Look and Feel.
Application.setUserAgentStylesheet(STYLESHEET_CASPIAN);

// Switch to JavaFX 8's Modena Look and Feel.
Application.setUserAgentStylesheet(STYLESHEET_MODENA);
```

A great way to learn how the pros do it (creating skins) is to look at good examples. You can extract the CSS files (`caspian.css` and `modena.css`) from the `jfxrt.jar` file or view the JavaFX source code, located at <http://openjdk.java.net>.

Note When we invoke the `setUserAgentStylesheet(null)` method by passing in a null value, the default style sheet (Modena) will be automatically loaded and set as the current look and feel. However, if you are using JavaFX 2.x, the default style sheet will be Caspian.

Using Scene's `getStylesheets().add(String URL)` Method

By calling the `getStylesheets().add()` method, you can style a single scene and its child nodes. As in the previous call to `setUserAgentStylesheet()`, you would pass in a URL string that represents a JavaFX CSS file. Later, you will see an example application that switches between different styles by using either the `setUserAgentStylesheet()` or the `getStylesheets().add()` method. You are probably wondering what the exact differences between the two methods are.

Because creating an entire look and feel would require hundreds if not thousands of lines of code in order to style every UI control in JavaFX, I basically created very small look-and-feel style sheet files to demonstrate the ability to swap themes. The simple example CSS files have just a handful of CSS styles for a small amount of UI controls that would be styled. In other words, the example's small CSS files are loaded via the `getStylesheets().add()` method, which styles only some of the nodes and not every node in the application.

Because the default style sheet (`Modena.css`) is loaded from the prior call via the `setUserAgentStylesheet(null)` method, the smaller custom CSS styling file can then “piggyback” onto the default style sheet. What’s nice is that you wouldn’t have to start from scratch to create a brand new look and feel. The following snippet of code initially invokes the `setUserAgentStylesheet()` method with a null value, which loads Modena as the default look and feel and then sets the scene’s additional styling by invoking the `getStylesheets().add()` method.

```
Application.setUserAgentStylesheet(null); // defaults to Modena

// apply custom look and feel to the scene.
scene.getStylesheets()
    .add(getClass().getResource("my_cool_skin.css")
        .toExternalForm());
```

An Example of Switching Themes

To show how to switch between various CSS style sheets (aka *skins* or *themes*), I've created an example application called the Look N Feel Chooser, which allows you to choose between different predefined look-and-feel themes. Figure 6-2 shows the Look and Feel Chooser example application initially displaying accordion UI control panes with common UI controls, beginning with the Modena look and feel.

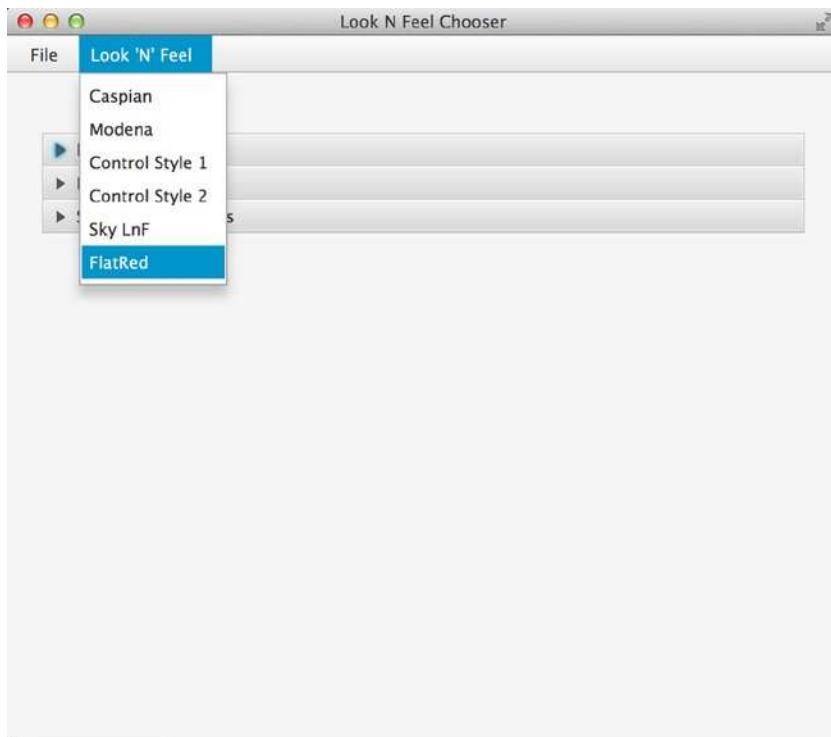


Figure 6-2. Predefined look-and-feel themes to choose from the Look 'N' Feel menu

In Figure 6-2 you'll notice the menu options to select a look and feel (skin). Once a look and feel has been selected, the application's appearance will change dynamically. Figure 6-3 shows the application switched to the FlatRed look and feel.

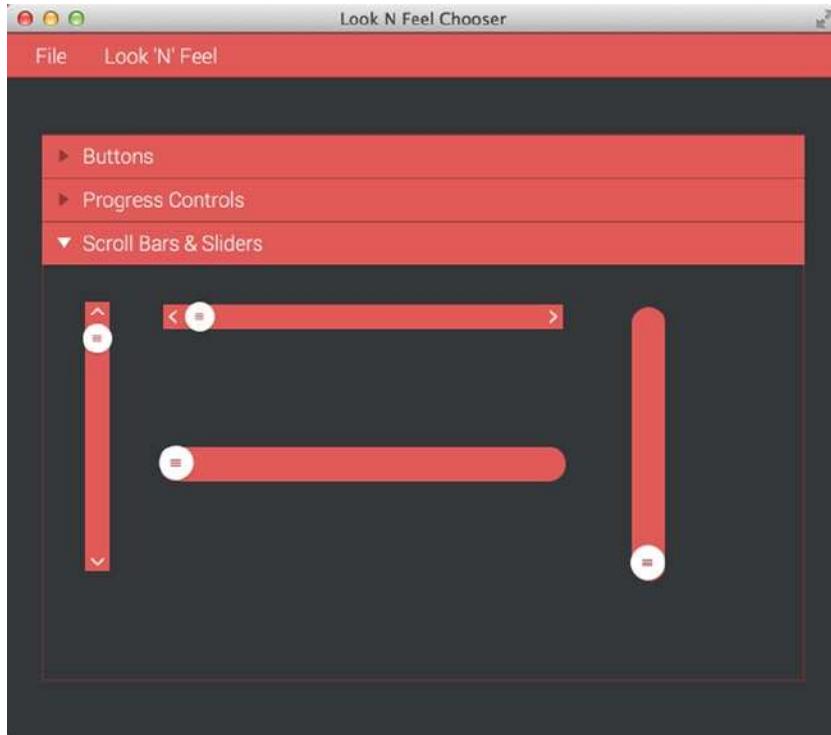


Figure 6-3. The Look N Feel Chooser application using the Flat Red look and feel

Notice the accordion UI control's title bar, reading “Scroll Bars & Sliders,” expanded with UI controls sporting the selected look and feel. This ability would be a great way to show off different look-and-feel themes to your users while their application continues to function as usual.

The next section shows the main source code of our Look N Feel Chooser application in Listing 6-2. For the sake of space, you will see only the start() method and not the entire source code. To see the entire source code listing of the Look N Feel Chooser application, please visit the book’s website to download the project.

The Look N Feel Chooser Example Application Code

By loading the Look N Feel Chooser project into the NetBeans IDE, you can see that the source code consists of five files: LookNFeelChooser.java, lnf_demo.fxml, controlStyle1.css, controlStyle2.css, flatred.css, and sky.css. The files with an extension of .ttf are TrueType fonts that are used in the FlatRed look and feel. Figure 6-4 shows what the project structure looks like in the NetBeans IDE.

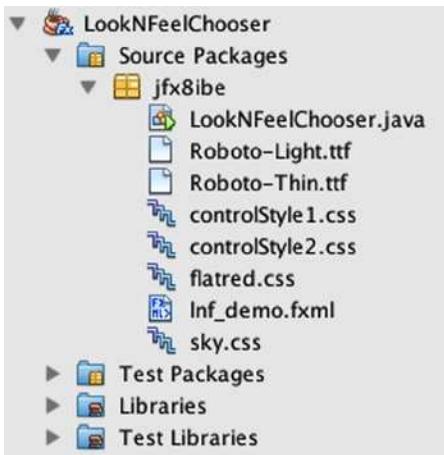


Figure 6-4. The *LookNFeelChooser* project structure in the NetBeans IDE

Looking at the project structure you'll notice the `LookNFeelChooser.java` file as the main driver application class. The `lnf_demo.fxml` file is an FXML-formatted file representing the center content, which contains the accordion UI control with other UI control elements. The FXML file was created by the Scene Builder tool, discussed later in this chapter. The rest of the files are JavaFX CSS style sheets that represent different look-and-feel themes: `controlStyle1.css`, `controlStyle2.css`, `flatred.css`, and `sky.css`.

Listing 6-2 shows the `init()` and `start()` methods of the `LookNFeelChooser.java` Application class. Remember that all JavaFX applications are first initialized via the `init()` method and then begin with the `start()` method. After you have examined Listing 6-2, we will discuss how it all works.

Listing 6-2. The *LookNFeelChooser* Project Source Code to Switch between Look and Feel (JavaFX CSS Style Sheets)

```

@Override public void init() {
    Font.loadFont(LookNFeelChooser.class
        .getResourceAsStream("Roboto-Thin.ttf"), 10)
        .getName();
    Font.loadFont(LookNFeelChooser.class
        .getResourceAsStream("Roboto-Light.ttf"), 10)
        .getName();
}

@Override public void start(Stage primaryStage) throws IOException {
    BorderPane root = new BorderPane();
    Parent content = FXMLLoader.load(getClass().getResource("lnf_demo.fxml"));
    Scene scene = new Scene(root, 650, 550, Color.WHITE);
    root.setCenter(content);

    // Menu bar
    MenuBar menuBar = new MenuBar();

```

```
// File menu
Menu fileMenu = new Menu("_File");
MenuItem exitItem = new MenuItem("Exit");
exitItem.setAccelerator(new KeyCodeCombination(KeyCode.X, KeyCombination.SHORTCUT_DOWN));
exitItem.setOnAction(ae -> Platform.exit());

fileMenu.getItems().add(exitItem);
menuBar.getMenus().add(fileMenu);

// Look and feel menu
Menu lookNFeelMenu = new Menu("_Look 'N' Feel");
lookNFeelMenu.setMnemonicParsing(true);
menuBar.getMenus().add(lookNFeelMenu);
root.setTop(menuBar);

// Look and feel selection
MenuItem caspianMenuItem = new MenuItem("Caspian");
caspianMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    setUserAgentStylesheet(STYLESHEET_CASPIAN);
});
MenuItem modenaMenuItem = new MenuItem("Modena");
modenaMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    setUserAgentStylesheet(STYLESHEET_MODENA);
});
MenuItem style1MenuItem = new MenuItem("Control Style 1");
style1MenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
        .add(getClass()
            .getResource("controlStyle1.css")
            .toExternalForm());
});
MenuItem style2MenuItem = new MenuItem("Control Style 2");
style2MenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
        .add(getClass()
            .getResource("controlStyle2.css")
            .toExternalForm());
});

MenuItem skyMenuItem = new MenuItem("Sky LnF");
skyMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
```

```

        .add(getClass()
            .getResource("sky.css")
            .toExternalForm());
    });

MenuItem flatRedMenuItem = new MenuItem("FlatRed");
flatRedMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
        .add(getClass()
            .getResource("flatred.css")
            .toExternalForm());
});
}

lookNFeelMenu.getItems()
    .addAll(caspianMenuItem,
        modenaMenuItem,
        style1MenuItem,
        style2MenuItem,
        skyMenuItem,
        flatRedMenuItem);

primaryStage.setTitle("Look N Feel Chooser");
primaryStage.setScene(scene);
primaryStage.show();
}
}

```

Note To run this example, make sure the CSS and TTF (true type font) files are located in the compiled class's 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 initially co-located with this source code example file. In NetBeans, you can select Clean and Build Project or you can copy files to your class's build area.

How It Works

The Look N Feel Chooser application is first initialized with the overloaded method `init()`, which loads the required fonts used for the FlatRed look-and-feel theme. After the `init()` method is executed the JavaFX application life-cycle will invoke the `start()` method.

In the `start()` method the code begins by creating a border pane layout, which then loads an FXML file to be placed as the center content region. FXML is an XML-based language to express JavaFX UIs. This provides a way to separate the presentation layer from the application logic layer. Typically, FXML is generated by a GUI builder tool that allows a designer to drag and drop controls and create UIs graphically. The following line uses the `FXMLLoader.load()` method to unmarshal (deserialize)

the center content. The center content pane is an AnchorPane layout containing an Accordion control holding other UI controls.

```
Parent content = FXMLLoader.load(getClass().getResource("lnd_demo.fxml"));
```

Later in this chapter you will learn about the Scene Builder tool, which allows you to create UIs to be saved as FXML-formatted files.

Continuing with our example code, you will notice the usual creation of a scene with a root node. Next, the code builds a menu bar with menu items. The first menu will be the File menu, to allow the user to exit the application. Notice that for a quick exit using a keyboard shortcut, the KeyCodeCombination instance allows the user to hit the Ctrl-X key combo. To learn about menus and keyboard shortcuts, please refer to Chapter 4.

The second menu is Look 'N' Feel, which allows the user to select predefined look-and-feel CSS style sheet files. The menu items are set to invoke handler code (lambda expressions) that is triggered based on the onAction event. The following code snippet is a menu item that is responsible for switching the application to use the Caspian look and feel:

```
MenuItem caspianMenuItem = new MenuItem("Caspian");
caspianMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(STYLESHEET_CASPIAN);
});
```

Notice that the onAction handler code containing two statements. In the JavaFX API the STYLESHEET_CASPIAN and STYLESHEET_MODENA variables are enums representing the CSS files based on their location on the classpath.

Because the rest of the code is pretty similar, I'm going to fast-forward to the last look-and-feel menu option, which switches to the Flat Red look and feel. The other menu items are essentially identical and invoke the same methods to clear previously loaded CSS style sheet files. For the Flat Red look and feel I used the getStylesheets().add() method to style only the current scene-level nodes. The Flat Red look and feel styles a subset of UI controls that includes sliders, scroll bars, buttons, and progress controls. The code snippet shown next sets up a menu item to be selected that will skin the application with the Flat Red look and feel. After the style is loaded that clears previously loaded style sheets, it loads the default look and feel (null) and then loads the flatred.css file, which will style the given scene.

```
MenuItem flatRedMenuItem = new MenuItem("FlatRed");
flatRedMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
        .add(getClass()
            .getResource("flatred.css")
            .toExternalForm());
});
```

JavaFX CSS

Now that you know how to load CSS style sheet files, let's discuss JavaFX CSS selectors and styling properties (rules). Similar to the way HTML5 uses CSS style sheets, there are selectors or style classes associated with Node objects on the Scene graph. All JavaFX Scene graph nodes have a `setId()`, a `getStyleClass().add()`, and a `setStyle()` method, respectively, to apply styling properties that could potentially change the node's background color, border, stroke, and so on.

Before we discuss selectors, I would like to refer you to the *JavaFX CSS Reference Guide* at <http://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html>. This invaluable reference guide will be very handy throughout this book and beyond.

What Are Selectors?

Similar to the W3C CSS (World Wide Web Consortium) standards for styling HTML elements, JavaFX CSS has the concept of *selectors*. Selectors are basically tags to help locate JavaFX nodes on the Scene graph to be styled using CSS style definitions. The two kinds of selector types are `id` and `class`. An `id` selector is a unique string name that is set on a scene node.

A `class` selector is also a string name that can be added as a tag to any JavaFX node. I should also point out that a `class` selector has no relation to the concepts of C++ or Java classes. `Class` selectors allow any number of nodes to contain the same `class` string name for the ability to style nodes with one CSS style definition. In the next sections you will learn more about how to define CSS selector types, and we will follow-up with an example.

CSS id Type Selectors

The `id` type selector is a unique string name assigned to a node. This means that no other node's `id` can be the same. When using `id` type selectors, you will invoke the `setId(String ID)` method on a JavaFX node object. For example, to target a `Button` instance whose `id` is `my-button`, you would invoke the `setId("my-button")` method. To style the button with an `id` of `my-button`, you would create a CSS style definition block declared with an `id` selector `#my-button`, as shown here:

```
#my-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;  
    -fx-font: bold italic 20pt "LucidaBrightDemiBold";  
}
```

This CSS styling block will be applied to a button with the unique `id` of `my-button`. Thus no other node will be allowed to contain an `id` of `my-button`. You will also notice that when we use `id` selectors in styling blocks, the CSS selector name is prefixed with the `#` symbol, and when setting the `id` in Java code the `#` symbol is not used.

CSS class Type Selectors

When using `class` type selectors you will invoke the `getStyleClass().add(String styleClass)` method to add a selector to a node. The method allows you to have multiple style classes for styling a node. Since the `getStyleClass()` method returns an `ObservableList` you can add and remove style classes in an ad hoc fashion to update its appearance dynamically. For example, let's target two buttons whose style classes (`ObservableList`) contain a `num-button` via the `getStyleClass().add("num-button")` method. The following is a CSS style definition block declared with a class selector `.num-button`:

```
.num-button {  
    -fx-background-color: white, rgb(189,218,230), white;  
    -fx-background-radius: 50%;  
    -fx-background-insets: 0, 1, 2;  
    -fx-font-family: "Helvetica";  
    -fx-text-fill: black;  
    -fx-font-size: 20px;  
}
```

This CSS styling block will be applied to buttons with a style class `num-button`. You will notice that when using `class` selectors, the CSS selector name is prefixed with a dot (.), and when adding the selector in Java code the (.) symbol is not present.

Selector Patterns

Up until now you've seen only simple selectors; however, selectors can have patterns for the ability to traverse the Scene graph's hierarchy of nodes from the root node to the child nodes. Of course, a full discussion of selector patterns is beyond the scope of this book, but I will briefly mention common selector patterns and introduce pseudo-classes.

Common Selector Patterns

Often you will want to style many nodes based on common selector patterns. A typical selector pattern you would want to perform would be to style child nodes whose parent is of a certain type. Another pattern would be to style two different kinds of nodes with a common property. The following example shows selector patterns for both of these use cases:

```
/* style all buttons who's parent is an HBox */  
.hbox > .button { -fx-text-fill: black; }  
  
/* style all labels and text nodes */  
.label, .text { -fx-font-size: 20px; }
```

As you can see, the selector pattern (`.hbox > .button`) styles `Button` nodes that are descendants of an `HBox` and is pretty straightforward. The greater-than symbol between the two selectors lets the system know which nodes to style. Also, as described in the *JavaFX CSS Reference Guide*, the selectors for UI controls have a naming convention. In this convention, selectors for JavaFX nodes are all lowercase letters, and if a control has more than one word they are separated by hyphens.

For instance, a `GridPane`'s class selector would be `.grid-pane`. Keep in mind that not all nodes have named class selectors by default, so please refer to the JavaFX CSS Reference guide.

The second use case has to do with setting a common property that is shared by different types of nodes. Our second example is a selector pattern that styles all `Label` nodes and `Text` nodes to have the same text font size. The comma denotes a list of selectors to style. In this scenario the font size will be styled to be 20 points.

Pseudo-Class Selectors

Pseudo-class selectors are used to style nodes that have different states. An example would be a `Button` node. A button has the following states: hover, selected, focused, and so on. To specify selectors with pseudo-classes you must append a colon and the state (type) to the main selector name. The following code snippet shows two selector patterns for a button that has a class selector `num-button`:

```
.num-button {  
    -fx-background-color: white,  
        bluish-gray,  
        white;  
    -fx-background-radius: 50%;  
    -fx-background-insets: 0, 1, 2;  
    -fx-font-family: "Helvetica";  
    -fx-text-fill: black;  
}  
.num-button:hover {  
    -fx-background-color: black,  
        white,  
        black;  
    -fx-text-fill: white;  
}
```

In this CSS code, the first (`.num-button`) selector merely styles a button in its normal state. However, the second selector with the appended colon and state (`:hover`) will change a subset of properties, altering them slightly. In this scenario when a mouse cursor moves over (hovers) a button with the style class `num-button:hover`, the color of the button and the text's fill color will be reversed. Figure 6-5 shows the pseudo-selector in action when the mouse is hovering over the number pad on button 3.

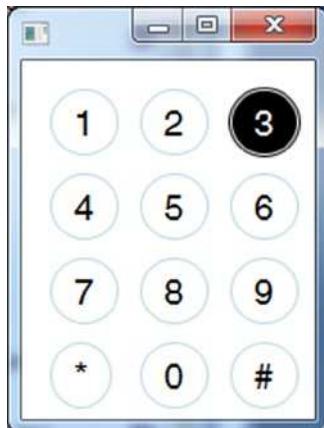


Figure 6-5. A selector with a pseudo-class of `hover` is used when the mouse cursor is on top of any number pad button

A Selector Styling Example

To demonstrate selectors using id and class type selectors, I've created an application resembling a smart phone's number pad, as shown in Figure 6-6.



Figure 6-6. An application to mimic a number pad from a smart phone

In Figure 6-6 you will notice a 4×3 grid of round buttons for a number pad and a green rectangular button beneath it to make a phone call. Listing 6-3 is the main `start()` method containing the source code. Listing 6-4 then shows the contents of the CSS file `mobile_buttons.css`.

Listing 6-3. The JavaFX Source Code for Our Number Pad Application

```

@Override
public void start(Stage primaryStage) {

    BorderPane root = new BorderPane();
    Scene scene = new Scene(root, 180, 250);
    scene.getStylesheets()
        .add(getClass().getResource("mobile_buttons.css")
            .toExternalForm());
    String[] keys = {"1", "2", "3",
                    "4", "5", "6",
                    "7", "8", "9",
                    "*", "0", "#"};
    GridPane numPad = new GridPane();
    numPad.getStyleClass().add("num-pad");

    for (int i=0; i < 12; i++) {
        Button button = new Button(keys[i]);
        button.getStyleClass().add("num-button");
        numPad.add(button, i % 3, (int) Math.ceil(i/3) );
    }
    // Call button
    Button call = new Button("Call");
    call.setId("call-button");
    call.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
    numPad.add(call, 0, 4);
    GridPane.setColumnSpan(call, 3);

    GridPane.setHgrow(call, Priority.ALWAYS);
    root.setCenter(numPad);
    primaryStage.setScene(scene);
    primaryStage.show();
}

```

Listing 6-4. The Contents of the CSS File `mobile_buttons.css`

```

.root {
    -fx-background-color: white;
    -fx-font-size: 20px;
    bright-green: rgb(59,223, 86);
    bluish-gray: rgb(189,218,230);
}

.num-pad {
    -fx-padding: 15px, 15px, 15px, 15px;
    -fx-hgap: 10px;
    -fx-vgap: 8px;
}

.num-button {
    -fx-background-color: white,
        bluish-gray,
        white;
}

```

```

    -fx-background-radius: 50%;
    -fx-background-insets: 0, 1, 2;
    -fx-font-family: "Helvetica";
    -fx-text-fill: black;
}
.num-button:hover {
    -fx-background-color: black,
        white,
        black;
    -fx-text-fill: white;
}
#call-button {
    -fx-background-color: white, bright-green;
    -fx-background-radius: 2;
    -fx-background-insets: 0, 1;
    -fx-font-family: "Helvetica";
    -fx-text-fill: white;
}
#call-button:hover {
    -fx-background-color: bright-green, white;
    -fx-background-radius: 2;
    -fx-background-insets: 0, 1;
    -fx-font-family: "Helvetica";
    -fx-text-fill: bright-green;
}

```

How It Works

The code starts by creating a scene that has a BorderPane as a root node. After the scene is created, the code loads the CSS style sheet file `mobile_buttons.css` to style the current scene's nodes. Next, the code simply creates a grid using the GridPane class and generates 12 buttons to be placed in each cell. Notice in the for loop that each button is set with the style class named num-button via the `getStyleClass().add()` method.

Last, the green call button will be added to the last row of the grid pane. Because the call button is unique, its id selector is set with `call-button`, and it will be styled using the id selector, which means the selector named inside the CSS file will be prefixed with the # symbol. To see some amazing button styling, head over to the FXexperience.com's blog entry on styling buttons at <http://fxexperience.com/2011/12/styling-fx-buttons-with-css> by Jasper Potts (Developer Experience Architect on Client Java at Oracle).

How to Define -fx- Based Styling Properties (Rules)

I'm sure by now you have noticed the many name-value pairs (properties) inside styling definition blocks having the prefix `-fx-`. These properties, often called rules, can be defined to have values that can set a region's border width, background fill colors, and so on. In this section you will learn how to style a node using selector styling blocks and the ability to override properties by using inline styling.

Styling a Node with a Selector Style Definition Block

A selector style definition block always begins with a selector name prefixed with a . or # symbol. The symbol determines the selector type as mentioned earlier. A selector will start a block with an open curly brace, followed by property or rule definitions. Each JavaFX theme property or rule will be prefixed with -fx- and its appropriate property name. A property name and value are separated by a : symbol, and the pair ends with a semicolon. To finish the styling block, you simply end the block with a closing curly brace. The syntax for a selector style definition block looks like this:

```
. or #<selector-name> <pattern>{
    -fx-<some-property> : <some-value>;
}
```

One last thing to mention is the ability to add comments to CSS styling definitions. To add comments you would use an opening slash asterisk /* and a closing asterisk slash */, the same as adding comments in C/C++ and Java. Shown next is an example of using comments in a selector style block definition:

```
.num-button {
    -fx-background-color: white, rgb(189,218,230), white;
/*
    This is a comment.
    -fx-background-radius: 50%;
*/
    -fx-background-insets: 0, 1, 2;
    -fx-font-family: "Helvetica";
    -fx-text-fill: black;
    -fx-font-size: 20px;
}
```

Styling a Node by Inlining JavaFX CSS Styling Properties

While CSS selector blocks are the recommended way to style your JavaFX nodes, there might be situations where you will want to override styling properties. For example you may want to enlarge a button and change its text color temporarily as a mouse cursor hovers (OnMouseEntered) over the specified button. When the mouse cursor is not hovering (OnMouseExited) the button's inline style is removed which then reverts the styling to the parent class or id selector styling blocks. To override a node's styling property set from its ancestor (parent) id or class selector, you can invoke the node's method `setStyle()`. To implement the prior example just mentioned the code snippet below implements a button with handler code that responds to OnMouseEntered and OnMouseExited events which toggles the button's text size and color.

```
Button button = new Button("Press Me");
button.getStyleClass().add("my-default-style");
button.setOnMouseEntered(actionEvent -> button.setStyle("-fx-font-size: 30px; -fx-text-fill: green;"));
button.setOnMouseExited(actionEvent -> button.setStyle(""));
```

Styling Properties (rules) Limitations

Because all graph nodes extend from the `Node` class, derived classes will be able to inherit styling properties from their ancestors. Knowing the inheritance hierarchy of node types is very important because the type of node will help determine the types of styling properties you can control. For instance, a `Rectangle` class extends from `Shape`, which extends from `Node`. Having said this, you will later see that some properties that you assumed would be on a node don't exist. So how do you know what properties exist? The solution is to refer to the *JavaFX CSS Reference Guide*. Again, based on the type of node there are limitations to the styles you are able to set.

Obeying the JavaFX CSS Rules

Did you know that styling properties (rules) can be overridden? To override properties you must learn about the order of precedence when rules are defined. Figure 6-7 depicts a typical Scene graph with a root (parent) node with child nodes. A root node such as a `BorderPane` layout node will contain a class selector of `root`, which is meant for *top-level* properties. These are basically properties common to many nodes, such as font size for text or background color. This is a great feature for theming, because many nodes that share a property will inherit that change. As mentioned earlier, some nodes will contain default selectors; for example, JavaFX Button instances will have a `.button` class selector.

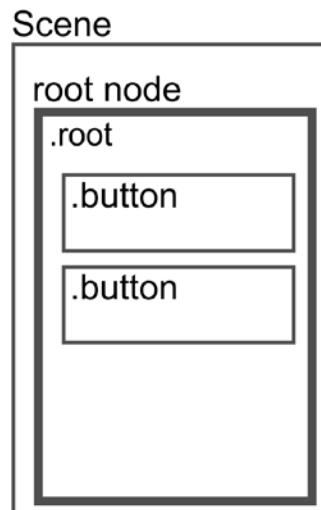


Figure 6-7. A scene with a root node which has buttons as child nodes

In Figure 6-7 the class selector root could have a property that is common across many nodes:

```
.root {  
    -fx-font-size: 12px;  
}  
  
.button {  
    -fx-font-size: 20px;  
}
```

When you want to style a button's text font size you have an opportunity to override the root's styling definition. In order to override the `-fx-font-size` property, the button style class definition block will override the 12 point font from the parent with a 20 point font.

Believe it or not, there is an additional way to override the style definition block of the class selector `.button` in addition to the method just shown. Similar to HTML5 CSS, each element has a `style` attribute. JavaFX's graph nodes also have a `style` attribute (property). Following is an example of how to set the `style` property programmatically in Java code:

```
Button button = new Button("press it!");  
button.setStyle("-fx-font-size: 30px;");
```

This code will override any `root-`, `id-`, or `class-level` selector style definition block. Basically, the invocation to the `setStyle()` method allows you to add any number of property-value pairs as long they are separated by the semicolon.

Scene Builder

Up to this point you've learned how to code and style UI elements programmatically. You are probably wondering if there is a better way to build UIs. Well, I am happy to inform you that the JavaFX team has built the free tool Scene Builder 2.0, which allows you to build UIs graphically. The Scene Builder tool has also been open-sourced and will begin to show up in your favorite IDE. This section provides a quick glimpse of the Scene Builder tool. To download and try Scene Builder, please visit Oracle's JavaFX tools site, at <http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html>.

Launching Scene Builder

After downloading and installing it, you will launch the Scene Builder tool. Figure 6-8 shows the Scene Builder tool.

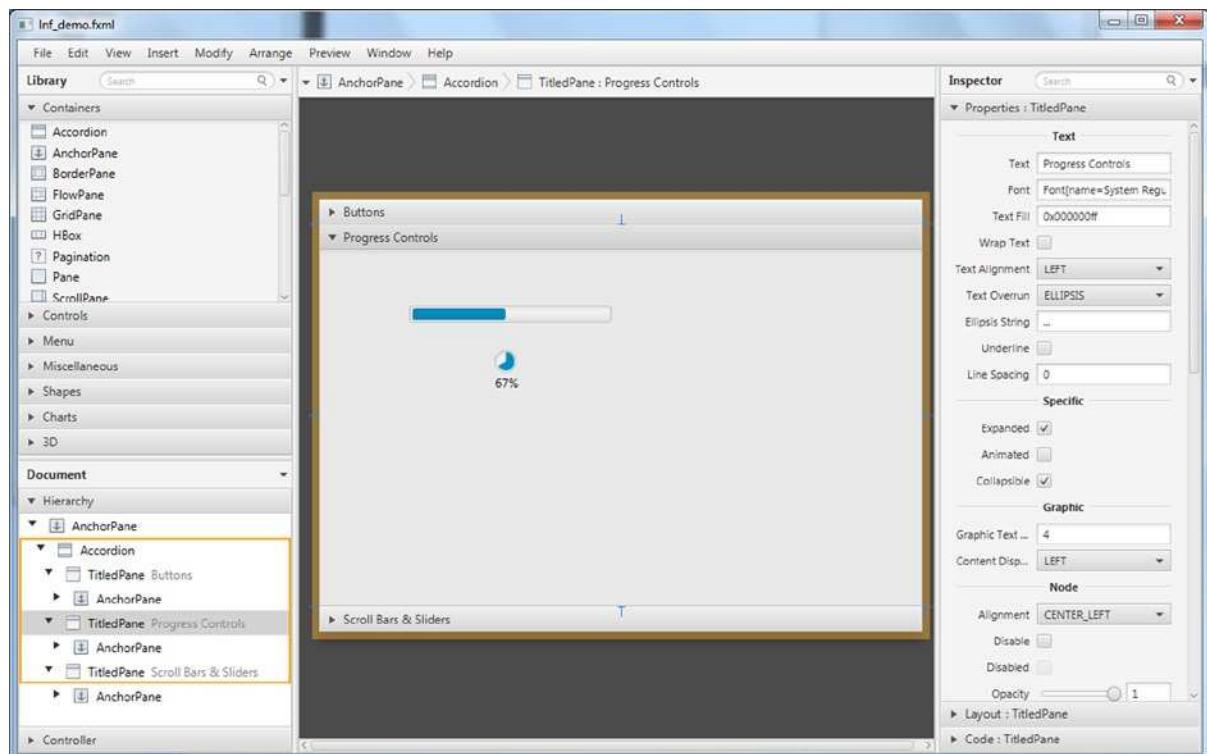


Figure 6-8. The Scene Builder tool

Here you can see the earlier Look N Feel Chooser project example with an AnchorPane layout node and an Accordion UI control in the center canvas area. On the left you see a split divider with UI elements that can be dragged onto the center canvas area.

Also on the left beneath the UI elements is the Document section. It has two sub-sections, Hierarchy and Controller. The Hierarchy panel allows you to view the current node hierarchy (better known as the document model). Finally, below the Hierarchy is the Controller panel. The Controller lets you create action code or presentation logic in a Java class. This allows a developer to have separate controller code bind to the UI elements.

On the right you see the Properties, Layout, and Code panels. The Properties panel allows you to style UI elements using JavaFX CSS properties, the Layout panel allows you to specify dimensions, transforms, and padding of a node's properties, and the Code panel allows you to map event methods to UI elements such as OnMousePressed.

FXML

The Scene Builder allows you to save UI projects as a single file in an XML format called FXML (FX markup language). The FXML file is saved with an extension of .fxml. FXML is a language that represents a JavaFX-based UI. The FXML language is beyond the scope of this book; however I will briefly describe how to load the files to be displayed.

Loading FXML into a Scene

FXML provides interesting advantages over the programmatic approach. One of the main advantages is the ability to decouple the GUI (view) layer from the code logic (controller) layer. As you learned in our discussion of UI patterns in Chapter 4, it is a good idea to separate GUI code from action code. Shown here is code to load an FXML file:

```
BorderPane root = new BorderPane();
Parent content = FXMLLoader.load(getClass().getResource("lnd_demo.fxml"));
root.setCenter(content);
```

As you can see, above the root pane is a BorderPane, and we use the static method `load` on the `javafx.fxml.FXMLLoader` class. The `load()` method will load (deserialize) the FXML file that was created by the Scene Builder tool.

Custom Controls

“Use the force, Luke.”

—Obi wan Kenobi

With all of your new-found abilities, there is yet another powerful API you will want to harness. Imagine a customer whose requirements call for specialized UI controls such as gauges, knobs, or light indicators. Surely you could attempt to use the existing UI controls to be styled, but it is rather difficult, if not impossible. So, what is a young padawan to do? (Now, say the quote). Welcome to the world of custom controls (the Controls API)!

As an example of creating a custom control, in this section you will learn how to create a light-emitting diode (LED) control. We'll begin with a description of the LED control. Then you will see how the code is structured as we look at the classes involved in the creation of custom controls. After that, you will get a chance to look at the code before we walk through it to discuss the code details.

The LED Custom Control

The light-emitting diode (LED) is something that you will find in many electronic devices to indicate status (such as on or off). So why not use an LED as a custom control in your code as a visual indicator? Typically, an LED contains two wires and a body made from clear plastic, as shown in Figure 6-9.



Figure 6-9. A typical light-emitting diode

Often the LED is mounted in a socket. To create a custom JavaFX control to emulate a physical object you first have to decide what materials and components you will portray.

In the case of the LED custom JavaFX control, we will approximate this appearance:

- Metal socket
- Plastic body
- Light effect on top of the plastic body due to the curved surface

One thing that Java developers often don't like, but which is really helpful for designing a custom control, is a drawing program—to be precise, a vector drawing program. When the appearance of your control is important, it makes sense to use a tool that is good for visualizing things; a drawing program, as used to create Figure 6-10. So the first thing you should do is to create a vector drawing of the control you plan to create.

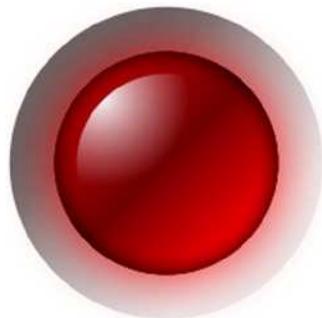


Figure 6-10. A vector drawing of the LED control

You can see the metal socket around the red plastic body and the white highlight. The red body contains an inner- and outer-shadow to create a more realistic look. If we look at the parts of the vector drawing, we will find three circles filled with gradients, as shown in Figure 6-11.

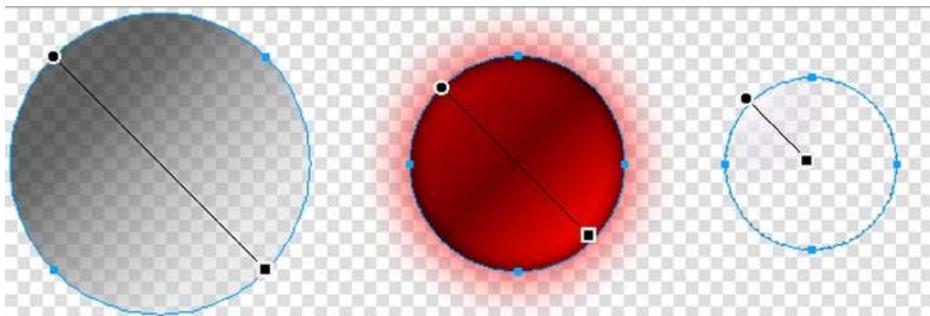


Figure 6-11. The three parts of the vector LED

You start coding in the drawing program because here you define the size, colors, gradients and positions of your control. The big advantage of using a drawing program is the direct visual feedback that you get by changing parameters like color, size, and position.

In JavaFX there's not only one way of creating a custom control but many. Here is a list of valid approaches:

- Customize the CSS of an existing control
- Extend an existing control
- Extend the Region node
- Create a control, a skin, and a CSS file
- Use the Canvas node

In this book we will focus on the approach that extends a Region and uses CSS. To get an idea of how to create the same LED control by using the Canvas node or by using a separate Control class, Skin. class, and a CSS file, look at the GitHub repository at <http://github.com/HanSolo/JFX8CustomControls>.

Structure of the LED Custom Control Example Code

Usually I structure the code in my controls in the following way:

- Constructor
- Initialization:
 - An init() method defines the initial size.
 - An initGraphics() method sets up the Scene graph of the control.
 - A registerListeners() method hooks up listeners to properties.
- A Methods block contains the get-, set- and property-methods.
- A Resizing block contains methods to resize and redraw (if needed) the control.

The Properties of the LED Control

The LED control will contain the logic of the control (its properties) and the visualization code. Because the LED is a very simple control we do not need many properties. For this example we will use five properties:

- On (Boolean property for current state)
- Blinking (Boolean property to switch on/off blinking)
- Interval (long property to define the blink interval)
- FrameVisible (Boolean property to switch on/off the metal socket)
- LedColor (object property of type Color to define the color of the LED)

We will write code for get-, set- and property methods for these properties as shown in Listing 6-5.

Listing 6-5. Property Manipulation Code

```
public final boolean isOn() {
    return null == on ? false : on.get();
}
public final void setOn(final boolean ON) {
    onProperty().set(ON);
}
public final BooleanProperty onProperty() {
    if (null == on) {
        on = new BooleanPropertyBase(false) {
            @Override protected void invalidated() {
                pseudoClassStateChanged(ON_PSEUDO_CLASS, get());
            }
            @Override public Object getBean() { return this; }
            @Override public String getName() { return "on"; }
        };
    }
    return on;
}

public final boolean isBlinking() {
    return null == blinking ? false : blinking.get();
}
public final void setBlinking(final boolean BLINKING) {
    blinkingProperty.set(BLINKING);
}
public final BooleanProperty blinkingProperty() {
    if (null == blinking) {
        blinking = new BooleanPropertyBase() {
            @Override public void set(final boolean BLINKING) {
                super.set(BLINKING);
                if (BLINKING) {
                    timer.start();
                } else {

```

```

        timer.stop();
        setOn(false);
    }
}
@Override public Object getBean() {
    return Led.this;
}
@Override public String getName() {
    return "blinking";
}
};

}

return blinking;
}

public final long getInterval() {
    return null == interval ? 500_000_000 : interval.get();
}
public final void setInterval(final long INTERVAL) {
    intervalProperty().set(INTERVAL);
}
public final LongProperty intervalProperty() {
    if (null == interval) {
        interval = new LongPropertyBase() {
            @Override public void set(final long INTERVAL) {
                super.set(clamp(SHORTEST_INTERVAL, LONGEST_INTERVAL, INTERVAL));
            }
            @Override public Object getBean() {
                return Led.this;
            }
            @Override public String getName() {
                return "interval";
            }
        };
    }
    return interval;
}

public final boolean isFrameVisible() {
    return null == frameVisible ? true : frameVisible.get();
}
public final void setFrameVisible(final boolean FRAME_VISIBLE) {
    frameVisibleProperty().set(FRAME_VISIBLE);
}
public final BooleanProperty frameVisibleProperty() {
    if (null == frameVisible) {
        frameVisible = new SimpleBooleanProperty(this, "frameVisible", _frameVisible);
    }
    return frameVisible;
}
}

```

```

public final Color getLedColor() {
    return null == ledColor ? Color.RED : ledColor.get();
}
public final void setLedColor(final Color LED_COLOR) {
    ledColorProperty().set(LED_COLOR);
}
public final ObjectProperty<Color> ledColorProperty() {
    if (null == ledColor) {
        ledColor = new SimpleObjectProperty<>(this, "ledColor", Color.RED);
    }
    return ledColor;
}

```

The Initialization Code of the LED Control

The first thing we have to do in the constructor of the LED control is load the corresponding CSS file and add the main style class as follows:

```

public Led() {
    getStylesheets().add(getClass().getResource("led.css").toExternalForm());
    getStyleClass().add("led");
}

```

Then we simply initialize the AnimationTimer that we will use to make the LED control blink:

```

lastTimerCall = System.nanoTime();
timer       = new AnimationTimer() {
    @Override public void handle(final long NOW) {
        if (NOW > lastTimerCall + getInterval()) {
            setOn(!isOn());
            lastTimerCall = NOW;
        }
    }
};

```

The last thing we have to do in the constructor is to call these methods to initialize the size of the control, initialize the Scene graph, and register some listeners:

```

init();
initGraphics();
registerListeners();

```

To make sure that our LED control will be correctly sized during initialization we set the minimum, preferred, and maximum sizes in the `init()` method, which looks as follows:

```

private void init() {
    if (Double.compare(getWidth(), 0) <= 0 || Double.compare(getHeight(), 0) <= 0 ||
        Double.compare(getPrefWidth(), 0) <= 0 || Double.compare(getPrefHeight(), 0) <= 0) {
        setPreferredSize(PREFERRED_SIZE, PREFERRED_SIZE);
    }
}

```

```

if (Double.compare(getMinWidth(), 0) <= 0 || Double.compare(getMinHeight(), 0) <= 0) {
    setMinSize(MINIMUM_SIZE, MINIMUM_SIZE);
}
if (Double.compare(getMaxWidth(), 0) <= 0 || Double.compare(getMaxHeight(), 0) <= 0) {
    setMaxSize(MAXIMUM_SIZE, MAXIMUM_SIZE);
}
}
}

```

Visualization Code

The Region node is a lightweight JavaFX container that can contain other nodes and be styled by CSS. By extending the Region node our custom control will contain the logic of the control and also the visualization code. In the `initGraphics()` method shown in Listing 6-6, the code sets up the Scene graph for the control by creating the nodes that are needed and applying the appropriate CSS styles to them.

Listing 6-6. Setting Up the LED Control Scene Graph

```

private void initGraphics() {
    // Create the node for the metal socket
    frame = new Region();
    frame.getStyleClass().setAll("frame");
    frame.setOpacity(isFrameVisible() ? 1 : 0);

    // Create the node for the main LED plastic body
    led = new Region();
    led.getStyleClass().setAll("main");
    led.setStyle("-led-color: " + (getLedColor()).toString().replace("0x", "#") + ";");

    // Create the inner shadow effect for the main LED body
    innerShadow = new InnerShadow(BlurType.TWO_PASS_BOX,
        Color.rgb(0, 0, 0, 0.65),
        8, 0d, 0d, 0d);

    // Create the drop shadow effect for the main LED body (the glow effect)
    glow = new DropShadow(BlurType.TWO_PASS_BOX,
        getLedColor(),
        20, 0d, 0d, 0d);
    glow.setInput(innerShadow);

    // Create the node for the highlight effect on the main LED body
    highlight = new Region();
    highlight.getStyleClass().setAll("highlight");

    // Add all nodes to the Scene graph of this control
    getChildren().addAll(frame, led, highlight);
}
}

```

So we have created each node that we need for our LED control and applied the appropriate style from the CSS file.

The LED Control CSS File

Each node that is created in the `initGraphics()` method gets its own CSS style class, which can be found in the `led.css` file. It looks like this:

```
/* The main led style class where the -led-color variable is defined */
.led {
    -led-color : red;
    -frame-color: linear-gradient(from 14% 14% to 84% 84%,
                                    rgba(20, 20, 20, 0.64706) 0%,
                                    rgba(20, 20, 20, 0.64706) 15%,
                                    rgba(41, 41, 41, 0.64706) 26%,
                                    rgba(200, 200, 200, 0.40631) 85%,
                                    rgba(200, 200, 200, 0.3451) 100%);
}

/* The .frame sub-class, which defines the fill for the metal socket */
.led .frame {
    -fx-background-color : -frame-color;
    -fx-background-radius: 1024;
}

/* The .main sub-class, which defines the fill for the LED plastic body when it's off */
.led .main {
    -fx-background-color : linear-gradient(from 15% 15% to 83% 83%,
                                            derive(-led-color, -80%) 0%,
                                            derive(-led-color, -87%) 49%,
                                            derive(-led-color, -80) 100%);
    -fx-background-radius: 1024;
}

/* The .main sub class with pseudo-class :on that defines the fill for the LED plastic
   body when it's on
*/
.led:on .main {
    -fx-background-color: linear-gradient(from 15% 15% to 83% 83%,
                                           derive(-led-color, -23%) 0%,
                                           derive(-led-color, -50%) 49%,
                                           -led-color 100%);
}

/* The .highlight sub-class that defines the fill of the highlight effect */
.led .highlight {
    -fx-background-color : radial-gradient(center 15% 15%, radius 50%,
                                            white 0%,
                                            transparent 100%);
    -fx-background-radius: 1024;
}
```

Because in CSS we can use percentage to define positions in gradients, we don't have to take care about the real size of the control to calculate the start and stop positions of the gradients. That is a huge advantage compared to Java Swing, where we had to calculate all these values every time the size of the control changed. So all these calculations will be done automatically by JavaFX. That reduces our resizing code a lot as you will see.

Resizing the LED Control

In JavaFX the size of the layout container determines the size of its children, so we have to ensure that our control is resized by its layout container. That means if we put our LED control in a StackPane (which resizes its children relative to its own size), the LED will be sized the same as the StackPane. Therefore we hook up listeners to the `widthProperty()` and `heightProperty()` of our control in the `registerListeners()` method.

```
private void registerListeners() {
    widthProperty().addListener(observable -> resize());
    heightProperty().addListener(observable -> resize());
    frameVisibleProperty().addListener(observable ->
        frame.setOpacity(isFrameVisible() ? 1 : 0));
    onProperty().addListener(observable -> led.setEffect(isOn() ? glow : innerShadow));
    ledColorProperty().addListener(observable -> {
        led.setStyle("-led-color: " + (getLedColor()).toString().replace("0x", "#") + ";");
        resize();
    });
}
```

In this method we hooked up listeners to all properties that might have an effect on either visualization or size of the LED control. When the listeners for the width and height are triggered, they will call the `resize()` method, and that will take care of sizing all nodes in our control.

```
private void resize() {
    double width = getWidth();
    double height = getHeight();

    // Determine the smallest dimension of our control
    size = width < height ? width : height;

    // Resize only if the current width and height > 0
    if (width > 0 && height > 0) {

        // Center the control if the parents dimensions are not square
        if (width > height) {
            setTranslateX(0.5 * (getWidth() - size));
        } else if (getHeight() > getWidth()) {
            setTranslateY(0.5 * (getHeight() - size));
        }

        // Set the radius of the inner shadow to 7% of the current size
        innerShadow.setRadius(0.07 * size);
    }
}
```

```

// Set the radius of the drop shadow to 36% of the current size and set the glow color
glow.setRadius(0.36 * size);
glow.setColor(getLedColor());

// Make the metal frame as big as the current size
frame.setPrefSize(size, size);

// Set the size of the LED plastic body to 72% of the current size and center it
led.setPrefSize(0.72 * size, 0.72 * size);
led.relocate(0.14 * size, 0.14 * size);

// Set the right effect dependent on the current state of the LED
led.setEffect(isOn() ? glow : innerShadow);

// Set the size of the highlight to 58% of the current size and center it
highlight.setPrefSize(0.58 * size, 0.58 * size);
highlight.relocate(0.21 * size, 0.21 * size);
}
}

```

As you can see, in the `resize()` method the only thing we have to do for each node is the calculation of its size and the relocation of the node. So first we calculate the minimum dimension of our LED (because it's square, we take the width if it's less than the height or vice versa). In addition we make sure that the resizing will be done only if the current size is larger than 0.

How It Works

Now that we have all things in place we can simply use our control like any other control, as shown here:

```

@Override public void start(Stage stage) {
    Led control = new Led();

    StackPane pane = new StackPane();
    pane.getChildren().add(control);

    Scene scene = new Scene(pane);

    stage.setTitle("JavaFX Led Control");
    stage.setScene(scene);
    stage.show();

    control.setBlinking(true);
}

```

Figure 6-12 shows the result.

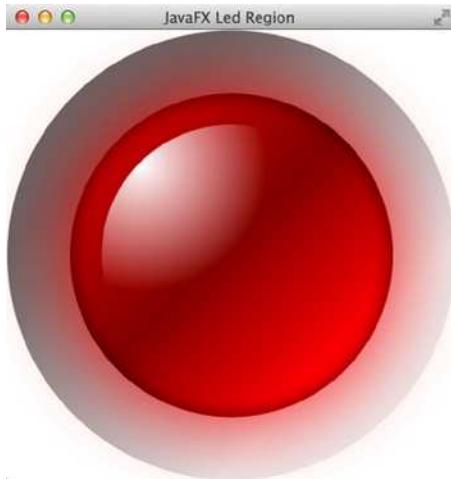


Figure 6-12. The Region-based custom LED Control

Other Ways to Create a Custom Control

As I mentioned earlier, there are different ways to create a custom control in JavaFX. The “extend Region” approach that was shown here is the most common one. The drawback that comes with this approach is that we mixed up the model/controller with the view, which is acceptable for small controls like our LED, but it might be a bad solution for a bigger control that relies on a data model or controls that are part of a controls library.

For those more complex controls, you should use a custom `Control.class` in combination with a custom `Skin.class` and your CSS file. The main difference here is that the properties with their get, set, and property methods will be placed in the `Control.class` and the visualization code will be placed in the `Skin.class`. This approach provides a proper separation between the controller logic and the view logic. The last approach to creating a custom control that I would like to mention is to use the `Canvas` node. The `Canvas` node represents a single node that in principle behaves like an image which you can draw on. The API that is used to draw on the `Canvas` node is taken from the HTML5 Canvas, which means you won’t have JavaFX nodes but you will simply draw on the `Canvas` node. (That is, you’ll be working in *immediate mode* rather than *retained mode*). You have to take care of clearing and redrawing the `Canvas` node surface. This approach might be useful for controls that contain very complex drawings to calculate and draw, as you would only need to do so once.

Summary

In this chapter you learned how to style your application with custom CSS files by using the `setUserAgentStylesheet(String URL)` and `getStylesheets().add(String URL)` methods. Then you saw how CSS is used in JavaFX by using the default CSS selectors or creating your own. You learned about the `id` type selectors, `class` type selectors, and CSS pseudo-classes. In addition, you learned how to use the selectors in different selector patterns. Next you took a short look at the Scene Builder and how to load FXML into a scene. Finally, you learned about custom controls in JavaFX and how create them by extending a `Region` node.

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 rich Internet applications (RIAs). One of the main benefits of the Media API is that it can distribute cross-platform 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 of 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 as audio and video take center stage.

You will be learning the following JavaFX media APIs:

- `javafx.scene.media.Media`
- `javafx.scene.media.MediaPlayer`
- `javafx.scene.media.MediaStatus`
- `javafx.scene.media.MediaView`

Media Events

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. In this section you will learn how to interact with media events.

With event-based programming in mind, you will discover nonblocking or callback behaviors when invoking media functions. Instead of tying code directly to a button via an `EventHandler`, you will be implementing code that will respond to the triggering of the media player's `OnXXXX` events, where `XXXX` is the event name.

When responding to media events, you will implement `java.lang.Runnable` functional interfaces (lambda expressions). These functional interfaces are callbacks that are lazily evaluated or invoked at a later time upon a triggered event. For instance, when playing media content you would create a lambda expression (closure) to be set on the `OnReady` event. For an example of playing media based on an event-driven approach, see the following code block:

```
Media media = new Media(url);
MediaPlayer mediaPlayer = new MediaPlayer(media);
Runnable playMusic = () -> mediaPlayer.play();
mediaPlayer.setOnReady(playMusic);
```

As you can see, the `playMusic` variable is assigned to a lambda expression (`Runnable`) to be passed into the media player's `setOnReady()` method. When the `OnReady` event has been encountered, the `playMusic` `Runnable` closure code would be invoked. Although I have not yet discussed how to set up the `Media` and `MediaPlayer` class instances, I wanted to first get you acquainted with these core concepts before moving further, because these concepts are used throughout this chapter.

Shown in Table 7-1 are all the possible media events that are raised to allow the developer to attach `Runnables` (or `EventHandlers`). All of the following are `Runnable` functional interfaces except for the `OnMarker` event.

Table 7-1. Media and MediaPlayer Events

Class	Set On Method	On Method Property Method	Description
Media	<code>setOnError()</code>	<code>onErrorProperty()</code>	When an error occurs
MediaPlayer	<code>setOnEndOfMedia()</code>	<code>onEndOfMediaProperty()</code>	When the end of the media play is reached
MediaPlayer	<code>setOnError()</code>	<code>onErrorProperty()</code>	When an error occurs
MediaPlayer	<code>setOnHalted()</code>	<code>onHaltedProperty()</code>	When the media status changes to HALTED
MediaPlayer	<code>setOnMarker()</code>	<code>onMarkerProperty()</code>	When a Marker event is triggered
MediaPlayer	<code>setOnPaused()</code>	<code>onPausedProperty()</code>	When a Paused event occurs
MediaPlayer	<code>setOnPlaying()</code>	<code>onPlayingProperty()</code>	When the media is currently playing
MediaPlayer	<code>setOnReady()</code>	<code>onReadyProperty()</code>	When the media player is in Ready state
MediaPlayer	<code>setOnRepeat()</code>	<code>onRepeatProperty()</code>	When the Repeat property is set
MediaPlayer	<code>setOnStalled()</code>	<code>onStalledProperty()</code>	When the media player is stalled
MediaPlayer	<code>setOnStopped()</code>	<code>onStoppedProperty()</code>	When the media player has stopped
MediaView	<code>setOnError()</code>	<code>onErrorProperty()</code>	When an error occurs in Media View

You will notice in Table 7-1 that the MediaPlayer class has the most events that can be triggered, while the Media and MediaView classes have only one event (OnError). You'll also notice the third column, showing onXXXXProperty() methods. These onXXXXProperty() methods are the same as the setOnXXXX() methods except that they also support property bindings; this gives them the ability to attach change and invalidation listeners. The following code snippet is an example of setting a Runnable (lambda expression) instance using the onXXXXProperty() method instead of the usual setOnXXXX() method:

```
Media media = new Media(url);
MediaPlayer mediaPlayer = new MediaPlayer(media);
mediaPlayer.onReadyProperty().set(() -> mediaPlayer.play());
```

Playing Audio

JavaFX's media API supports loading audio files with extensions such as .mp3, .wav, and .aiff. Also new in JavaFX 8 is the ability to play audio in HTTP live streaming format, also known as HLS (file extension .m3u8). HLS is beyond the scope of this book and will not be covered in this chapter, but with some further research, JavaFX could be used to build a live radio broadcast application.

Playing audio files is extremely straightforward in JavaFX. Given a valid URL location to a file, you would instantiate a javafx.scene.media.Media class. The Media object is then passed to a new instance of a javafx.scene.media.MediaPlayer object. The last step is to invoke the media player object's play() method when the OnReady event is triggered. The following code loads and plays an MP3 audio file located on a web server:

```
Media media = new Media("http://some_host/eye_on_it.mp3");
MediaPlayer mediaPlayer = new MediaPlayer(media);
mediaPlayer.setAutoPlay(true);
```

Note When loading media files, make sure the file location is a formatted string that follows the standard URL specification.

The media file can reside on a web server, in a jar file, or a local file system as long as the filename string is formatted to follow the standard URL specification.

You will also notice the code's last statement, which calls the method setAutoPlay(true). Calling the setAutoPlay() is really a shortcut for setting the setOnReady() event handler method with a Runnable functional interface. Both techniques will result in playing the audio file once the MediaPlayer instance is in a ready state.

Note For low-latency playback of audio files, use the javafx.scene.media.AudioClip class. A typical scenario is to play a given sound multiple times consecutively, as with sound effects used in games.

An MP3 Player Example

Now that you know how to load and play audio media, let's look at a fun example that involves playing music and displaying a colorful visualization. In this section you will learn how to create an MP3 audio player. Before we explore the example's code in Listing 7-1, let's look at Figure 7-1 for a preview of our audio player's UI.

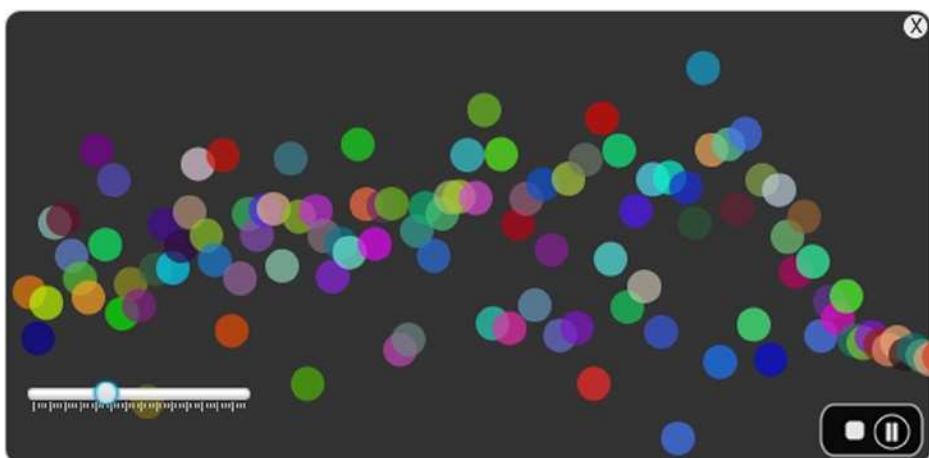


Figure 7-1. Our example JavaFX audio player

In this example we will again use the drag-and-drop metaphor, as we did in Chapter 5 with image files. In the same manner as in the earlier example, the user navigates to the local file system or a browser URL address to locate a media file to be dragged and dropped onto the application. In the case of our MP3 player, you will locate an audio file with a file extension of .mp3, .wav or .aif to be used in this example.

As you can see in Figure 7-1, our MP3 player displays randomly colored circles bouncing around based on the media's audio spectrum information (its phase array). The audio player has a custom button panel control located in the bottom-right corner. The button panel control lets the user pause, resume, and stop the playing of music. You'll also notice the seek position slider (aka the progress and seek position slider) on the bottom-left side, which allows the user to seek backward or forward in the media (audio or video).

The Stop, Play, and Pause Buttons

The left image in Figure 7-2 shows Stop (square) and Play (triangle wedge) buttons. When the Stop button is pressed, the media is repositioned to the beginning. When the Pause button is pressed, the media will maintain its current position within the media playback. Also, as the media is paused, the Play button will appear, so that the user can resume the music. The right image in Figure 7-2 shows the button panel containing a Stop and a Pause button (circle with two vertical lines). While the music is currently playing, the user can press the Pause button to pause the music.

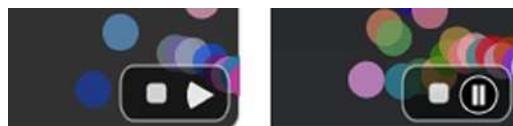


Figure 7-2. Custom button panel control having Stop (rectangle), Play (triangle), and Pause (circle) buttons

The Progress and Seek Position Slider Control

Next, you'll notice the progress and seek position slider control in the lower-left corner of the application window shown in Figure 7-3. Assuming the media is paused, the seek position slider control allows you to move backward or forward into the media's current time. Also, the slider's "thumb" will move from left to right as the media play is progressing.



Figure 7-3. The Seek Position slider control allows the user to seek backward or forward in the media

The Close Button

Finally, notice the Close button, shown in Figure 7-4, which simply allows the user to quit the application.

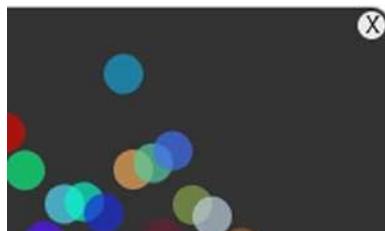


Figure 7-4. The Close button, located in the upper-right corner, allows the user to quit the application

MP3 Audio Player Source Code

Listing 7-1 is the source code for a JavaFX-based MP3 audio player. The source code from a previous edition of this book was refactored slightly to break things up into methods instead of one large start() method. In addition to the refactoring, I pulled out the code that used the programmatic approach to style JavaFX nodes. Instead of styling JavaFX nodes with setter() methods, I simply created a CSS style sheet. Listing 7-2 shows the CSS style sheet's selector definitions.

Listing 7-1. An MP3 Audio Player Application (*PlayingAudio.java*)

```
package jfx8ibe;

import java.net.MalformedURLException;
import java.util.Random;
import javafx.application.*;
import javafx.beans.value.ChangeListener;
import javafx.geometry.Point2D;
import javafx.scene.*;
import javafx.scene.control.Slider;
import javafx.scene.input.*;
import javafx.scene.media.*;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;
import javafx.scene.text.*;
import javafx.stage.*;
import javafx.util.Duration;

/**
 * Chapter 7 Playing Audio using JavaFX media API.
 *
 * @author cdea
 */
public class PlayingAudio extends Application {

    private MediaPlayer mediaPlayer;
    private Point2D anchorPt;
    private Point2D previousLocation;
    private ChangeListener<Duration> progressListener;

    private static Stage PRIMARY_STAGE;
    private static final String STOP_BUTTON_ID = "stop-button";
    private static final String PLAY_BUTTON_ID = "play-button";
    private static final String PAUSE_BUTTON_ID = "pause-button";
    private static final String CLOSE_BUTTON_ID = "close-button";
    private static final String VIS_CONTAINER_ID = "viz-container";
    private static final String SEEK_POS_SLIDER_ID = "seek-position-slider";

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

```
@Override
public void start(Stage primaryStage) {
    PRIMARY_STAGE = primaryStage;
    PRIMARY_STAGE.initStyle(StageStyle.TRANSPARENT);
    PRIMARY_STAGE.centerOnScreen();

    Group root = new Group();
    Scene scene = new Scene(root, 551, 270, Color.rgb(0, 0, 0, 0));

    // load JavaFX CSS style
    scene.getStylesheets()
        .add(getClass().getResource("playing-audio.css")
            .toExternalForm());
    PRIMARY_STAGE.setScene(scene);

    // Initialize stage to be movable via mouse
    initMovablePlayer();

    // application area
    Node applicationArea = createApplicationArea();

    // Container for random circles bouncing about
    Node vizContainer = new Group();
    vizContainer.setId(VIS_CONTAINER_ID);

    // Create the button panel
    Node buttonPanel = createButtonPanel();

    // Progress and seek position slider
    Slider progressSlider = createSlider();

    // Update slider as video is progressing
    progressListener = (observable, oldValue, newValue) ->
        progressSlider.setValue(newValue.toSeconds());

    // Initializing to accept files
    // dragged over surface to load media
    initFileDragNDrop();

    // Create the close button
    Node closeButton = createCloseButton();

    root.getChildren()
        .addAll(applicationArea,
               vizContainer,
               buttonPanel,
               progressSlider,
               closeButton);

    primaryStage.show();
}
```

```

/**
 * Initialize the stage to allow the mouse cursor to
 * move the application using dragging.
 *
 */
private void initMovablePlayer() {
    Scene scene = PRIMARY_STAGE.getScene();
    // starting initial anchor point
    scene.setOnMousePressed(mouseEvent
        -> anchorPt = new Point2D(mouseEvent.getScreenX(),
                                   mouseEvent.getScreenY()))
);

// dragging the entire stage
scene.setOnMouseDragged(mouseEvent -> {
    if (anchorPt != null && previousLocation != null) {
        PRIMARY_STAGE.setX(previousLocation.getX()
            + mouseEvent.getScreenX()
            - anchorPt.getX());
        PRIMARY_STAGE.setY(previousLocation.getY()
            + mouseEvent.getScreenY()
            - anchorPt.getY());
    }
});

// set the current location
scene.setOnMouseReleased(mouseEvent
    -> previousLocation = new Point2D(PRIMARY_STAGE.getX(),
                                         PRIMARY_STAGE.getY())
);

// Initialize previousLocation after Stage is shown
PRIMARY_STAGE.addEventHandler(WindowEvent.WINDOW_SHOWN,
    (WindowEvent t) -> {
        previousLocation = new Point2D(PRIMARY_STAGE.getX(),
                                       PRIMARY_STAGE.getY());
    });
}

/**
 * A simple rectangular area as the surface of the app.
 * @return Node a Rectangle node.
 */
private Node createApplicationArea() {
    Scene scene = PRIMARY_STAGE.getScene();
    Rectangle applicationArea = new Rectangle();
    // add selector to style app-area
    applicationArea.setId("app-area");
}

```

```
// make the app area rectangle the size of the scene.  
applicationArea.widthProperty()  
    .bind(scene.widthProperty());  
applicationArea.heightProperty()  
    .bind(scene.heightProperty());  
return applicationArea;  
}  
  
/**  
 * Initialize the Drag and Drop ability for media files.  
 */  
private void initFileDragNDrop() {  
  
    Scene scene = PRIMARY_STAGE.getScene();  
    scene.setOnDragOver(dragEvent -> {  
        Dragboard db = dragEvent.getDragboard();  
        if (db.hasFiles() || db.hasUrl()) {  
            dragEvent.acceptTransferModes(TransferMode.LINK);  
        } else {  
            dragEvent.consume();  
        }  
    });  
    // Dropping over surface  
    scene.setOnDragDropped(dragEvent -> {  
        Dragboard db = dragEvent.getDragboard();  
        boolean success = false;  
        String filePath = null;  
        if (db.hasFiles()) {  
            success = true;  
            if (db.getFiles().size() > 0) {  
                try {  
                    filePath = db.getFiles()  
                        .get(0)  
                        .toURI().toURL().toString();  
                    playMedia(filePath);  
                } catch (MalformedURLException ex) {  
                    ex.printStackTrace();  
                }  
            }  
        } else {  
            // audio file from some host or jar  
            playMedia(db.getUrl());  
            success = true;  
        }  
        dragEvent.setDropCompleted(success);  
        dragEvent.consume();  
    }); // end of setOnDragDropped  
}
```

```

/**
 * Creates a node containing the audio player's
 * stop, pause and play buttons.
 *
 * @return Node A button panel having play,
 * pause and stop buttons.
 */
private Node createButtonPanel() {
    Scene scene = PRIMARY_STAGE.getScene();
    // create button control panel
    Group buttonGroup = new Group();

    // Button area
    Rectangle buttonArea = new Rectangle(60, 30);
    buttonArea.setId("button-area");

    buttonGroup.getChildren()
        .add(buttonArea);

    // stop button control
    Node stopButton = new Rectangle(10, 10);
    stopButton.setId(STOP_BUTTON_ID);
    stopButton.setOnMousePressed(mouseEvent -> {
        if (mediaPlayer != null) {
            updatePlayAndPauseButtons(true);
            if (mediaPlayer.getStatus() == Status.PLAYING) {
                mediaPlayer.stop();
            }
        }
    });
}); // setOnMousePressed()

// play button
Arc playButton = new Arc(12, // center x
    16, // center y
    15, // radius x
    15, // radius y
    150, // start angle
    60); // length
playButton.setId(PLAY_BUTTON_ID);
playButton.setType(ArcType.ROUND);
playButton.setOnMousePressed(mouseEvent -> mediaPlayer.play());

// pause control
Group pauseButton = new Group();
pauseButton.setId(PAUSE_BUTTON_ID);
Node pauseBackground = new Circle(12, 16, 10);
pauseBackground.getStyleClass().add("pause-circle");

```

```
Node firstLine = new Line(6, // start x
                           6, // start y
                           6, // end x
                           14); // end y
firstLine.getStyleClass()
    .add("pause-line");
firstLine.setStyle("-fx-translate-x: 34;");

Node secondLine = new Line(6, // start x
                           6, // start y
                           6, // end x
                           14); // end y
secondLine.getStyleClass().add("pause-line");
secondLine.setStyle("-fx-translate-x: 38;");

pauseButton.getChildren()
    .addAll(pauseBackground, firstLine, secondLine);

pauseButton.setOnMousePressed(mouseEvent -> {
    if (mediaPlayer!=null) {
        updatePlayAndPauseButtons(true);
        if (mediaPlayer.getStatus() == Status.PLAYING) {
            mediaPlayer.pause();
        }
    }
}); // setOnMousePressed()

playButton.setOnMousePressed(mouseEvent -> {
    if (mediaPlayer != null) {
        updatePlayAndPauseButtons(false);
        mediaPlayer.play();
    }
}); // setOnMousePressed()
buttonGroup.getChildren()
    .addAll(stopButton,
           playButton,
           pauseButton);
// 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));
return buttonGroup;
}
```

```

/**
 * The close button to exit application
 *
 * @return Node representing a close button.
 */
private Node createCloseButton() {
    Scene scene = PRIMARY_STAGE.getScene();
    Group closeButton = new Group();
    closeButton.setId(CLOSE_BUTTON_ID);
    Node closeBackground = new Circle(5, 0, 7);
    closeBackground.setId("close-circle");
    Node closeXmark = new Text(2, 4, "X");
    closeButton.translateXProperty()
        .bind(scene.widthProperty()
            .subtract(15));
    closeButton.setTranslateY(10);
    closeButton.getChildren()
        .addAll(closeBackground, closeXmark);
    // exit app
    closeButton.setOnMouseClicked(mouseEvent -> Platform.exit());
    return closeButton;
}

/**
 * After a file is dragged onto the application a new MediaPlayer
 * instance is created with a media file.
 *
 * @param stage The stage window (primaryStage)
 * @param url The URL pointing to an audio file
 */
private void playMedia(String url) {
    Scene scene = PRIMARY_STAGE.getScene();

    if (mediaPlayer != null) {
        mediaPlayer.pause();
        mediaPlayer.setOnPaused(null);
        mediaPlayer.setOnPlaying(null);
        mediaPlayer.setOnReady(null);
        mediaPlayer.currentTimeProperty()
            .removeListener(progressListener);
        mediaPlayer.setAudioSpectrumListener(null);
    }
    Media media = new Media(url);
    // display media's metadata
    for (String s : media.getMetadata().keySet()) {
        System.out.println(s);
    }
    mediaPlayer = new MediaPlayer(media);

    // as the media is playing move the slider for progress
    mediaPlayer.currentTimeProperty()
        .addListener(progressListener);
}

```

```

mediaPlayer.setOnReady(() -> {
    updatePlayAndPauseButtons(false);
    Slider progressSlider =
        (Slider) scene.lookup("#" + SEEK_POS_SLIDER_ID);
    progressSlider.setValue(0);
    progressSlider.setMax(mediaPlayer.getMedia()
        .getDuration()
        .toSeconds());
    mediaPlayer.play();
}); // setOnReady()

// back to the beginning
mediaPlayer.setOnEndOfMedia( ()-> {
    updatePlayAndPauseButtons(true);
    // change buttons to play and rewind
    mediaPlayer.stop();
}); // setOnEndOfMedia()

// setup visualization (circle container)
Group vizContainer =
    (Group) PRIMARY_STAGE.getScene()
        .lookup("#" + VIS_CONTAINER_ID);
mediaPlayer.setAudioSpectrumListener(
    (double timestamp,
     double duration,
     float[] magnitudes,
     float[] phases) -> {
        vizContainer.getChildren().clear();
        int i = 0;
        int x = 10;
        double y = PRIMARY_STAGE.getScene().getHeight() / 2;
        Random rand = new Random(System.currentTimeMillis());
        // Build random colored circles
        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));
            vizContainer.getChildren().add(circle);
            i += 5;
        }
    });
}); // setAudioSpectrumListener()

}

/**
* Sets play button visible and pause button not visible when
* playVisible is true otherwise the opposite.
*

```

```

 * @param playVisible - value of true the play becomes visible
 * and pause non visible, otherwise the opposite.
 */
private void updatePlayAndPauseButtons(boolean playVisible) {
    Scene scene = PRIMARY_STAGE.getScene();
    Node playButton = scene.lookup("#" + PLAY_BUTTON_ID);
    Node pauseButton = scene.lookup("#" + PAUSE_BUTTON_ID);

    // hide or show buttons
    playButton.setVisible(playVisible);
    pauseButton.setVisible(!playVisible);
    if (playVisible) {
        // show play button
        playButton.toFront();
        pauseButton.toBack();
    } else {
        // show pause button
        pauseButton.toFront();
        playButton.toBack();
    }
}

/** 
 * A position slider to seek backward and forward
 * that is bound to a media player control.
 *
 * @return Slider control bound to media player.
 */
private Slider createSlider() {
    Slider slider = new Slider(0, 100, 1);
    slider.setId(SEEK_POS_SLIDER_ID);
    slider.valueProperty()
        .addListener((observable) -> {
            if (slider.isValueChanging()) {
                // must check if media is paused before seeking
                if (mediaPlayer != null &&
                    mediaPlayer.getStatus() == MediaPlayer.Status.PAUSED) {

                    // convert seconds to millis
                    double dur = slider.getValue() * 1000;
                    mediaPlayer.seek(Duration.millis(dur));
                }
            }
        });
    Scene scene = PRIMARY_STAGE.getScene();
    slider.setTranslateX(10);
    slider.translateYProperty()
        .bind(scene.heightProperty()
            .subtract(50));
    return slider;
}
}

```

Listing 7-2. MP3 Audio Player's CSS style sheet (playing-audio.css)

```
.root {  
    -white-ish: rgba(255, 255, 255, .90);  
    -black-ish: rgba(0, 0, 0, .80);  
}  
#app-area {  
    -fx-arc-width: 20;  
    -fx-arc-height: 20;  
    -fx-fill: -black-ish;  
    -fx-stroke: green;  
    -fx-stroke-width: 2;  
    -fx-stroke: -white-ish;  
}  
#button-area {  
    -fx-arc-width: 15;  
    -fx-arc-height: 20;  
    -fx-fill: -black-ish;  
    -fx-stroke: -white-ish;  
}  
#stop-button {  
    -fx-arc-width: 5;  
    -fx-arc-height: 5;  
    -fx-fill: -white-ish;  
    -fx-translate-x: 15;  
    -fx-translate-y: 10;  
    -fx-stroke: -white-ish;  
}  
#play-button {  
    -fx-fill: -white-ish;  
    -fx-translate-x: 40;  
}  
.pause-circle {  
    -fx-stroke: -white-ish;  
    -fx-translate-x: 30;  
}  
.pause-line {  
    -fx-stroke-width: 3;  
    -fx-stroke: -white-ish;  
    -fx-translate-y: 6;  
}  
#close-circle {  
    -fx-fill: -white-ish;  
}  
.slider {  
    -fx-show-tick-labels: false;  
    -fx-show-tick-marks: true;  
}  
.slider .axis .axis-label {  
    -fx-text-fill: white;  
}
```

How It Works

Before we begin I want to describe some changes that I made in this book from the earlier edition. I basically refactored the source code from the prior book's edition relating to media. Because of this refactoring, I will mention privately scoped methods quite often. These private methods are used to break-up the code into more manageable pieces, so that you can better focus on each feature of the media player application.

We will drag and drop audio files onto the example MP3 player, as we did in Chapter 5's photo viewer application, 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 loading audio files. To load audio files, JavaFX currently supports the following file formats: mp3, wav, aiff, and m3u8. To see the supported encoding types please refer to the Javadocs documentation of the package summary of `javafx.scene.media`.

Looking back at Figure 7-2 you might notice that the button panel control has a look and feel similar to the photo viewer application in Figure 5-1 of Chapter 5. In this example, I've modified the button controls to resemble buttons, similar to many media player applications.

As an added bonus, the MP3 player will appear as an irregularly shaped, semitransparent window without a title bar and borders. The application window can also be dragged around the desktop using the mouse cursor. Now that you know how to operate the music player, let's walk through the code.

The Audio Player Application's Instance Variables

First, the code contains instance variables at the top of the class that will maintain state information for the lifetime of the application. Table 7-2 describes all instance variables used in our MP3 audio player application.

Table 7-2. MP3 Player Application Instance Variables

Variable	Data Type	Example	Description
mediaPlayer	MediaPlayer	-	A media player object that plays audio and video.
anchorPt	Point2D	100,100	Screen coordinates where the user begins to drag the stage window.
previousLocation	Point2D	0,0	The upper-left corner of the stage's previous coordinate; assists in dragging the stage window.
progressListener	ChangeListener<Duration>	5 (sec.)	Bound to the media player's <code>currentTimeProperty()</code> method. Updates the UI slider control.

In Table 7-2 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 can see two variables named `anchorPt` and `previousLocation`. These variables are used to maintain the stage window's positions on the desktop when the user drags the application window across the screen.

The `anchorPt` variable is used to save the starting coordinates of a mouse press when the user begins to drag the window. When calculating the upper-left bounds of the application window during a mouse-drag operation, the `previousLocation` variable will contain the previous window's screen X and Y coordinates.

Not listed in Table 7-2 are static variables such as `PRIMARY_STAGE`, `STOP_BUTTON_ID`, `PLAY_BUTTON_ID`, and so on. The static `PRIMARY_STAGE` variable is a global variable reference as a convenience for all privately scoped methods for easy access to the main stage window. The many `static final String` variables named `XXXX_ID`, having an `_ID` suffix, are node IDs that can be later retrieved via the scene's `lookup()` method. For example, if the scene graph contains a node with an `id` of `stop-button` (`node.setId("stop-button")`), you could later invoke the `scene.lookup("#stop-button")` method to retrieve the Stop button node from the scene graph.

Setting Up the Stage Window

Now that you know about the instance variables, let's jump into the `start()` method code. In previous chapters relating to GUIs, you saw that GUI applications normally contain a title bar and windowed borders surrounding the Scene. Here, I want to demonstrate the ability to create irregularly shaped semi-transparent windows. The code invokes the `stage(primaryStage)` object's `initStyle()` method with a `StageStyle.TRANSPARENT` value; as a result, the stage window has no title bar (*is undecorated*) and is invisible.

Next, the code centers the stage window on the screen on application startup via the `centerOnScreen()` method. After centering the stage you'll notice the usual setup of a scene by applying a JavaFX style sheet file (`playing-audio.css`). Listing 7-2 shows the contents of the file `playing-audio.css`. There, you'll notice that the root class selector has two global color attributes, named `-white-ish` and `-black-ish`, created to be reused in other styling definition blocks. If you are not familiar with JavaFX CSS styling, please refer to Chapter 6.

Back in our `start()` method in Listing 7-1, after applying the CSS style sheet the code invokes the private `initMovablePlayer()` method, which is responsible for making the stage window draggable. Because you don't have a native title bar, you will find it convenient to be able to drag any part of the application's surface. The `initMovablePlayer()` method makes use of the instance variables `anchorPt` and `previousLocation` mentioned earlier, to calculate points based on relative screen coordinates when mouse events (press, drag, or release) are triggered.

Because our scene is invisible, the code will create the application background area by invoking the private method `createApplicationArea()`. This method returns a simple rounded rectangle as the background of our MP3 audio player application. The returned shape node will be added to the root node of the scene first, and all other nodes added to the root node after that. This order enables all other nodes to sit on top of the background application area.

The Setup Container Node for Visualizations (the Scene.lookup() Method)

After creating an application area background (the rounded rectangle), the code creates a container node (Group) that will later hold randomly colored circle shapes. The container node is refreshed with new circles that appear to dance around the application area. Following is the code to create the container node (vizContainer) that provides our visualization area node as audio is being played:

```
// Container for random circles bouncing about
Node vizContainer = new Group();
vizContainer.setId(VIS_CONTAINER_ID); // vis-container
```

Did you notice that the id selector was set to viz-container? By setting the id selector, we enable the code to retrieve the container node later via the scene's lookup() method, as shown here:

```
Group vizContainer = (Group) stage.getScene().lookup("#viz-container");
```

This code snippet shows the invocation of the lookup() method of the scene object. You will notice the '#' symbol prefixed with the id type selector name. According to the Javadocs documentation "If more than one node matches the specified selector, this function returns the first of them. If no nodes are found with this ID, then null is returned."

Using the scene's lookup() method facility is a convenient way to retrieve nodes from the scene graph in a random-access way without having to create globally defined instance variables.

Creating a Custom Button Panel

Continuing with our start() method, the code creates a custom button panel by invoking the private method createButtonPanel(). This method creates custom buttons by using JavaFX shapes (rectangles, arcs, circles, and lines) that represent the Stop, Play, and Pause buttons for our MP3 audio player.

When creating shapes or custom nodes, you can add event handlers to nodes in order to respond to mouse events. 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.

To attach event handlers for a mouse press, simply call the setOnMousePress() method by passing in an EventHandler<MouseEvent> instance. The following code block adds a lambda expression that responds to a mouse press for the stopButton node:

```
Node stopButton = new Rectangle(10, 10);
stopButton.setId(STOP_BUTTON_ID);
stopButton.setOnMousePressed(mouseEvent -> {
    if (mediaPlayer != null) {
        updatePlayAndPauseButtons(true);
        if (mediaPlayer.getStatus() == Status.PLAYING) {
            mediaPlayer.stop();
        }
    }
}); // setOnMousePressed()
```

In this code, a square node is defined as our Stop button, set with an id of the static String variable STOP_BUTTON_ID, which is really the string value stop-button. Next, the code sets an event handler for the OnMousePressed event to stop the media player from playing. You'll notice the method call to updatePlayAndPauseButtons(true), which toggles the display of the Play button and hiding of the Pause button. Otherwise, we pass in a boolean value of false, so the Pause button will be displayed and the Play button will be hidden.

Simply put, the media player object is a state machine that transitions between states internally. Some methods use other threads that aren't on the JavaFX application thread. So it is typically good practice to query the state of the media player before invoking methods such as the stop(). In the case of the Stop button, you've seen that the code checks for the Status.PLAYING before invoking the stop() method on the media player object.

Because all the buttons use code similar to the preceding code snippet (EventHandler), I will list only the method calls that each button will ultimately perform on the media player. 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();
```

Play Progress, Rewind, and Fast Forward

After creating the custom button panel in our start() method, the code invokes the private method createSlider(), which returns a JavaFX Slider UI control. The slider control represents a progress position slider that moves based on the position of the media as it is being played. When the media is paused, the user may also drag the slider control's "thumb" to seek backward and forward in time through the audio or video media.

Seeking Backward or Forward in the Media

The following code implements a slider control representing a progress (seek) position slider control for the MP3 audio player application:

```
Slider slider = new Slider(0, 100, 1);
slider.setId(SEEK_POS_SLIDER_ID);
slider.valueProperty()
    .addListener((observable) -> {
        if (slider.isValueChanging()) {
            // must check if media is paused before seeking
            if (mediaPlayer != null
                && mediaPlayer.getStatus() == MediaPlayer.Status.PAUSED) {
                //convert seconds to millis
                double dur = slider.getValue() * 1000;
                mediaPlayer.seek(Duration.millis(dur));
            }
        }
}); //addListener()
```

Notice the slider control's valueProperty() method and the call to the addListener() method, which has an InvalidationToken (lambda expression) that detects whether the user has moved the slider control, thus changing the value property. The lambda parameter for an InvalidationToken (functional interface) is an object of type javafx.beans.Observable that refers to the slider's value property.

The code first checks whether the slider's value has changed, by using the method isValueChanging(). It's important to perform this check before the seek() method because of performance efficiencies. Using the InvalidationToken, the slider's value changes only when it's needed (an example of lazy evaluation). If you used a ChangeListener (eager evaluation), the value would change too often and could overwhelm the media player.

Using a Slider Control to Show Play Progress

Still in our start() method, the code assigns the variable progressListener to a ChangeListener (lambda) instance as shown in the following code snippet:

```
// update slider as media is progressing
progressListener = (observable, oldValue, newValue) ->
    progressSlider.setValue(newValue.toSeconds());
```

The progressListener is responsible for updating the slider's "thumb" as the media player is playing audio or video media. The listener is added onto the media player object's currentTimeProperty() in the playMedia() method. In this private method, the code cleans up by removing the singleton progressListener instance, which was bound to an old media player object, and adds the progressListener onto a newly created media player object. By removing old references to the progress listener, we allow the Java runtime to perform its garbage collection process, thus preventing memory leaks.

Setting Up Drag-and-Drop Support for Audio Files

Continuing within our start() method, notice the call to a private method initFileDragNDrop(). I created this method to support the drag-and-drop capability that is responsible for loading an audio file from the local file system or browser URL address field. Because the implementation is nearly identical to Chapter 5's photo viewer example, I will not discuss the code details, except to mention the invocation of the private method playMedia(). After the file is dropped the URL string is passed into the method playMedia(). This method will conveniently clean up and create a new media player and immediately play the media.

Cleaning Up

Inside the initFileDragNDrop() method is the OnDragDropped event handler code, which invokes the private playMedia() method. This method begins by checking for the existence of a previously created mediaPlayer object. If there was a previous media player instance, the code would stop the media player and clean up any old references to events, listeners, and properties.

Media Metadata

Next, the code creates a `Media` class instance with a valid URL string. After creating a `Media` instance, the code loops through the media's metadata via the `getMetadata()` method to display the keys onto the console. Sometimes MP3 music files will contain information about the artist, tracks, or song titles. By interrogating the media object's metadata, you can display the information to the user.

After listing out metadata, the code creates a new `MediaPlayer` instance with a new `Media` object and adds the `progressListener` onto the media player's `currentTimeProperty()` method. As mentioned earlier for the progress position slider control, the "thumb" will progress forward as the media is being played.

Playing Media (the OnReady Event)

Continuing within the `playMedia()` method, the code creates a `Runnable` (lambda expression) instance to respond to the `OnReady` event on the media player object. The code first resets the Play and Pause buttons of the custom button control panel by invoking the `updatePlayAndPauseButtons()` method. Passing in `true` displays the Play button and hides the Pause button; otherwise the Pause button is shown and the Play button is hidden.

Next, the code initializes the slider control based on the media's total duration. Finally, the code invokes the media player object's `play()` method to begin playing the music media.

Rewinding (the OnEndOfMedia Event)

After setting up the media player's `OnReady` event, the code sets up the `OnEndOfMedia` event. This event is triggered when the media player play position has reached the end. The code toggles the Play and Pause buttons' appearance and rewinds the media to the start, placing it in a Ready state.

Updating the Visualization Using the AudioSpectrumListener Interface

Do you remember the earlier discussion about the MP3 audio player's visualization of randomly colored circles dancing about as the music plays? Still within the `playMedia()` method, let's look at how to implement this mesmerizing effect.

Continuing with our `playMedia()` method, the code will obtain the container node `viz-container` that was created earlier. The following lines retrieve the container node using the scene's `lookup()` method:

```
// setup visualization (circle container)
Group vizContainer = (Group) PRIMARY_STAGE.getScene()
    .lookup("#" + VIS_CONTAINER_ID);
```

Next, the code sets the media player's audio spectrum listener with an `AudioSpectrumListener` (lambda expression) instance. The listener is notified periodically with updates of the audio spectrum. The following code updates the container node (`vizContainer`) with randomly colored Circle shapes based on the phases array:

```
mediaPlayer.setAudioSpectrumListener(  
    (double timestamp,  
     double duration,  
     float[] magnitudes,  
     float[] phases) -> {  
        vizContainer.getChildren().clear();  
        int i = 0;  
        int x = 10;  
        double y = PRIMARY_STAGE.getScene().getHeight() / 2;  
        Random rand = new Random(System.currentTimeMillis());  
        // Build random colored circles  
        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));  
            vizContainer.getChildren().add(circle);  
            i += 5;  
        }  
    }); // setAudioSpectrumListener()
```

This code creates randomly colored circle nodes to be positioned and placed on the scene based on the phases array, an array of float values. To draw each colored circle, the code increments the Circle's center X by 5 pixels and adds the circle's center Y with each phase value multiplied by 100. Shown next is the code that plots each randomly colored circle:

```
circle.setCenterX(x + i);  
circle.setCenterY(y + (phase * 100));  
... // setting the circle  
i+=5;
```

What exactly is this type of listener? According to the Javadocs, 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, tempo, and so on. To create an instance of an `AudioSpectrumListener` you will create a lambda expression with parameters of the interface's method `spectrumDataUpdate()`. Table 7-3 lists all the inbound parameters for the audio spectrum listener's method. For more details refer to the Javadocs documentation for `javafx.scene.media.AudioSpectrumListener`.

Table 7-3. 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[]	-50.474	An array of float values representing each band's spectrum magnitude in decibels (nonpositive float value).
phases	float[]	1.22	An array of float values representing each band's phase. Each value range is between -Math.PI and +Math.PI.

Quitting (the Close Button)

The rest of the `start()` method in our MP3 audio player's code creates a Close button by calling the private method `createCloseButton()`, which returns a custom node that has handler code to exit the application.

Playing Video

Playing videos in JavaFX is quite simple. You will use the same APIs as with audio to load media into a JavaFX media player object. The only difference is that you will need a `MediaView` node to display the video onto the scene. Because `MediaView` is a JavaFX node, you will be able to transform, scale, translate, and apply effects. In this section you will learn about cross-platform video media file formats, and later you will see an example of a JavaFX video player.

Note As of the writing of this book, the JavaFX media player API supports the video formats MPEG-4 H.264/AVC (.mp4) and VP6 using an .flv container format.

MPEG-4

MPEG-4 is a multimedia container format containing H.264/AVC video. Currently MPEG-4 is a compressed video format that is widely used to view video on the Internet. You'll find that some high-definition video recorders can automatically convert raw video to MPEG-4 to produce files with the extension .mp4. For the devices that do not produce mp4 files, you currently need to purchase movie or video editing software capable of converting raw videos into MP4-formatted files.

The following software, among others, can convert raw video into the portable MPEG-4 format:

- Adobe Premiere
- Apple Final Cut Pro
- Apples iMovie
- Apple Quick Time Pro

- Sony Vegas Movie Studio
- Sony Vegas Pro
- FFmpeg

VP6 .flv

JavaFX 2.x and later 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 2010, On2 was acquired by Google to build VP7 and VP8 to be open and free to advance HTML5.

Please refer to the Javadocs 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>).

You are probably wondering how to obtain such a file of the type FLV when 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 one of the examples in this chapter relating to closed captioning. To obtain the .flv file, navigate to the following URL:

<http://www.mediacollege.com/adobe/flash/video/tutorial/example-flv.html>.

Next, locate the link entitled Windy 50s Mobility Scooter Race, which points to our .flv media file (20051210-w50s.flv). In order to download a link to the desired file, right-click to select Save Target As or Save. Once you have saved the file locally on your file system, you can use it for examples in this chapter.

It is *not* required that you have FLV files for this chapter. And you may use MPEG-4 (mp4) formatted files for the examples.

A Video Player Example

Figure 7-5 shows an Mpeg-4 (.mp4) home video shot from my FlipCam. I simply dragged the mp4 file right onto the surface of the application, and the video began playing. In this section you will learn how to implement a basic JavaFX video player application.

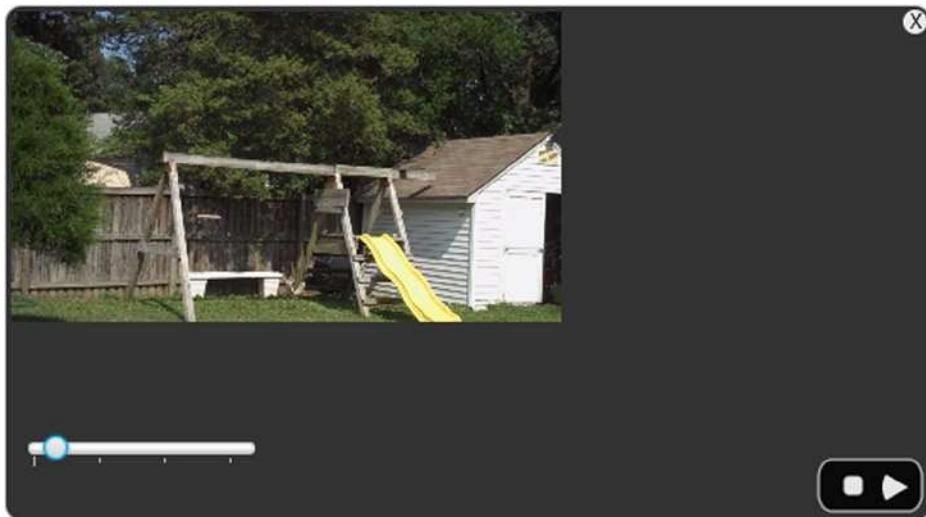


Figure 7-5. A basic JavaFX video player

To create a video player, all you need to do is reuse the previous MP3 audio player example and make very minor adjustments. Actually, the code is mostly identical to the MP3 audio player example, except that the visualization container node (Group) is replaced with a `javafx.scene.media.MediaView` node as shown in Figure 7-5.

Because the core functionality hasn't changed, you will experience the same features such as seeking, stopping, pausing, and playing media. Another bonus feature added to the video player is the ability to put the video into full-screen mode by double-clicking the application window area. To restore the stage window, repeat the double-click or press the Esc key.

Note Because all video player examples in this chapter were built from the MP3 audio player project, you will notice that features such as seeking, stopping, pausing, and playing media are identical. For the sake of brevity in the remaining examples I will list only the relevant code parts, demonstrating new features, and omit sections that existed in the core code from the MP3 Audio Player example.

Video Player Source Code

Listing 7-3 is the implementation of a JavaFX video player application. After you have examined the code, I will discuss how it works.

Listing 7-3. Source code for a JavaFX Video Player Application

```
/*
 * Chapter 7: Playing Video
 *
 * @author cdea
 */
public class PlayingVideo extends Application {
    // ... the rest of the instance variables
    private static final String MEDIA_VIEW_ID = "media-view";

    // ... The rest of the static variables

    @Override
    public void start(final Stage primaryStage) {
        // ... Code omitted

        // Initialize stage to have fullscreen ability
        initFullScreenMode();

        // Create a media view to display video
        MediaView mediaView = createMediaView();

        // make media view as the second node on the scene.
        root.getChildren().add(1, mediaView);

        // ... Code omitted
    }

    /**
     * Attaches event handler code (mouse event) to
     * toggle to Full screen mode. Double click the scene.
     */
    private void initFullScreenMode() {
        Scene scene = PRIMARY_STAGE.getScene();
        // Full screen toggle
        scene.setOnMouseClicked((MouseEvent event) -> {
            if (event.getClickCount() == 2) {
                PRIMARY_STAGE.setFullScreen(!PRIMARY_STAGE.isFullScreen());
            }
        });
    }

    /**
     * Initialize the stage to allow the mouse cursor to move the
     * application using dragging.
     * @param stage
     */
    private void initMovablePlayer() {
        // ... Code omitted
    }
}
```

```
/**  
 * Initialize the Drag and Drop ability for audio/video files.  
 * @param stage Primary Stage  
 */  
private void initFileDragNDrop() {  
  
    // ... Code omitted  
}  
  
/**  
 * After a file is dragged into the application a new  
 * MediaPlayer is created with a media file.  
 * @param stage The stage window (primaryStage)  
 * @param url The url pointing to an audio file  
 */  
private void playMedia(String url) {  
  
    // ... Code omitted  
    // create a new media player  
    // ... Code omitted  
  
    // set the media player to display video  
    MediaView mediaView = (MediaView) scene.lookup("#" + MEDIA_VIEW_ID);  
    mediaView.setMediaPlayer(mediaPlayer);  
} // playMedia()  
  
/**  
 * Create a MediaView node.  
 *  
 * @return MediaView  
 */  
private MediaView createMediaView() {  
    MediaView mediaView = new MediaView();  
    mediaView.setId(MEDIA_VIEW_ID);  
  
    mediaView.setPreserveRatio(true);  
    mediaView.setSmooth(true);  
    mediaView.fitWidthProperty()  
        .bind(PRIMARY_STAGE.getScene()  
            .widthProperty()  
            .subtract(220));  
    mediaView.fitHeightProperty()  
        .bind(PRIMARY_STAGE.getScene()  
            .heightProperty()  
            .subtract(30));  
    // sometimes loading errors occur  
    mediaView.setOnError(mediaErrorEvent -> {  
        mediaErrorEvent.getMediaError()  
            .printStackTrace();  
    });  
  
    return mediaView;  
}
```

```
/**  
 * Creates a node containing the audio player's stop,  
 * pause and play buttons.  
 *  
 * @param primaryStage  
 * @return Node A button panel having play,  
 * pause and stop buttons.  
 */  
private Node createButtonPanel() {  
    // ... Code omitted  
}  
  
/**  
 * A simple rectangular area as the surface of the app.  
 * @param stage the primary stage window.  
 * @return Node a Rectangle node.  
 */  
private Node createApplicationArea() {  
    // ... Code omitted  
}  
  
/**  
 * A slider to seek backward and forward that is bound to  
 * a media player control.  
 * @return Slider control bound to media player.  
 */  
private Slider createSlider() {  
    // ... Code omitted  
}  
  
/** The close button to exit application  
 * @return Node representing a close button.  
 */  
private Node createCloseButton() {  
    // ... Code omitted  
}  
  
/**  
 * Sets play button visible and pause button not visible  
 * when playVisible is true otherwise the opposite.  
 * @param playVisible - value of true the play becomes  
 * visible and pause non visible.  
 */  
private void updatePlayAndPauseButtons(boolean playVisible) {  
    // ... Code omitted  
}  
}
```

How It Works

Let's begin by discussing the instance and static variables defined in the class. Because this example is mostly the same as the MP3 audio player example, I will only discuss the newer variables. In Listing 7-3 the code creates a static final String MEDIA_VIEW_ID variable for the MediaView object's id for later retrieval using the scene's lookup() method.

Setting Up the Stage Window

Beginning with the start() method, the code has already set up the capability to drag the stage window, via the call to the initMovablePlayer() method (code omitted).

Next, the code invokes the initFullScreenMode() method, which attaches handler code to listen for a mouse double-click event. When a user double-clicks the background surface (scene), the application and media view will become full screen. The following code snippet is handler code set on the OnMouseClicked event.

```
// Full screen toggle
scene.setOnMouseClicked((MouseEvent event) -> {
    if (event.getClickCount() == 2) {
        PRIMARY_STAGE.setFullScreen(!PRIMARY_STAGE.isFullScreen());
    }
});
```

The MediaView Node

After adding the full-screen capability, the code invokes the private method createMediaView(), which returns a MediaView node that is responsible for displaying the video. The following code calls the createMediaView() to be added as the second node, which places the media view node on top of the application background (rounded rectangle):

```
// Create a media view to display video
MediaView mediaView = createMediaView();

// make media view as the second node on the scene.
root.getChildren().add(1, mediaView);
```

To see what the createMediaView() method actually returns, let's look at how it was implemented. The method begins by instantiating a new MediaView class and setting the id with MEDIA_VIEW_ID (media-view). Next, it invokes the media view object's setPreserveRatio() with a value of true. The setPreserveRatio() method will preserve the aspect ratio of the video when being resized. The resizing occurs when the application goes into full screen mode and back to normal size. The remaining code binds properties relating to the width and height relative to the size of the scene's (stage window) dimensions.

Playing Video

Finally, consider the private method `playMedia()`, which is responsible for receiving a media file after a drag-and-drop. Here, is where the code takes the existing media player object to be associated with the `MediaView` node created earlier. The following code uses the scene's `lookup()` method to retrieve the media view and reference the newly created media player instance:

```
// set the media player to display video
MediaView mediaView = (MediaView) scene.lookup("#" + MEDIA_VIEW_ID);
mediaView.setMediaPlayer(mediaPlayer);
```

Simulating Closed Captioning: Marking a Position in a Video

Have you ever wanted to implement closed-captioned video? In the past there have been discussions about JavaFX 8 possibly supporting the standard closed-captioning specification for video media; however, the state of this feature is unclear as of this writing. Fortunately, there is hope; JavaFX supports media event markers to achieve the same behavior as closed-captioned video.

Closed-Captioning Video Example

To demonstrate a closed-captioned video example, I created a NetBeans project called `ClosedCaptionVideo`. For the sake of space I will show here only the code parts that are relevant to media event markers (Listing 7-4). To see the full source code, please visit the book's website to download the code for this chapter.

When we discussed video media file formats earlier, I noted that an FLV file could be found at the following URL: <http://www.mediacollege.com/adobe/flash/video/tutorial/example-flv.html>. Here, I used the file `20051210-w50s.flv` for this closed caption video example. Figure 7-6 depicts closed captions (Text) being displayed at certain points in the video.



Figure 7-6. Closed-captioned video

Listing 7-4. Media Marker Events

```
// obtain media
Media media = new Media(url);

// add media markers
media.getMarkers().put("Starting race", Duration.millis(1959));
media.getMarkers().put("He is begining \nto get ahead",
    Duration.millis(3395));
media.getMarkers().put("They are turning \nthe corner",
    Duration.millis(6060));
media.getMarkers().put("The crowds \ncheer",
    Duration.millis(9064));
media.getMarkers().put("He makes the \nfinish line",
    Duration.millis(11546));

MediaPlayer mediaPlayer = new MediaPlayer(media);

Text closedCaption = (Text) scene.lookup("#" + CLOSED_CAPTION_TEXT_ID);
mediaPlayer.setOnMarker((MediaMarkerEvent event) ->
    closedCaption.setText(event.getMarker().getKey()))
); setOnMarker()
```

How It Works

Listing 7-4 shows a newly created `Media` object based on a valid URL string. Next, the code adds key/value pairs as markers in `media`. Each marker pair consists of a key of type `String` and a value of type `Duration` (in milliseconds). The duration is the marker in time into the media that will trigger an `OnMarker` event. The media can be video or audio.

You'll immediately notice that the `setOnMarker()` method is set with a lambda expression with a parameter of type `MediaMarkerEvent`. The `MediaMarkerEvent` closure code is invoked when the `OnMarker` event is raised. The closure code simply sets the `closedCaption` (`javafx.scene.text.Text`) node with the key's value as the closed-captioning text.

Summary

In this chapter you discovered the world of JavaFX media. You began by learning about media events and event-based programming when dealing with the JavaFX Media APIs. Next, you saw how easy it is to take a media file as a URL string to be loaded and played. After learning about the `MediaPlayer` API, you got a chance to look at an MP3 audio player example that demonstrated features such as seek, stop, pause and play. Also, you learned how to implement a cool visualization using the `AudioSpectrumListener` interface.

After you learned about audio media, we discussed video. With video media you were able to learn about currently supported media formats such as Mpeg-4 and FLV. Then you saw an example of a basic JavaFX video player application. Here, you discovered that playing video is the same as playing audio, except for the use of the `MediaView` node, which is capable of displaying the video. Finally, you were exposed to a short example of media marker events (`MediaMarkerEvent`) to simulate a closed-captioned video.

Chapter 8

JavaFX on the Web

JavaFX provides capabilities to interoperate with HTML5 content. The underlying web page-rendering engine in JavaFX is the popular open-source API called WebKit. This API is used in Apple's Safari browsers, Amazon's Kindle devices, and was used in Google's Chrome browser prior to version 27 (the WebKit fork called Blink). HTML5 is the newly accepted standard markup language for rendering content in web browsers. HTML5 content consists of JavaScript, CSS, Scalable Vector Graphics (SVG), Canvas API, Media, XML, 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 native desktop software. This new breed of application is called the Rich Internet Application (RIA).

In this chapter, you will examine example applications that do the following:

- Display HTML5 Content (SVG-based Analog Clock)
- Communicate from Java to JavaScript (WeatherWidget)
- Communicate from JavaScript to Java (WeatherWidget)

Before delving into the example applications, we will discuss the following core Web APIs:

- `javafx.scene.web.WebEngine`
- `java.net.HttpURLConnection`
- `javafx.scene.web.WebView`

WebEngine

JavaFX provides a non-GUI component capable of loading HTML5 content, called the WebEngine API (`javafx.scene.web.WebEngine`). This API is basically an object instance of the `WebEngine` class to be used to load a file containing HTML5 content. The HTML5 file to be loaded can be located on a

local file system, a web server, or inside a JAR file. When you load a file using a web engine object, a background thread is used to load web content so that it does not block the JavaFX application thread. In this section you will learn the following two `WebEngine` methods for loading HTML5 content:

- `load(String URL)`
- `loadContent(String HTML)`

WebEngine's `load()` Method

The `WebEngine` API, like many APIs that use background threads, adheres to an event-based programming model. Being event-based usually means that method invocations are asynchronous (callbacks); for example, a web engine can load web content asynchronously from a remote web server and notify handler code when the content is finished loading. After being notified, the handler code would then be running on the JavaFX application thread. The code in Listing 8-1 loads HTML content from a remote web server in a background worker thread. To monitor or determine whether the worker thread has finished a `javafx.beans.value.ChangeListener` (lambda expression) is added to the state property. Please refer to chapter 3 on Lambda expression if you are not familiar with functional interfaces.

Listing 8-1. `WebEngine` with a `ChangeListener` Functional Interface that determines whether the background loading is finished

```
WebEngine webEngine = new WebEngine();
webEngine.getLoadWorker()
    .stateProperty()
    .addListener( (obs, oldValue, newValue) -> {
        if (newValue == SUCCEEDED) {
            // finished loading...
            // process content...
        }
}); // addListener()

// begin loading...
webEngine.load("http://www.oracle.com");
```

In Listing 8-1 you'll notice a `ChangeListener` lambda expression being added onto the state property via the `addListener()` method. The state property contains the current status or life cycle of the web engine's background worker thread. The states are based on the `State` enum, which is owned by the `javafx.concurrent.Worker` interface. After setting up the listener, the code begins the loading process by using the web engine's `load()` method. Also notice that the string passed into the `load()` method is a valid URL string representing the location of the web content file.

When a `ChangeListener` lambda expression is added to the state property, the parameters are as follows:

- `obs` is of type `ObservableValue<? extends Worker.State>`, which references the state property on the web engine object.
- `oldValue` is of type `Worker.State` enum and contains the previous value before the state property has changed.
- `newValue` is of type `Worker.State` enum also and contains the new state property to be changed into.

The type specifiers for each parameter of the lambda expression (`ChangeListener`) were left off to allow the parameter signature to be more concise. After the state property changes, the `newValue` parameter can be checked for various thread states. In Listing 8-1 the `newValue` is used to check the state of the worker thread, an example of eager evaluation.

The following are all of the possible worker thread states:

READY, SCHEDULED, RUNNING, SUCCEEDED, CANCELLED, FAILED

To determine whether the worker thread is finished loading the web content, the code in Listing 8-1 shows the `ChangeListener` inside of the `addListener()` method simply checking the worker thread's state against the `Worker.SUCCEEDED` enum value.

WebEngine's `loadContent()` Method

One additional way to load HTML5 content is by using the web engine's `loadContent(String htmlText)` method. What's nice about the `loadContent()` method is that HTML content represented as a string can be pregenerated dynamically without having to fetch content from a remote server. The following code snippet loads pregenerated HTML content:

```
webEngine.loadContent("<html><body><b>JavaFX Rocks!</b></body></html>");
```

Now that you know two strategies to load HTML content, you will see how to obtain and manipulate the raw content, along with see common types of data formats that may be obtained from the web engine object after the web content is loaded.

HTML DOM Content

The web engine has the ability to load the current page's document object model (DOM) as XML content following the W3C standards-based APIs for Java. Typically, developers who are familiar with an XML Document Object Model (DOM) can easily interrogate `org.w3c.dom.Document` objects. This section shows how to obtain a W3C XML Document object and the raw XML as a `String` object.

Obtaining an `org.w3c.dom.Document` (DOM) Object

After a web engine instance successfully loads HTML content, an XML DOM can be easily obtained by invoking the web engine's `getDocument()` method. The following code snippet obtains a `Document` (`org.w3c.dom.Document`) instance assuming the web engine is finished loading the HTML or XML content.

```
webEngine.load("http://myserver.com/my_cool_xml_data.xml");
.
.
// after content is finished loading (Worker.SUCCEEDED)
org.w3c.dom.Document xmlDom = webEngine.getDocument();
```

Using Raw XML Content as a String

Depending on the situation, sometimes it is necessary to obtain the raw XML data as a string of text. Developers often use libraries such as JDOM or dom4j to convert easily between the two data format types. Listing 8-2 shows how to convert from an org.w3c.dom.Document object to a String object without the need of using those libraries.

Listing 8-2. Converting an XML DOM into a String

```
TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
StringWriter stringWriter = new StringWriter();
transformer.transform(new DOMSource(webEngine.getDocument()),
    new StreamResult(stringWriter));
String xml = stringWriter.getBuffer().toString();
```

Raw HTML5 Content via the JavaScript Bridge

Early in this chapter we discussed how the underlying web engine uses WebKit to load and interact with HTML5 content. What does it mean to interact with HTML5 content? JavaFX's WebEngine API has a JavaScript bridge that allows Java code to invoke JavaScript functions or script code inside HTML5 content. So another question would be how then does one obtain the raw HTML5 content as a string?

Some common use cases out in the field are the following: obtaining raw HTML content for web crawling (used in search engines), caching web pages, screen scraping, data mining, and data analysis. To obtain the raw HTML5 you will need to interact with the JavaScript bridge to access the web content's DOM using the web engine's executeScript() method. The following code statement accesses the HTML document (DOM) to obtain the raw content from the documentElement.outerHTML (the topmost element):

```
String html = (String) webEngine.executeScript("document.documentElement.outerHTML");
```

Communicating from Java to JavaScript

Another example of using the method executeScript() is shown in Listing 8-3, where the code calls into a JavaScript function sayHello() to add text into an existing HTML DIV element with an id of my_message. Listing 8-4 shows the HTML content.

Listing 8-3. Java Calling a JavaScript Function

```
// From Java Code
webEngine.executeScript("sayHello('Hi there');");
.
.
.
```

Listing 8-4. HTML Content Containing the JavaScript Function sayHello()

```
... Omitted HTML content
<script>
function sayHello( msg ) {
    document.getElementById('my_message').innerHTML = msg;
}
</script>
.
.
<div id="my_message"></div>
```

Being able to call JavaScript is a powerful facility, allowing Java code to use the WebEngine object's JavaScript engine to manipulate the DOM. But what about JavaScript code talking to Java code?

Communicating from JavaScript to Java

Java 8 has the ability to allow JavaScript code to make calls into Java code. Allowing JavaScript to call Java code means that web developers can now take advantage of many Java APIs. In this section we will briefly discuss how to allow JavaScript code to communicate with Java code.

Enabling JavaScript to talk to Java is easy. The first step is obtaining the WebEngine object's JSObject object via a call to the method executeScript("window"). The JSObject is a wrapper object to allow the bridge between the two languages.

After obtaining the JSObject, any object can be passed into the JavaScript engine's context simply by invoking the setMember() method. To invoke the setMember() method, you pass in a string name and the corresponding object. Listing 8-5 shows how to set up the JSObject for the JavaScript engine to talk to Java.

Listing 8-5. Allowing JavaScript Code to Make Upcalls to Java Code

```
// The web view's web engine
webView.getEngine()
    .getLoadWorker()
    .stateProperty()
    .addListener( (obs, oldValue, newValue) -> {
        if (newValue == Worker.State.SUCCEEDED) {
            // Let JavaScript make upcalls to this (Java) class
            JSObject jsobj = (JSObject) webView.getEngine()
                .executeScript("window");
            jsobj.setMember("ABCD", new HelloWorld());
        }
    });
});
```

You can see in Listing 8-5 that the HelloWorld class instance is set as the member on the (JSObject) object. Listing 8-6 shows the simple HelloWorld class.

Listing 8-6. Java Class With the Public Method sayGoodbye()

```
public class HelloWorld () {  
    public String sayGoodbye(String name) {  
        return "Hasta la vista " + name;  
    }  
}
```

After you've set a `HelloWorld` instance as a `JSONObject` member named `ABCD`, the JavaScript function shown in Listing 8-7 can invoke the Java method `sayGoodbye()`.

Listing 8-7. JavaScript Code Calling Java Code

```
.  
. .  
<script>  
function sayGoodbye(name) {  
    var message = ABCD.sayGoodbye(name);  
    document.getElementById('my_message').innerHTML = message;  
}  
</script>  
. .  
<div id="my_message"></div>
```

As a warning, please be careful exposing methods that allow client code to gain access that can potentially harm or view private data.

Caution Because JavaScript code has the capability to make calls into Java, it is very important to make sure you aren't exposing APIs that could potentially harm systems or view private data.

The URL and HttpURLConnection APIs

Trying to fit a square peg into a round hole

—Sydney Smith

As mentioned earlier in the chapter, the `WebEngine` object is a non-GUI component capable of loading web content, so you might be wondering if it is possible to make web service calls capable of retrieving serialized data that has different data formats such as the popular JSON format. To answer this question, you will want to know the `WebEngine` API's responsibilities and limitations. In this section you will also briefly learn about how to use Java 8's core `java.net.URL` and `java.net.HttpURLConnection` APIs to make proper RESTful web requests.

WebEngine Limitations

When using JavaFX's WebEngine API to retrieve web content such as JSON data, you will encounter some limitations; some of these involve the ability to fetch (GET) data from or submit (POST) it to a web server asynchronously (using AJAX), such as when retrieving JSON data.

Of course there are a handful of interesting ways (hacks) to get JSON data by using the WebEngine object by calling into the HTML DOM to obtain text content. Following is an example of how to obtain the JSON text content inside an HTML DOM.

```
webEngine.load("http://myserver.org/current-weather?format=json");
.
.
.

// handler code to obtain JSON data.
String json = (String) webEngine.getDocument()
    .getElementsByTagName("body")
    .item(0)
    .getTextContent();
```

The handler code traverses the HTML's DOM to obtain text content retrieved from the web engine object. Even though it looks fairly easy to implement in order to get JSON data as a String, I do not recommend using this approach, because it is fragile and nonstandard. From a code maintenance standpoint, the code could be taken out of context and a potential bug could surface in the future. I'm sure you've heard the phrase "Just because you can do something doesn't mean you should."

You will quickly find that the WebEngine API's responsibility is basically for loading HTML5 and interacting with its content (HTML/JavaScript). The WebEngine API is not responsible for making web service calls or serializing data with different transport protocols.

So, is there a better way? Of course there is. Originally, since versions 1.1 and 1.2, Java has had the HttpURLConnection API. While there are many libraries you could choose from that simplify web service calls the next section will show you how to do things purely in Java having no library dependencies. Next, are examples on how to make RESTful requests using the HttpURLConnection API.

Making RESTful Call Requests

In the Enterprise Web development world you will inevitably encounter the concept of RESTful web services. Based on the concept of representational state transfer (REST), RESTful web services are lightweight HTTP requests that involve XML- or JSON-formatted data. To make RESTful calls the web request will contain a request method that tells the web server the type of HTTP data flow transaction. Similar to database transactions, RESTful conventions follow the concept of CRUD (Create, Read, Update, and Delete), where the create stage is a POST request, read is a GET, update is a PUT, and delete is a DELETE.

The HTTP GET Request

Because RESTful services are beyond the scope of this book, I will detail only the two common RESTful call requests, GET and POST. Listing 8-8 is the familiar HTTP GET request that developers typically perform when retrieving JSON data.

Listing 8-8. A RESTful GET Request to Retrieve Weather Data

```
// rest of the class code omitted
public static void main(String[] args){
    System.out.println(
jsonGetRequest("http://myserver.org/current-weather?q=Pasadena,MD&format=json"));
.
.
.

public static String jsonGetRequest(String urlQueryString) {
    String json = null;
    try {
        URL url = new URL(urlQueryString);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setDoOutput(true);
        connection.setInstanceFollowRedirects(false);
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setRequestProperty("charset", "utf-8");
        connection.connect();
        InputStream inStream = connection.getInputStream();
        json = streamToString(inStream); // input stream to string
    } catch (IOException ex ) {
        ex.printStackTrace();
    }
}

return json;
}
```

In Listing 8-8 the static method `jsonRequest()` is responsible for retrieving JSON data from a RESTful web service. The `jsonRequest()` method receives a string URL that references a RESTful web service endpoint that provides weather data. The code begins by instantiating a `java.net.URL` class with a valid URL address with a query string containing a city and state.

Next, a valid URL object is created and a connection is opened by the `openConnection()` method to begin a session. Here, the underlying connection is an instance of an `HttpURLConnection`. The connection object is downcast to expose methods that allow the developer to specify connection settings such as the request method, content type, and character set.

Once the connection object is finished with the proper connection settings, the code invokes the `connect()` method to establish the connection to begin receiving data. Finally, the established connection the JSON data is retrieved by obtaining an input stream object via the connection's `getInputStream()` method.

One-liner: Reading a File into a String

One last bonus trick is the private method `streamToString()`, which is responsible for reading the connection's input stream to be converted into a string for the caller. In the case of retrieving JSON data from a Web response, an input stream will need to be converted to a string. The following code snippet shows how to convert an input stream into a `String` object:

```
/**
 * Quick one liner that delimits on the end of file character and
 * returning the whole input stream as a String.
 * @param inputStream byte input stream.
 * @return String contents of the text file
 */
private String streamToString(InputStream inputStream) {
    String text = new Scanner(inputStream, "UTF-8")
        .useDelimiter("\\\\Z")
        .next();
    return text;
}
```

You will quickly notice the strange one-line statement that “magically” converts an input stream into a `String` object. The `streamToString()` method uses the convenient `Scanner` API. This API is excellent for reading text files that have delimited text lines, such as comma-separated values (CSV). `Scanner` objects default their delimiter as an end-of-line character (`\n`).

By setting the end-of-file (EOF) marker (`\\\\Z`) as the delimiter, the `next()` method will return the text stream before the end, creating entire text field. Of course, there are other ways to retrieve data, and probably more efficient ways, too; however, you can see that the `Scanner` API is capable of handling common situations with small to medium-sized data requests (up to ~3MB). The size limit occurs because the data is converted to a `String` object and, depending on the number of requests, memory resources can be easily consumed—possibly overwhelming the application.

HTTP POST Request

The code to make a RESTful POST request looks like the GET request in the previous example, except that the data submitted will not be specified as query string parameters such as `?param1=1¶m2=555`. Instead, to send a JSON object that is represented as a `String`, a `DataOutputStream` object will need to be created to serialize the data as a string payload for delivery. Listing 8-9 shows an example of a fictitious bank using an HTTP RESTful POST service that creates a new customer bank account.

Listing 8-9. A RESTful POST Request

```
String jsonPayload = "{ name: 'john doe', amount: '5000', xtype: 'new_acct'}";
String urlHost = "https://mybank.org/transaction";

URL url = new URL(urlHost);
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setDoOutput(true);
```

```
connection.setDoInput(true);
connection.setInstanceFollowRedirects(false);
connection.setRequestMethod("POST");
connection.setRequestProperty("Content-Type", " application/json");
connection.setRequestProperty("charset", "utf-8");
connection.setRequestProperty("Content-Length", "" + jsonPayload.getBytes().length);
connection.setUseCaches (false);
byte[] bytes = jsonPayload.getBytes(Charset.forName("UTF-8"));
DataOutputStream outStream = new DataOutputStream(connection.getOutputStream ());
outStream.writeBytes(bytes);
outStream.flush();
outStream.close();
InputStream inStream = connection.getInputStream();
String json = streamToString(inStream); // input stream to string
connection.disconnect();
```

In Listing 8-9 the code invokes both `setDoOutput()` and `setDoInput()` methods on the `connection` object and sets them to true. The `setDoOutput()` method sets up the connection for receiving data coming from the remote web server, and the `setDoInput()` method sets up the client to allow data to be transmitted to the remote web server.

Continuing with the setting of the connection before posting the data, the code invokes the `setInstanceFollowRedirects()` method to prevent the server from changing the URL location of the client (a URL redirect). The rest of the code simply updates the header information before the web request is submitted.

Fetching data is fun; however, let's see how to display content. Now that you've seen various strategies for fetching web-based content, you will finally get a chance to display HTML5 web content using the JavaFX `WebView` node.

Viewing HTML5 Content (`WebView`)

JavaFX provides a GUI `WebView` (`javafx.scene.web.WebView`) node that can render HTML5 content onto the Scene graph. A `WebView` node is basically a mini-browser that is capable of responding to web events and allows a developer to interact with the HTML5 content. Because of the close relationship between loading web content and the ability to display web content, the `WebView` node object also contains a `WebEngine` instance.

In this section I've created an example that uses the `WebView` API to render HTML5 content. The example is a JavaFX application that displays an SVG-based analog clock. To obtain the full source code, visit the book's website to download the `DisplayingHtml5Content` project.

Example: An HTML5 Analog Clock

As an example of using HTML5 content to be rendered onto a `WebView` node, let's look at an analog clock that was created using an SVG file, based on the W3C's scalable vector graphics (SVG) markup language. Figure 8-1 depicts an SVG-based analog clock that is rendered onto a JavaFX `WebView` node.

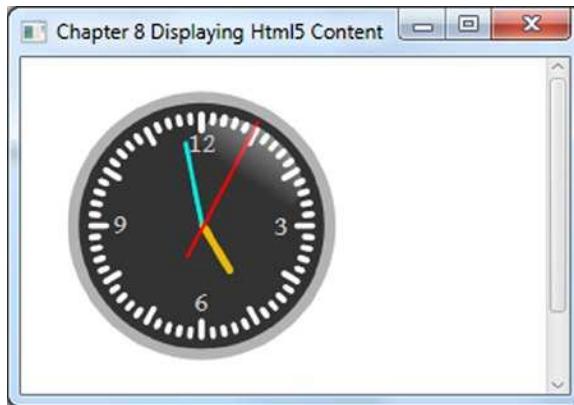


Figure 8-1. SVG-based analog clock (WebView node)

You can see that the analog clock face has hour, minute, and second hands. The clock hands will be initialized and positioned with the current time. Of course, as time goes by the hands will move in a clockwise rotation.

Analog Clock Source Code

Listing 8-10 is the source code for the analog clock application shown in Figure 8-1. The analog clock is based on an SVG file loaded into a JavaFX application's scene graph. Listing 8-11 shows the SVG file containing the majority of the functional code that displays the clock face and the JavaScript code that animates the hour, minute, and second hands.

Listing 8-10. Analog Clock Source Code (*DisplayingHtml5Content.java*)

```
package jfx8ibe;

import java.net.MalformedURLException;
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;

/** An SVG analog clock rendered using the WebView node.
 * @author cdea
 */
public class DisplayingHtml5Content extends Application {

    @Override
    public void start(Stage primaryStage) throws MalformedURLException {
        primaryStage.setTitle("Chapter 8 Displaying Html5 Content");
        WebView browser = new WebView();
        Scene scene = new Scene(browser,320,250, Color.rgb(0, 0, 0, .80));
        primaryStage.setScene(scene);
    }
}
```

```
// The following line is commented out because of
// bug: RT-35757 unable to load svg files URL location inside a jar
//URL url = getClass().getResource("clock.svg");

URL url = getClass().getResource("clock.html");
browser.getEngine().load(url.toExternalForm());
primaryStage.show();
}

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

Listing 8-11. The clock.html (or clock.svg) File, Containing an SVG Representation of an Analog Clock

```
<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="clock.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="#00ffff" 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: shiny glare, black button cover (center)

</svg>

```

How It Works

HTML5 allows the use of scalable vector graphics (SVG) content to be shown in browsers. SVG is similar to JavaFX's Scene graph (retained mode), in which nodes can be scaled at different sizes while preserving shape details without pixelation. In this example the code shown in Listing 8-10 represents the Java source code, and Listing 8-11 represents the SVG content code that is loaded into a WebView node.

Before running the example code, make sure the `clock.html` (or `clock.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 (`clock.html` or `clock.svg`) to the build path. You may also want to copy the `clock.html` (or `clock.svg`) file manually to reside in the build path co-located where the `DisplayHtml5Content.class` file is located if you are running the application on the command line.

Note In the previous edition of this book, the SVG code was in a file with a filename extension of `.svg`; however, in this book the file has been renamed with an extension of `.html`. I did this because of a bug (RT-35757) that was found, which prevents (or prevented) files with the extension of `.svg` from being loaded and rendered onto a `WebView` node. The issue appears to occur when the specified `.svg` file URL points to a file inside a jar file. The workaround to the issue is to rename the file with an extension of `.html`. By the time you read this, the issue may already be resolved.

Let's jump right into the code from Listing 8-10, beginning with the `start()` method. This method begins by setting the Stage window's title bar. Then the code instantiates a `WebView` node as the root when creating the `Scene` object. After creation, the scene is set as the Stage window's current scene.

Before loading the file, the code obtains a valid URL string of the location of the file based on the method `getClass().getResources()`.

When loading a URL you will notice the call to `getEngine().load()`, where the `getEngine()` method will return an instance of a `javafx.scene.web.WebEngine` object. This basically means that all `WebView` classes instantiated will implicitly create their own `javafx.scene.web.WebEngine` instances. The following code snippet is an excerpt from Listing 8-10, showing the JavaFX's `WebEngine` object loading the `clock.html` (or `clock.svg`) file:

```
WebView browser = new WebView();
URL url = getClass().getResource("clock.html");
browser.getEngine().load(url.toExternalForm());
```

You are probably wondering why the Java source code is so small. The code is small because the `WebView` (`javafx.scene.web.WebView`) node is tasked only to render HTML5 content. This means that the majority of the code shown in Listing 8-11 is contained in the `clock.html` (or `clock.svg`) file. This file contains SVG markup language and JavaScript code. Next, let's look at how the SVG content was created.

Inkscape and SVG

The analog clock was created using the popular open source SVG drawing program Inkscape. Because Inkscape is beyond the scope of this book, I will not be discussing the details of the tool. To learn more about Inkscape, please visit <http://www.inkscape.org> for tutorials and demos. To learn more about SVG, visit <http://www.w3schools.com/svg/default.asp>.

In brief, Inkscape allows designers to create shapes, text, and effects to generate illustrations. Because SVG files are considered HTML5 content, these files can also be displayed in an HTML5-capable browser such as the WebView node.

Running the example project called `DisplayingHtml5Content`, you have seen that the clock's hands move or animate by rotating clockwise. You are probably wondering how the clock's hands are rotated or positioned. Simply put, when manipulating SVG elements or any HTML5 elements, the JavaScript language will be employed. In fact in this example the clock's hour, minute, and second hand node elements are rotated by using JavaScript code that exists in Listing 8-11 between the `<script>` tag elements.

Between these `<script>` tags is the JavaScript code responsible for initializing and updating the clock's hands. The main code uses the `updateTime()` function to calculate the position of each second, minute, and hour hand. Before the start of the animation of the clock hands, the code sets 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, the SVG code begins to animate the clock arms by using the `animateTransform` element. The following is an 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 example, visit <http://screencasters.heathenx.org/blog> to learn about all things Inkscape.

Another impressive and beautiful display of custom controls that focuses on clocks, gauges, and dials is the Steel Series and the Enzo library by Gerrit Grunwald. To be totally amazed, visit his blog at <http://harmoniccode.blogspot.com>.

WebEvents

If you've read Chapter 7's discussion of JavaFX Media and how the APIs are designed to be event-driven, you'll be happy to know that the JavaFX Web APIs also adhere to an event-driven programming model. Web events mirror the way browsers raise client side events such as alerts, errors, and statuses. In this section you will briefly learn about how to respond to Web-based events.

If you are familiar with JavaScript client-side web development, you most likely know how to pop up an alert dialog box with a message. If you don't know how to display an alert box, the following JavaScript code snippet will pop up an alert dialog box (assuming you are using a browser):

```
<script>
    alert('JavaFX is Awesome');
</script>
```

Typically, old-school web developers will use alerts to debug web pages. However, modern web developers more likely use the JavaScript function `console.log()` to output information into a console provided by Firebug or the Chrome browser's developer tools. So, what do you think would happen if an HTML web page that contained the `alert()` code just shown was loaded into a `WebView` node?

When the code is executed inside a JavaFX `WebView` node, a native dialog window will not pop up. But the `OnAlert` event does get raised as a `javafx.scene.web.WebEvent` object. Setting up handler code is easy. To set up the handler, use the `setOnAlert()` method with an inbound parameter of type `WebEvent`.

The following code snippet will receive `WebEvent<String>` objects containing the string message to be output onto the console:

```
webView.getEngine().setOnAlert((WebEvent<String> wEvent) -> {
    System.out.println("Alert Event - Message: " + wEvent.getData());
});
```

Developers who want to create their own dialog windows upon an alert-triggered message can simply respond to `WebEvents` and use `Stage` instances. Table 8-1 shows all of the `WebEngine`'s web events.

Table 8-1. `javafx.scene.web.WebEngine` *WebEvents and Properties*

Set On Method	On Method Property	Description
<code>setOnAlert()</code>	<code>onAlertProperty()</code>	JavaScript alert handler
<code>setOnError()</code>	<code>onErrorProperty()</code>	WebEngine error handler
<code>setOnResized()</code>	<code>onResizedProperty()</code>	JavaScript resize handler
<code>setOnStatusChanged()</code>	<code>onStatusChanged()</code>	JavaScript status handler
<code>setOnVisibilityChanged()</code>	<code>onVisibilityChangedProperty()</code>	JavaScript window visibility handler
<code>setConfirmHandler()</code>	<code>confirmHandlerProperty()</code>	JavaScript Confirm window

Now that you know how to retrieve data, manipulate HTML, and respond to browser-based web events, let's look at an example that combines all of the techniques that you've learned so far. This example is a weather widget that fetches JSON data from a RESTful endpoint. You will also see bidirectional communication between Java and JavaScript.

Weather Widget Example

The example demonstrated in this section is a weather widget that combines all of the web-based concepts mentioned so far. The example is a JavaFX application that loads JSON weather information from the Open Weather Map API at <http://openweathermap.org>. Figure 8-2 depicts the Weather Widget example with the current weather conditions of a city.



Figure 8-2. A Weather Widget displaying the current condition

Note The web service in this example is powered by the folks from Open Weather Map (<http://openweathermap.org>). I encourage you to support them as they continue to provide a valuable service with maps and weather data around the world for application and mobile developers.

The Weather Widget allows the user to enter a city or region into the search field to determine the weather. After searching the screen will display the following: city, country, day of week, time, weather condition, Weather description, Icon, temperature, humidity, and wind speed. To obtain the project source, please visit the book's web site to download the code for the NetBeans project WeatherWidget2.

Source Code

The WeatherWidget2 project has two source files. Listing 8-12 contains the JavaFX (Java) application code, and Listing 8-13 contains the HTML and JavaScript code.

Listing 8-12. Source code of the Weather Widget (*WeatherWidget2.java*)

```
package jfx8ibe;

import java.io.*;
import java.net.*;
import java.util.*;
import javafx.application.*;
import javafx.concurrent.*;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
```

```
import javafx.scene.web.*;
import javafx.stage.Stage;
import netscape.javascript.JSObject;

/**
 * The WeatherWidget2 application demonstrates the
 * use of the following:
 * <pre>
 * 1) Communications from Java to JavaScript
 * 2) Communications from JavaScript to Java
 * 3) RESTful GET Web service endpoint
 * 4) JSON Objects
 * 5) Handling HTML/JavaScript WebEvents
 * 6) Debugging using Firebug lite
 * </pre>
 * Weather APIs are from Open Weather Map http://openweathermap.org
 *
 * @author cdea
 */
public class WeatherWidget2 extends Application {

    private WebView webView;
    public static final String WEATHER_URL =
        "http://api.openweathermap.org/data/2.5/weather";
    private String cachedUrl;

    @Override
    public void start(Stage stage) {
        // create the scene
        stage.setTitle("Chapter 8 Weather Widget 2");
        BorderPane root = new BorderPane();
        Scene scene = new Scene(root, 300, 300);
        stage.setScene(scene);

        // WebView node to display local html content
        webView = new WebView();
        root.setCenter(webView);

        // Turns on Firebug lite for debugging
        // html,css, javascript
        //enableFirebug(webView);

        // The web view's web engine
        webView.getEngine()
            .getLoadWorker()
            .stateProperty()
            .addListener( (obs, oldValue, newValue) -> {
                if (newValue == Worker.State.SUCCEEDED) {
                    // Let JavaScript make upcalls to this (Java) class
                    JSObject jsobj = (JSObject)
                        webView.getEngine()
                            .executeScript("window");
                }
            });
    }
}
```

```

        jsobj.setMember("WeatherWidget2", this);
        // default city's weather (a sunny place)
        queryWeatherByLocationAndUnit("Miami,FL", "c");
    }
});

webView.getEngine().setOnAlert((WebEvent<String> t) -> {
    System.out.println("Alert Event - Message: " + t.getData());
});

// load weather_template.html
String htmlFile = streamToString(
    getClass().getResourceAsStream("weather_template.html"));
webView.getEngine().loadContent(htmlFile);
stage.show();
}

/**
 * Quick one-liner that delimits using the end of file character and
 * returns the whole input stream as a String. Use for small files.
 * @param inputStream byte input stream.
 * @return String a file or JSON text
 */
public String streamToString(InputStream inputStream) {
    String text = new Scanner(inputStream, "UTF-8")
        .useDelimiter("\Z")
        .next();
    return text;
}

/** -- C --
 * Called from the JavaScript function findWeatherByLocation()
 * inside the weather_template.html file.
 * @param cityRegion
 * @param unitType
 */
public void queryWeatherByLocationAndUnit(String cityRegion,
    String unitType) {

    String queryString = null;
    queryString = cityRegion;
    String units = "f".equalsIgnoreCase(unitType) ? "imperial": "metric";

    String weatherRequest = WEATHER_URL +
        "?q=" + queryString +
        "&" + "units=" + units +
        "&" + "mode=json";
    // JavaScript code periodically updates the forecast using the
    // setInterval() function by using the cached URL.
    if (null == cityRegion && cachedUrl != null) {

```

```

        weatherRequest = cachedUrl;
    } else {
        cachedUrl = weatherRequest;
    }

    // Spawn background thread to fetch weather info.
    new Thread(createWeatherQueryWorker(weatherRequest,
        unitType)).start();
}

/** -- D --
 * Returns a Task responsible for fetching and populating
 * the html display area.
 * @param weatherRequest the json request
 * @param unitType f is for degrees in fahrenheit and
 *                  c for degrees celsius.
 * @return Task
 */
private Task createWeatherQueryWorker(String weatherRequest,
    String unitType) {

    return new Task() {
        @Override
        protected Object call() throws Exception {
            // On worker thread... blocking call to jsonGetRequest()
            String json = jsonGetRequest(weatherRequest); //

            Platform.runLater(() ->
                // On the JavaFX Application Thread....
                webView.getEngine()
                    .executeScript(
                        "populateWeatherData('" + json + "', " +
                        "'"+ unitType +'"');");
            ); // runLater()
            return true;
        } // call()
    }; // return
}

/** 
 * JSON retrieval using a RESTful GET request.
 * @param urlQueryString
 * @return
 */
public String jsonGetRequest(String urlQueryString) {
    String json = null;
    try {
        System.out.println("Request: " + urlQueryString);
        URL url = new URL(urlQueryString);
        HttpURLConnection connection =
            (HttpURLConnection) url.openConnection();
        connection.setDoOutput(true);

```

```
connection.setInstanceFollowRedirects(false);
connection.setRequestMethod("GET");
connection.setRequestProperty("Content-Type", "application/json");
connection.setRequestProperty("charset", "UTF-8");
connection.connect();
InputStream inStream = connection.getInputStream();
json = streamToString(inStream);
System.out.println("Response: " + json);
} catch (IOException ex) {
    ex.printStackTrace();
}
return json;
}

/**
 * Enables Firebug Lite for debugging a webEngine.
 * @param engine the webEngine for which debugging is to be enabled.
 */
private static void enableFirebug(WebView webView) {
    // Code omitted...
}

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

Listing 8-13. HTML Template File for the Weather Widget (`weather template.html`)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Weather Widget 2</title>
    <style type="text/css">
      body { background-color:#ffffff; }

      .tileTextDisplay {
        font-family: arial,sans-serif;
        font-weight: lighter !important;
        color: #878787 !important;
      }
      .largerFont {
        font-size: x-large !important;
      }
      .mediumFont {
        font-size: medium !important;
      }
      #weather-temp {
        font-family: 'Arial', sans-serif;
        font-size: 64px;
        color: #212121 !important;
      }
    </style>
  </head>
  <body>
    <div class="tileTextDisplay">Temperature:</div>
    <div id="weather-temp">87</div>
  </body>
</html>
```

```
#unitType {  
    font-family: 'Arial', sans-serif;  
    font-size: 20px;  
    color: #000000 !important;  
}  
</style>  
  
<script>  
  
// Every period query and populate the forecast using cached URL  
setInterval(function(){  
    WeatherWidget2.queryWeatherByLocationAndUnit(null, "c");  
,  
(1000 * 60 * 60 * 3) // every 3 hours  
}); // set interval  
  
var weekday=new Array(7);  
weekday[0]="Sunday";  
weekday[1]="Monday";  
weekday[2]="Tuesday";  
weekday[3]="Wednesday";  
weekday[4]="Thursday";  
weekday[5]="Friday";  
weekday[6]="Saturday";  
  
/** -- B --  
 * Makes an upcall to Java code  
 * WeatherWidget2 clas method queryWeatherByLocationAndUnit()  
 */  
function findWeatherByLocation() {  
    var cityInfo = encodeURIComponent(document.getElementById('search-field').value);  
    setInnerText("error-msg", "");  
    WeatherWidget2.queryWeatherByLocationAndUnit(cityInfo, "c");  
}  
  
/** -- E --  
 * Called from Java after data is fetched.  
 */  
function populateWeatherData(weather, unitType) {  
    var jsonWeather = eval("(" + weather + ")");  
    if (jsonWeather.message) {  
        if (jsonWeather.message.indexOf('Error') === 0 ) {  
            document.getElementById('error-msg').innerHTML = jsonWeather.message;  
            alert(jsonWeather.message);  
            return;  
        }  
    }  
  
    setInnerText('city', jsonWeather.name);  
    setInnerText('country', jsonWeather.sys.country);  
    var weatherTime = new Date(jsonWeather.dt);  
    var timeStr = timeFormat(jsonWeather.dt);
```

```

        setInnerText('weather-day-time', weekday[weatherTime.getDay()] + " " + timeStr);
        setInnerText('weather-current', jsonWeather.weather[0].main);
        setInnerText('weather-current-desc', jsonWeather.weather[0].description);
        document.getElementById('weather-icon').src = "http://openweathermap.org/img/w/" +
jsonWeather.weather[0].icon + ".png";
        setInnerText('weather-temp', jsonWeather.main.temp);
        setInnerText('weather-humidity', "Humidity: " + jsonWeather.main.humidity + "%");
        var windSpeed = (unitType === 'f') ? 'mph' : 'km/h';
        setInnerText('weather-wind-speed', "Wind: " + jsonWeather.wind.speed + " " +windSpeed);

    }

function setInnerText(id, text) {
    document.getElementById(id).innerText = text;
}

function timeFormat( millis ) {
    var weatherTime = new Date(millis);
    var hours = weatherTime.getHours();
    var minutes = weatherTime.getMinutes();
    var meridian = hours >=12 ? 'PM' : 'AM';
    hours = hours % 12;
    hours = hours ? hours : 12; // hour '0' means '12'
    minutes = minutes < 10 ? '0' + minutes : minutes;
    var timeStr = hours + ':' + minutes + ' ' + meridian;
    return timeStr;
}
</script>
</head>
<body id="weather_background">
<form>
    <input id="search-field" placeholder="Enter city, state/country" type="text" name="searchField"/>
    <!-- A -->
    <input id="search-button" type="button" onclick="findWeatherByLocation()" name="searchButton" value="Search"/>
</form>
<div id="error-msg"></div>
<div id="city" class="tileTextDisplay largerFont"></div>
<div id="country" class="tileTextDisplay mediumFont"></div>
<div id="weather-day-time" class="tileTextDisplay mediumFont"></div>
<div id="weather-current" class="tileTextDisplay mediumFont"></div>
<div id="weather-current-desc" class="tileTextDisplay mediumFont"></div>
<div>
    <img id="weather-icon" style="float:left;height:64px;width:64px" src="" alt=""/>
    <div id="weather-temp" style="padding-left:10px; float:left;"></div>
    <div style="float:left; font-size:20px; margin-top:6px">
        <span style="display:inline">°<span id="unitType">C</span></span>
    </div>
</div>
<div style="clear:both;"></div>
<div id="weather-humidity" class="tileTextDisplay mediumFont"></div>
<div id="weather-wind-speed" class="tileTextDisplay mediumFont"></div>
</html>

```

How It Works

The code is divided into two parts, the Java code shown in Listing 8-12 and HTML/JavaScript code shown in Listing 8-12. The purpose of this example is to demonstrate bidirectional communication between Java and JavaScript while providing RESTful HTTP GET of JSON data. Because all of its aspects, from retrieving data and manipulating HTML to using the JavaScript bridge, have already been discussed, I will discuss only the high-level steps of the start() method of the JavaFX WeatherWidget project from Listing 8-13.

The following are the steps taken in the start() method of Listing 8-12.

1. Set up the scene and stage window with a BorderPane as the root.
2. Create a WebView node to display html content.
3. Create handler code that responds when the WebView's content is finished loading.
 - a. Pass the application into the web engine to allow upcalls from JavaScript code.

```
// Let JavaScript make upcalls to this (Java) class
JSObject jsobj = (JSObject) webView.getEngine()
    .executeScript("window");
jsobj.setMember("WeatherWidget2", this);
```

- b. Default to the weather forecast of Miami, Florida, by fetching JSON weather data to be populated into the HTML page. The lowercase "c" is for the unit type, degrees Celsius.
queryWeatherByLocationAndUnit("Miami,FL", "c");
4. Create handler code for the OnAlert property on the WebView's web engine. The handler code responds to HTML's JavaScript alert() function (WebEvent).

```
webView.getEngine().setOnAlert((WebEvent<String> t) ->
    System.out.println("Alert Event - Message: " + t.getData()));
```

5. Load the local HTML file weather_template.html to be loaded and displayed onto the WebView.

```
// load weather_template.html
String htmlFile = streamToString(
    getClass().getResourceAsStream("weather_template.html"));
webView.getEngine().loadContent(htmlFile);
```

In Listings 8-12 and 8-13, notice the letters A to E within the comments; these show the application flow path from searching the weather for a city and region to finally displaying the data in HTML. The following are the steps from A to E:

Step A: The button click invokes the JavaScript function `findWeatherByLocation()`:

```
<input id="search-button" type="button" onclick="findWeatherByLocation()" name="searchButton" value="Search"/>
```

Step B: The upcall to the actual Java code method `queryWeatherByLocationAndUnit()`:

```
function findWeatherByLocation()
```

Step C: The Java method that uses background thread (Task) to make a RESTful GET to retrieve weather data.:

```
public void queryWeatherByLocationAndUnit(String cityRegion, String unitType)
```

Step D: After the JSON data is fetched, an `executeScript()` method calls into the JavaScript function `populateWeatherData()`:

```
String json = jsonGetRequest(weatherRequest);

Platform.runLater(() ->
    // On the JavaFX Application Thread....
    webView.getEngine()
        .executeScript(
            "populateWeatherData('" + json + "', '" +
            "'" + unitType + "' );")
); // runLater()
```

Step E: The HTML is populated with the JSON data representing the weather:

```
function populateWeatherData(weather, unitType)
```

One last feature of the Weather Widget example to highlight is its ability to update the weather forecast periodically, every three hours. (According to Open Weather Map, their weather forecast updates are typically every three hours.) In Listing 8-13, notice the JavaScript code function call to `setInterval()` located at the beginning of the `<script>` element tag. By making the upcall to the Java method `queryWeatherByLocationAndUnit()` and passing in a null value, the query will actually use the previous (cached) URL. The following code snippet is the `setInterval()` function implementation from Listing 8-13 that periodically queries and populates the weather forecast.

```
// Every period query and populate the forecast using cached URL
setInterval(function(){
    WeatherWidget2.queryWeatherByLocationAndUnit(null, "c");
},
(1000 * 60 * 60 * 3) // every 3 hours
); // set interval
```

Summary

In this chapter you learned about the WebEngine API for retrieving Web HTML5 content. You began by learning two ways to load web content using the WebEngine's `load()` and `loadContent()` methods. After learning about how to load web content, you saw how to obtain raw content such as a DOM XML or XML as a String object. Next, you learned about the powerful JavaScript bridge that allows bidirectional communication between Java and JavaScript. Getting familiar with web development concepts, you also learned about some of the limitations to JavaFX's WebEngine API. Within these limitations on making RESTful web service calls, you got a chance to learn about how to make RESTful GET and POST requests. After learning the many ways to retrieve data and manipulate HTML, you were able to render HTML5 content by running a small example of an SVG-based analog clock. Finally, you saw all the concepts combined into the last example of a weather widget; this widget made calls to retrieve JSON-formatted weather data to be displayed using HTML5.

JavaFX 3D

JavaFX 2 supported transformation of 2D GUI components into a third visual dimension via a rotation in the z-axis. However, only now with the advent of JavaFX 8 can you realize a true 3D scene with adjustable lights, camera, and models as first-class citizens of the language. With JavaFX 8 it is possible to combine these capabilities directly with the better-known 2D GUI components discussed in this book, both visually and in source code. The advent of dedicated GPU hardware has lowered the barrier to specific language support for 3D rendering so that the performance in JavaFX is on a par with third-party implementations of previous versions of Java.

A small book could be dedicated to all the capabilities of the 3D graphics tools within JavaFX 8. For the sake of introductory and intermediate learners, this chapter focuses on quickly establishing scenes with true three-dimensional visualizations of simple objects. These concepts are then extended by describing typical customizations that would be found in more complex scenes.

In this chapter you will learn the following concepts and techniques:

- Geometry primitives
- Cameras and lights
- Custom `TriangleMesh` models
- Event handling and interacting with 3D scenes

Basic 3D Scenes in JavaFX

For this half of the 3D chapter we will use a continuously developed example. The following sections will take you through the steps from establishing a basic empty 3D scene to creating a scene with objects that can be viewed from different distances and angles.

A Very Basic 3D Scene Example

The best part about the new JavaFX 3D support is that 3D objects and rendering are now fully integrated GUI features of the Java language. Care was taken in designing the 3D packages such that there are very few changes in paradigm from their 2D counterparts. The difference in establishing a basic 3D scene is merely in the camera. Listing 9-1 demonstrates this.

Listing 9-1. Initializing a 3D Scene with Camera

```
/* From SimpleScene3D.java */
package jfx8ibe;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.PerspectiveCamera;
import javafx.scene.Scene;
import javafx.scene.input.KeyCode;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.DrawMode;
import javafx.scene.shape.Sphere;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;

public class SimpleScene3D extends Application {

    private PerspectiveCamera camera;
    private final double cameraModifier = 50.0;
    private final double cameraQuantity = 10.0;
    private final double sceneWidth = 600;
    private final double sceneHeight = 600;
    private double mouseXold = 0;
    private double mouseYold = 0;
    private final double cameraYlimit = 15;
    private final double rotateModifier = 25;

    @Override
    public void start(Stage primaryStage) {

        //Step 1a: Build your Scene and Camera

        Group sceneRoot = new Group();
        Scene scene = new Scene(sceneRoot, sceneWidth, sceneHeight);
        scene.setFill(Color.BLACK);
        PerspectiveCamera camera = new PerspectiveCamera(true);
        camera.setNearClip(0.1);
        camera.setFarClip(10000.0);
        camera.setTranslateZ(-1000);
        scene.setCamera(camera);
    }
}
```

```
primaryStage.setTitle("SimpleScene3D");
primaryStage.setScene(scene);
primaryStage.show();    }

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

This code will give you a full 3D scene. Running the application, however, will show nothing but an empty black screen. An empty scene is not very exciting, so we will add some objects soon. But this example demonstrates that any 3D scene differs from a 2D layout by only two lines:

```
PerspectiveCamera camera = new PerspectiveCamera(true);
scene.setCamera(camera);
```

Creating a 3D camera node such as a `PerspectiveCamera` actually creates a separate scene graph node, much in the same way you might create a `BorderPane` or `Button`. In that sense it must be added to the scene like any other node. However, the camera node is special in that it is added via the `scene.setCamera()` method. The Boolean parameter for the `PerspectiveCamera` constructor determines where the 3D origin is. Setting this to `true` will establish all 3D layouts from the origin of `(0,0,0)` so that all axes cross in the very center of the screen. This is typical for most 3D scenes you would expect to build using JavaFX. Setting this parameter to `false` establishes the 3D origin at the upper-left corner, similar to that of a 2D component layout. For most scenes `PerspectiveCamera(true)` is ideal; however, some scenes, especially scientific data displays, are easier with an upper-left origin.

Note A good rule of thumb for selecting the `camera` class is that if you plan on translating and rotating the camera around the scene much, you should use `PerspectiveCamera(true)`.

Primitives

Like most 3D graphics libraries, JavaFX 8 comes with several out-of-the-box geometric shapes that can be added to a scene. These shapes are commonly referred to as *primitives* and can be added to a scene in the same way that any traditional `Node` type can be added.

Adding a Primitive Example

Now let us extend the previous very simple scene example so that we actually see something. To do this we will create a `Cylinder` object, give it a colored material, and add it to the scene. Listing 9-2 is the code that demonstrates this.

Listing 9-2. Adding a Cylinder Primitive to a JavaFX 3D Scene

```
/* SimpleScene3D.java */

//Step 1b: Add a primitive
final Cylinder cylinder = new Cylinder(50, 100);
final PhongMaterial blueMaterial = new PhongMaterial();
blueMaterial.setDiffuseColor(Color.DARKBLUE);
blueMaterial.setSpecularColor(Color.BLUE);
cylinder.setMaterial(blueMaterial);
sceneRoot.getChildren().add(cylinder);
//End Step 1b
```

Adding this code to our previous example will add the cylinder and set its color. Recompile and run the application, and you should see a view like Figure 9-1.

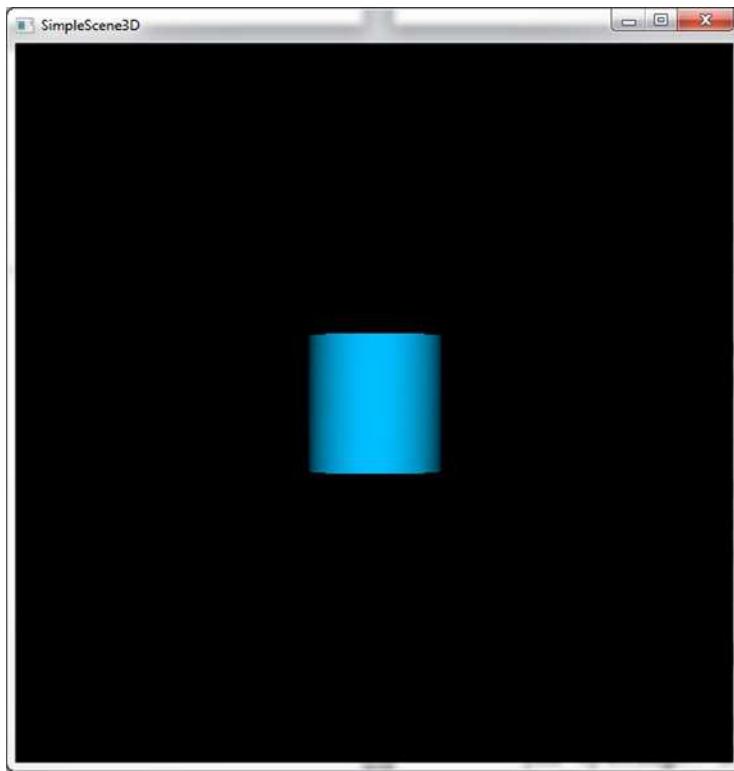


Figure 9-1. Cylinder with PhongMaterial and DrawMode.FILL

This is better than an empty screen. The Cylinder constructor used here:

```
final Cylinder cylinder = new Cylinder(50, 100);
```

specifies the diameter and height of the shape. To achieve that deep color blue and fancy specular highlight we used this statement:

```
final PhongMaterial blueMaterial = new PhongMaterial();
```

PhongMaterial is an extension of the javafx.paint hierarchy and allow for a rich, smooth interpolation over 3D surfaces. A PhongMaterial object can be reused between primitives and other 3D nodes as described later. If you do not want to use a PhongMaterial, the default rendering for a primitive or any Shape3D object is a wireframe view. This will be demonstrated in later examples. Once your shape is constructed and colored you can add it to your scene with a statement like this:

```
sceneRoot.getChildren().add(cylinder);
```

Here the Cylinder primitive is treated like any other graphic node you might use or create. The Cylinder primitive is added to the scene by default at the origin, or (0,0,0).

Simple Translate and Rotate Example

Now that we have something to look at, we can utilize the third dimension by transforming it. JavaFX 3D supports all the standard 3D transformations you would expect with easy-to-use classes. The following transformations, subclassed from the Transform class, all support 3D nodes and groups:

- Affine
- Rotate
- Scale
- Shear
- Translate

Typically, in any 3D scene you would be most interested in rotation and translation, which are most concerned with moving and changing the orientation of objects, lights, and the camera. JavaFX Shape3D objects, such as the Cylinder class, provide setter methods that let you easily apply a simple rotation or translation. Listing 9-3 demonstrates how to use these methods with the Cylinder we have in our scene.

Listing 9-3. Translating and Rotating a Primitive into Position

```
/* SimpleScene3D.java */

//Step 1c: Translate and Rotate primitive into position
cylinder.setRotationAxis(Rotate.X_AXIS);
cylinder.setRotate(45);
cylinder.setTranslateZ(-200);

//End Step 1c
```

Adding the code from Listing 9-3 to our current 3D scene application will generate a view where the cylinder has been rotated and translated toward the camera. It should be similar to Figure 9-2.

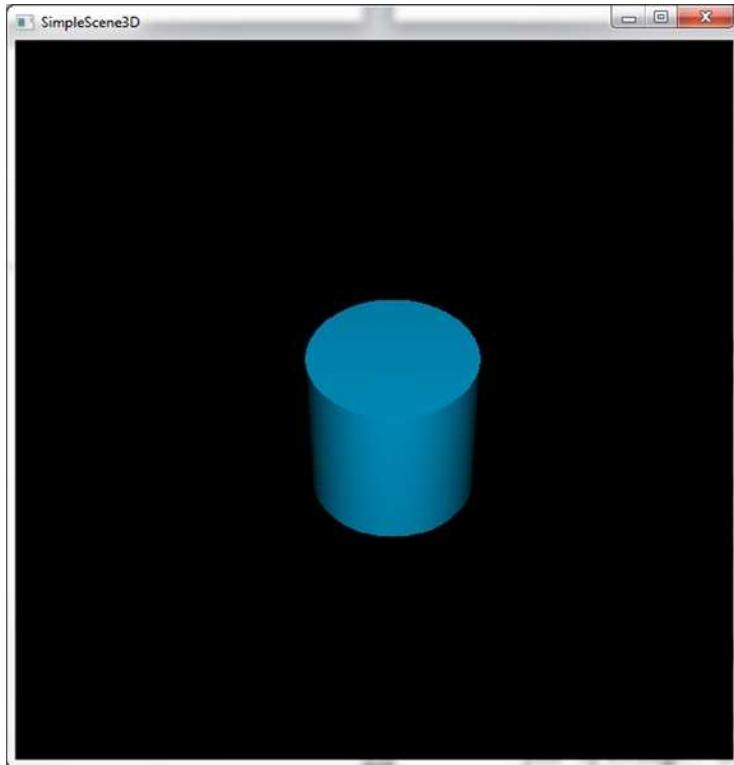


Figure 9-2. A cylinder rotated 45 degrees on the x-axis

For the first time in our running example, we have demonstrated the third dimension using a rotation in the x-axis and a translation in the z-axis. Rotations are set in degrees, with a range of 0 to 360 degrees. It is important to first set the current rotation axis using `setRotationAxis()`. If you need multiple rotations in multiple axes, you will need to call the `setRotationAxis()` method each time the axis changes. Each call to `setRotationAxis()` affects only the current `Shape3D` object. Translations are set in pixels along the given axis of the `set` command and are handled in the same way that traditional 2D components are translated.

Note The coordinate system for a JavaFX 3D scene uses a “y pointing down” approach, which means that the y-axis is positive down. As with most 3D systems, the z-axis is positive toward the viewer, and the x-axis is positive to the right.

Multiple Primitive Transformation Example

This example will expand on our current setup by adding and transforming Cube and Sphere primitives in the same scene as our Cylinder. The code is provided in Listing 9-4.

Listing 9-4. Adding and Transforming Box and Sphere Primitives

```
/* SimpleScene3D.java */

//Step 1d: Add and Transform more primitives
final PhongMaterial greenMaterial = new PhongMaterial();
greenMaterial.setDiffuseColor(Color.DARKGREEN);
greenMaterial.setSpecularColor(Color.GREEN);
final Box cube = new Box(50, 50, 50);
cube.setMaterial(greenMaterial);
final PhongMaterial redMaterial = new PhongMaterial();
redMaterial.setDiffuseColor(Color.DARKRED);
redMaterial.setSpecularColor(Color.RED);
final Sphere sphere = new Sphere(50);
sphere.setMaterial(redMaterial);

cube.setRotationAxis(Rotate.Y_AXIS);
cube.setRotate(45);
cube.setTranslateX(-150);
cube.setTranslateY(-150);
cube.setTranslateZ(150);

sphere.setTranslateX(150);
sphere.setTranslateY(150);
sphere.setTranslateZ(-150);
sceneRoot.getChildren().addAll(cylinder,cube,sphere);
//End Step 1d
```

This code snippet shows multiple materials and colors being applied to multiple primitives. Each primitive is translated and rotated to different depths and axes of our 3D scene. Adding this code snippet to our example should provide a view similar to Figure 9-3.

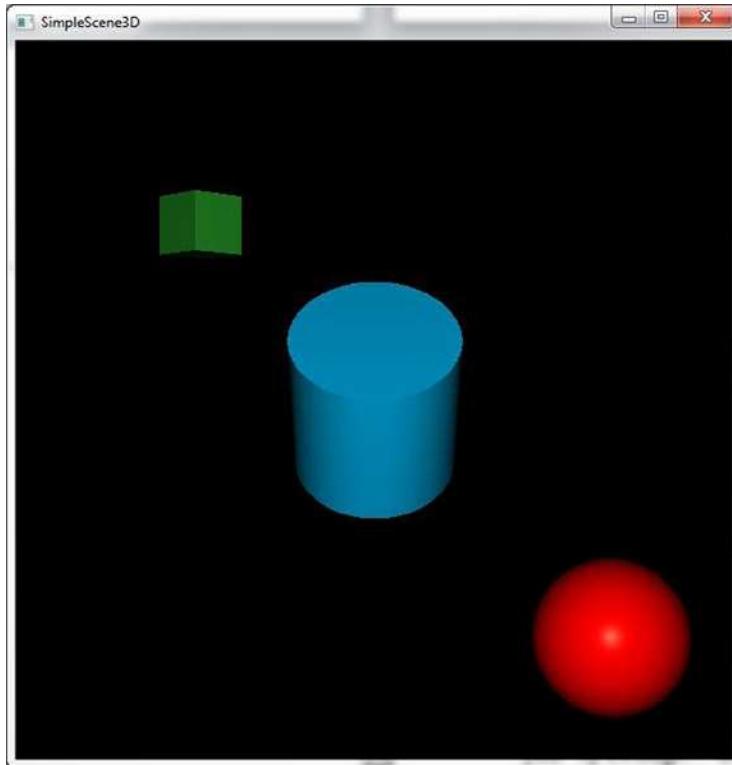


Figure 9-3. Cube, cylinder, and sphere primitives

Having multiple `Shape3D` objects in the same scene demonstrates a great visual depth. However, as you have added multiple `Nodes` or `Groups` to your 3D scene, you can change your method call from `add()` to `addAll()` as in the following line from Listing 9-4:

```
sceneRoot.getChildren().addAll(cylinder,cube,sphere);
```

In that sense, there is no difference between adding 2D GUI components as `Scene` children and the way you handle them here in 3D.

All Together Now: Grouped Primitives

As your scenes become more complicated, it will become cumbersome to add each 3D object to your scene. Often primitives and other `Shape3D` nodes are used in conjunction to form more complex objects. These composite objects are typically transformed as one, though each component is often visualized independently. To do this, you simply use a `Group` as demonstrated in Listing 9-5.

Listing 9-5. Grouping and Transforming Primitives

```
/* SimpleScene3D.java */

    //Step 1e: All Together Now: Grouped Primitives
Group primitiveGroup = new Group(cylinder,cube,sphere);
primitiveGroup.setRotationAxis(Rotate.Z_AXIS);
primitiveGroup.setRotate(180); //Rotate the Group as a whole
sceneRoot.getChildren().addAll(primitiveGroup);
//End Step 1e
```

To add each primitive to the Group object, use the following constructor:

```
Group primitiveGroup = new Group(cylinder,cube,sphere)
```

Many bulk operations can be simplified. Listing 9-5 demonstrates a rotation about the z-axis of 180 degrees, which effectively rotates the entire scene as one homogenous unit. Adding the new code to our running Primitives example should provide a view like Figure 9-4.

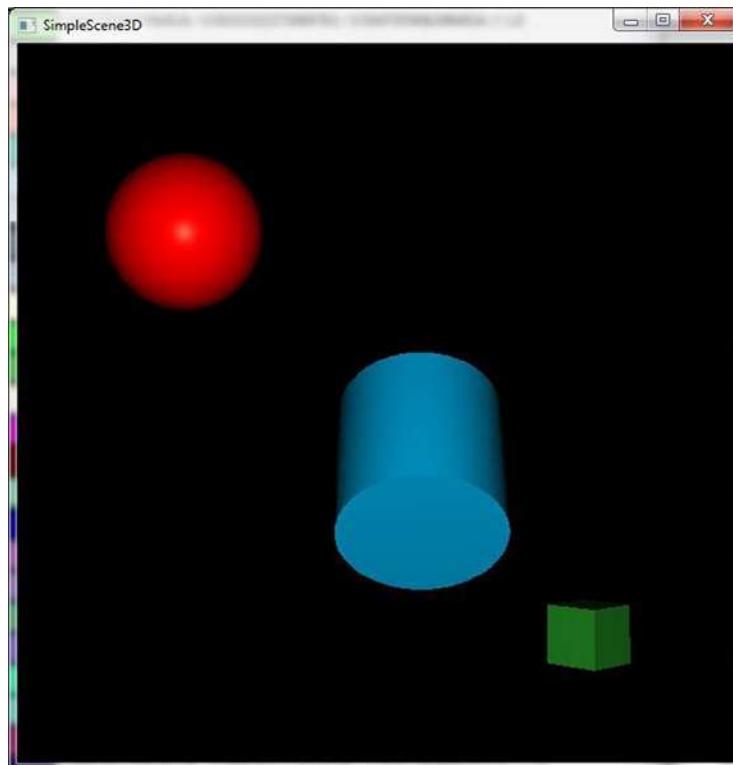


Figure 9-4. Grouped primitives rotated uniformly

Be sure to add the new Group object to your scene instead of your individual primitives, as noted in Listing 9-5. Individual Primitive objects can still be accessed, rotated, and translated independently via the Group object's `getChildren()` method or by direct reference.

Note When adding primitives, groups, and other Shape3D Node objects to a 3D scene, remember that the default is the origin. Most primitives are centered on the Origin by a calculated center of mass.

Interacting with Your Scene

Rendering objects into 3D scenes is really only useful if you can interact with that scene. At a minimum the user should have the ability to move through the scene and do a little sight-seeing. This section of the chapter will walk through an example of a simple First-person Perspective set of controls. The Primitive example from the previous section will be used as a platform to build from.

Primitive Picking for Primitives

To let the user interact directly with the individual objects of a 3D scene, typically with a mouse, your code needs to detect mouse events and whether those events are related to something within your scene. The good news is that event-handling within a 3D scene is established in the same pattern as a traditional 2D GUI. Listing 9-6 demonstrates the code to add a `MouseClicked` event handler.

Listing 9-6. Translating a Primitive by Mouse Event

```
/* SimpleScene3D.java */

//Step 2a: Primitive Picking for Primitives
scene.setOnMouseClicked(event-> {
    Node picked = event.getPickResult().getIntersectedNode();
    if(null != picked) {
        double scalar = 2;
        if(picked.getScaleX() > 1)
            scalar = 1;
        picked.setScaleX(scalar);
        picked.setScaleY(scalar);
        picked.setScaleZ(scalar);
    }
});
//End Step 2a
```

Adding this event handler to the scene in our running example creates an interaction where clicking a given 3D object will either double its size or reduce it back to normal. Clicking once on each primitive should provide a view similar to Figure 9-5.

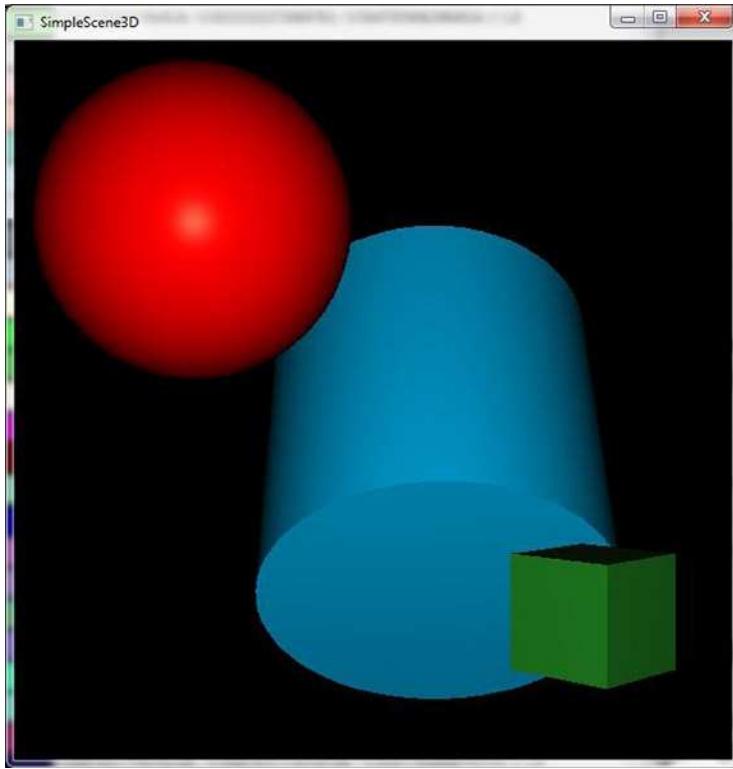


Figure 9-5. 3D objects scaled up in all three axes

The magic here is in determining whether the MouseEvent occurred in congruence with a 3D object in the scene. The magic line of code is this:

```
Node picked = event.getPickResult().getIntersectedNode();
```

where the MouseEvent returns a PickResult. In 3D-speak, a PickResult is a test of intersection of x,y screen coordinates onto x,y,Z 3D Scene coordinates. While PickResults were available in JavaFX 2 for 2D GUI components, JavaFX 8 makes them functional not only in the 3D space but as simple accessible objects from various events. As with all MouseEvent types, the event can be queried for a PickResult. If a PickResult occurred, then the getIntersectedNode() will return the Node object representation of the scene object picked. If no object was picked, getIntersectedNode() returns null.

Note Be sure to scale objects equally in all three axes, as shown in Listing 9-6. Otherwise, your object will quickly become stretched in odd ways.

First-Person Movement Using the Keyboard

A convenient way to navigate through a 3D scene is through a first-person perspective where you “fly through” the scene. Often this is done through a combination of mouse and keyboard, where the keyboard navigates the x- and z-axes and the mouse directs the camera to rotate. This is a typical arrangement for simulations and tactical displays as well as games.

To add keyboard navigation through a 3D scene you simply add the standard JavaFX key event handlers. The code in Listing 9-7 is a simple example of an `OnKeyPressed` event handler that provides the x- and z-axis navigation just described. The movement produced by this handler actually translates the `Camera` node itself through the 3D scene.

Listing 9-7. A Simple First-Person Movement Keyboard Event Handler

```
/* SimpleScene3D.java */
    //Step 2b: Add a Movement Keyboard Handler
    scene.setOnKeyPressed(event -> {
        double change = cameraQuantity;
        //Add shift modifier to simulate "Running Speed"
        if(event.isShiftDown()) { change = cameraModifier; }
        //What key did the user press?
        KeyCode keycode = event.getCode();
        //Step 2c: Add Zoom controls
        if(keycode == KeyCode.W) { camera.setTranslateZ(camera.getTranslateZ() + change); }
        if(keycode == KeyCode.S) { camera.setTranslateZ(camera.getTranslateZ() - change); }
        //Step 2d: Add Strafe controls
        if(keycode == KeyCode.A) { camera.setTranslateX(camera.getTranslateX() - change); }
        if(keycode == KeyCode.D) { camera.setTranslateX(camera.getTranslateX() + change); }
    });
    //End Step 2b-d
```

Breaking down this code snippet, we see that JavaFX makes adding the event handler to your scene very simple and straightforward. It is the same as adding an event handler to any traditional 2D GUI component, as demonstrated with the statement

```
scene.setOnKeyPressed(event -> {
```

Adding composite event handling (holding the Shift key while pressing another key) is simple via the event object itself. In Listing 9-7 we do this with `event.isShiftDown()`, which returns a simple Boolean. These additional Meta keys (like the Shift key) are either true or false regardless of what actual event occurred and can be utilized whether the event is keyboard, mouse or even touch.

To determine which way to move, the camera simply acquires the `KeyCode` from the event:

```
KeyCode keycode = event.getCode();
```

and branches logically on a simple equality check:

```
if(keycode == KeyCode.W) {
    camera.setTranslateZ(camera.getTranslateZ() + change);
}
```

`KeyCode.W` is part of the `KeyCode` enum, which provides every possible key with a `Comparable` interface. Very convenient for building keyboard interfaces to any application. In this example, the proximate keys `{W}`, `{S}`, `{A}`, and `{D}` move the camera perspective forward, backward, left and right, respectively.

First-Person Camera Movement Using the Mouse

The previous Keyboard Handler example provided half of a first-person perspective, in which you “fly through” your 3D scene. The keyboard is used to slide along the `x`- and `z`-axes. What is actually happening is that the `PerspectiveCamera` node is translating along those axes. However, the previous example limits these translations to a single plane. In order to simulate looking around (the “Head on a Swivel” effect), we need to use the mouse.

To add mouse control of the camera swivel for a 3D scene, you simply add a JavaFX `MouseMoved` event handler. The code in Listing 9-8 is a rudimentary example of an `OnMouseMoved` event handler and some basic trigonometry that swivels the camera node. The movement produced by this handler actually rotates the `Camera` node itself on the `x`- and `y`-axes.

Listing 9-8. A Simple First-Person Camera Rotate Control MouseEvent Handler

```
/* SimpleScene3D.java */

//Step 3: Add a Camera Control Mouse Event handler
Rotate xRotate = new Rotate(0,0,0, Rotate.X_AXIS);
Rotate yRotate = new Rotate(0,0,0, Rotate.Y_AXIS);
camera.getTransforms().addAll(xRotate,yRotate);
scene.addEventHandler(MouseEvent.ANY, event -> {
    if (event.getEventType() == MouseEvent.MOUSE_PRESSED ||
        event.getEventType() == MouseEvent.MOUSE_DRAGGED) {
        //acquire the new Mouse coordinates from the recent event
        double mouseXnew = event.getSceneX();
        double mouseYnew = event.getSceneY();
        if (event.getEventType() == MouseEvent.MOUSE_DRAGGED) {
            //calculate the rotational change of the camera pitch
            double pitchRotate =xRotate.getAngle()+(mouseYnew - mouseYold) / rotateModifier;
            //set min/max camera pitch to prevent camera flipping
            pitchRotate = pitchRotate > cameraYlimit ? cameraYlimit : pitchRotate;
            pitchRotate = pitchRotate < -cameraYlimit ? -cameraYlimit : pitchRotate;
            //replace the old camera pitch rotation with the new one.
            xRotate.setAngle(pitchRotate);
            //calculate the rotational change of the camera yaw
            double yawRotate=yRotate.getAngle()-(mouseXnew - mouseXold) / rotateModifier;
            yRotate.setAngle(yawRotate);
        }
        mouseXold = mouseXnew;
        mouseYold = mouseYnew;
    }
});
//End Step 3
```

This code snippet is by far the single largest addition made so far to our running example; however, most of it is needed to facilitate a smooth swivel for the Camera node. Additionally, these rotations are applied in a different way than we've done previously. Before covering the details of how to rotate the camera, we will first examine the mouse event handling itself:

```
scene.addEventHandler(MouseEvent.ANY, event -> {
```

Again, JavaFX makes adding a MouseEvent handler to the scene very simple and straightforward. As with all input events, we can acquire the x and y coordinates from the MouseEvent. Our goal here is to capture any mouse event, whether pressing the buttons or moving the mouse. We have provided code to handle either situation here in our handler. For a user to move the camera, they must first press either of the mouse buttons. Doing so updates a private variable storing the latest x and y coordinates of the mouse. The MouseEvent.MOUSE_DRAGGED event is special in that it requires an actively held mouse button. If this logic is reached, a simple conversion of the coordinate deltas to an angular change is implemented. Again this is a very rudimentary method for performing camera movement in a simple first-person perspective. Most high-performance first-person games and interactive environments use more exotic calculations and include the many corner cases that tend to occur in use. One of these corner cases is vertical rotation beyond 180 degrees. Left alone, the camera would spin without limit vertically, an effect sometimes referred to as "somersaults" or "camera whip." In Listing 9-8 the following lines effectively limit camera rotation on the x-axis, which visually appears to be the y-axis:

```
pitchRotate = pitchRotate > cameraYlimit ? cameraYlimit : pitchRotate;
pitchRotate = pitchRotate < -cameraYlimit ? -cameraYlimit : pitchRotate;
```

Note Remember that what a camera "sees" is opposite from the way the camera is positioned within the scene. For example, "looking up" along the y-axis is actually a rotation on the x-axis.

Finally, when dealing with multiple repeated transformations, especially rotations, a better way to apply these transformations to an object is using the object's `getTransforms().addAll()` method as shown here:

```
camera.getTransforms().addAll(xRotate,yRotate);
```

This will guarantee that all your rotate, translate, and scale transformations are applied as a composite transformation. If you were to call `setRotate` or `setTranslate` multiple times for each rotation, each subsequent `setRotate` call would replace the previous. The end result would be a rotation or translation that is equal only to the last transformation called in sequence.

Beyond the Basics

In this half of the 3D chapter we will develop a new example, focused on ways to provide custom renderings and interactions. Some of the concepts discussed in the first half will be built upon. We will explain `TriangleMesh` objects, including how to "wind" a 3D object using triangles and how to add a `TriangleMesh` object to a 3D scene. You will then learn how to convert the scene to reusable code, which can allow for many customizations that could be the start of your own library of 3D primitives. Finally, we will explore an alternative camera arrangement that is suitable for examining centralized objects.

Custom 3D Objects using the TriangleMesh class

Sometimes the prepackaged 3D primitives provided by JavaFX 8 aren't enough to build the scene you want. It is true that many real-world objects could be represented by a composite of these primitives using the Group type and some clever transformation. However, using only the base primitives would be cartoonish and very cumbersome to maintain within a code base. Most practical applications require custom shapes and models, and rendering must be fast. The good news is that JavaFX 8 provides a means to do this using the TriangleMesh class. The TriangleMesh class is a Shape3D class just like Box, Cylinder, and Sphere, which provides direct access to the points (vertices), faces, and texture coordinate collections that make up the rendered geometry. Even the Box, Cylinder, and Sphere are composed by a TriangleMesh where all points and faces are computed based on relevant parameters. For the following custom TriangleMesh example we will use the previous example as a starting point, as shown in Listing 9-9.

Listing 9-9. Initializing a 3D Scene with Camera

```
/* From TriangleMeshes.java */
package jfx8ibe;

import javafx.application.Application;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.scene.AmbientLight;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.CullFace;
import javafx.scene.shape.DrawMode;
import javafx.scene.shape.MeshView;
import javafx.scene.shape.TriangleMesh;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;

public class TriangleMeshes extends Application {

    private PerspectiveCamera camera;
    private final double sceneWidth = 600;
    private final double sceneHeight = 600;
    private double scenex, sceney = 0;
    private double fixedXAngle, fixedYAngle = 0;
    private final DoubleProperty angleX = new SimpleDoubleProperty(0);
    private final DoubleProperty angleY = new SimpleDoubleProperty(0);
```

```

@Override
public void start(Stage primaryStage) {
    //Step 1: Build your Scene and Camera
    Group sceneRoot = new Group();
    Scene scene = new Scene(sceneRoot, sceneWidth, sceneHeight);
    scene.setFill(Color.BLACK);
    PerspectiveCamera camera = new PerspectiveCamera(true);
    camera.setNearClip(0.1);
    camera.setFarClip(10000.0);
    camera.setTranslateZ(-1000);
    scene.setCamera(camera);

    primaryStage.setTitle("TriangleMeshes");
    primaryStage.setScene(scene);
    primaryStage.show();
    //End Step 1
}
public static void main(String[] args) {
    launch(args);
}
}
}

```

“Winding” and Wuthering

For our custom object we will build and render something that is geometrically simple like a square pyramid. This shape has five faces: four equilateral triangle faces and a square base. Its simplicity will allow us to focus on the `TriangleMesh` interface rather than dealing with difficult calculations to compose complex objects. The goal of a `TriangleMesh` is to define the process by which a series of triangles are rendered so as to give the illusion of a continuous 3D surface. Each triangle is connected and defined by a set of vertices in the 3D scene. The method is called “winding” because when defining the vertices of each triangle, the order in which the vertices are declared indicates to the camera when and how to render the face of that triangle. To explain the process of constructing and viewing a `TriangleMesh` object, I will describe a method that uses parameters to construct a square pyramid. The complete method is shown in Listing 9-10.

Listing 9-10. A Reusable Method for Building a Custom Pyramid `TriangleMesh`

```

/* From TriangleMeshes.java */

//Step 2a: Create a general Pyramid TriangleMesh building method with height and hypotenuse
private Group buildPyramid(float height, float hypotenuse,
                           Color color,
                           boolean ambient,
                           boolean fill) {
    final TriangleMesh mesh = new TriangleMesh();
    //End Step 2a
    //Step 2b: Add 5 points, later we will build our faces from these
    mesh.getPoints().addAll(
        0,0,0,          //Point 0: Top of Pyramid
        0,height,-hypotenuse/2, //Point 1: closest base point to camera
        -hypotenuse/2,height,0, //Point 2: leftmost base point to camera
        hypotenuse/2,height,0, //Point 3: farthest base point to camera
        0,height,hypotenuse/2 //Point 4: rightmost base point to camera
    );
}
}

```

```

); //End Step 2b
//Step 2c:
//for now we'll just make an empty texCoordinate group
mesh.getTexCoords().addAll(0,0);
//End Step 2c
//Step 2d: Add the faces "winding" the points generally counter clock wise
mesh.getFaces().addAll( //use dummy texCoords
    0,0,2,0,1,0, // Vertical Faces "wind" counter clockwise
    0,0,1,0,3,0, // Vertical Faces "wind" counter clockwise
    0,0,3,0,4,0, // Vertical Faces "wind" counter clockwise
    0,0,4,0,2,0, // Vertical Faces "wind" counter clockwise
    4,0,1,0,2,0, // Base Triangle 1 "wind" clockwise because camera has rotated
    4,0,3,0,1,0 // Base Triangle 2 "wind" clockwise because camera has rotated
); //End Step 2d
//Step 2e: Create a viewable MeshView to be added to the scene
//To add a TriangleMesh to a 3D scene you need a MeshView container object
MeshView meshView = new MeshView(mesh);
//The MeshView allows you to control how the TriangleMesh is rendered
meshView.setDrawMode(DrawMode.LINE); //show lines only by default
meshView.setCullFace(CullFace.BACK); //Removing culling to show back lines
//End Step 2e
//Step 2f: Add it to a group, this will be useful later
Group pyramidGroup = new Group();
pyramidGroup.getChildren().add(meshView);
//End Step 2f
//Step 2g: Customizing your Pyramid
if(null != color) {
    PhongMaterial material = new PhongMaterial(color);
    meshView.setMaterial(material);
}
if(ambient) {
    AmbientLight light = new AmbientLight(Color.WHITE);
    light.getScope().add(meshView);
    pyramidGroup.getChildren().add(light);
}
if(fill) {
    meshView.setDrawMode(DrawMode.FILL);
}
//End Step 2g
return pyramidGroup;
}

```

The new method `buildPyramid()` converts a height, representing the distance from the base to the pyramid apex, and a hypotenuse, the distance of the diagonal of the base, into vertices—that is, wound triangle faces—and builds a `TriangleMesh` object. The remaining parameters allow for customizing how the pyramid is rendered and will be explained later in the chapter. The first step is to declare a new `TriangleMesh` object. If you already had your points, faces, and texture coordinates defined from another process (or model file), you could pass this into an alternative `TriangleMesh` constructor. For our example we define our points using this statement:

```
mesh.getPoints().addAll(float x, float y, float z)
```

The `addAll()` method adds float triplets, which represent a point in the scene coordinate system. The order by which these points are added matters because their indices are used to “wind” the triangle faces later. Before winding the triangles, you must first define the 2D texture coordinates that will be associated with each triangle face. This can get complicated to explain. Fortunately the texture coordinates are only needed if you intend to wrap your 3D `TriangleMesh` with an image texture file. For this example we will create a single placeholder coordinate which will be used to fill out the faces using:

```
mesh.getTexCoords().addAll(float x, float y);
```

Finally we create our triangles by winding the points defined earlier. Here's the code from Listing 9-10:

```
mesh.getFaces().addAll( //use dummy texCoords
    0,0,2,0,1,0,  // Vertical Faces "wind" counter clockwise
    0,0,1,0,3,0,  // Vertical Faces "wind" counter clockwise
    0,0,3,0,4,0,  // Vertical Faces "wind" counter clockwise
    0,0,4,0,2,0,  // Vertical Faces "wind" counter clockwise
    4,0,1,0,2,0,  // Base Triangle 1 "wind" clockwise because camera has rotated
    4,0,3,0,1,0   // Base Triangle 2 "wind" clockwise because camera has rotated
); //End Step 2d
```

The `getFaces().addAll()` method accepts integers, which represent a composite of predefined points and texture coordinates. Figure 9-6 breaks down the first face from our example, demonstrating the winding of the points in a counterclockwise direction.

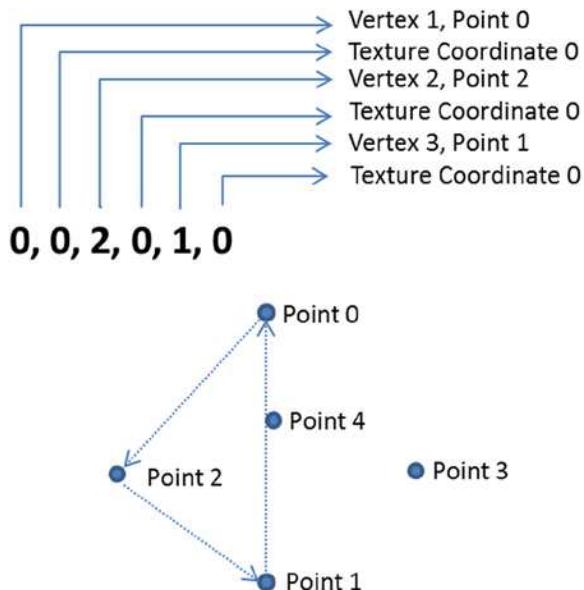


Figure 9-6. Triangle winding using predefined pyramid points

Note We use two triangle sections to make up the square base (down) face of the pyramid.

The counterclockwise winding order tells the camera system that the “front” of the face is facing up, or in this case out from our pyramid object. To build the square base we again use triangles, using two, but the winding is done clockwise. This is because the upward faces of these triangles are actually facing inward toward our pyramid. By winding clockwise, we effectively tell the camera to render the faces flipped. This will give the illusion of a solid geometric object.

A MeshView to a Fill

TriangleMesh objects cannot be added and rendered by themselves, because their geometry must be added to a viewable Scenegraph Node object. JavaFX 8 provides this capability via the MeshView class. In Listing 9-10 we use the following statement:

```
MeshView meshView = new MeshView(mesh);
```

The MeshView class not only displays the TriangleMesh added to it but allows you to customize lights, fill, materials, and other features, as demonstrated in Listing 9-10. Let’s use our new buildPyramid() method to exercise the flexibility of the MeshView class. Listing 9-11 uses buildPyramid() to create a pyramid with a GoldenRod colored PhongMaterial and a DrawMode set to LINE. These additional visual options were built into our buildPyramid() method in steps 2e and 2f from Listing 9-10.

Listing 9-11. Creating a Single Pyramid

```
/* From TriangleMeshes.java */

//Step 3a: Create a Pyramid with a color and add to the Scene
//Lets assume our height is 100 and our width (hypotenuse) at the base is 50.
Group pyramid1 = buildPyramid(100,200,Color.GOLDENROD,false,false);
Group pyramidGroup = new Group(pyramid1);
sceneRoot.getChildren().addAll(pyramidGroup);
```

This will render a pyramid similar to Figure 9-7. Most of the pyramid1 object is not viewable, because the default scene lighting is positioned in a way that doesn’t reflect much of the geometry. To see the entire geometry regardless of light system, turn on ambient lighting:

```
Group pyramid1 = buildPyramid(100,200,Color.GOLDENROD,true,false);
```



Figure 9-7. *TriangleMesh* with default light

Let's also translate the entire geometry closer to the camera:

```
pyramid1.setTranslateX(-100);
```

You will get a view similar to Figure 9-8. The geometry you defined in your *TriangleMesh* is now clearly visible.

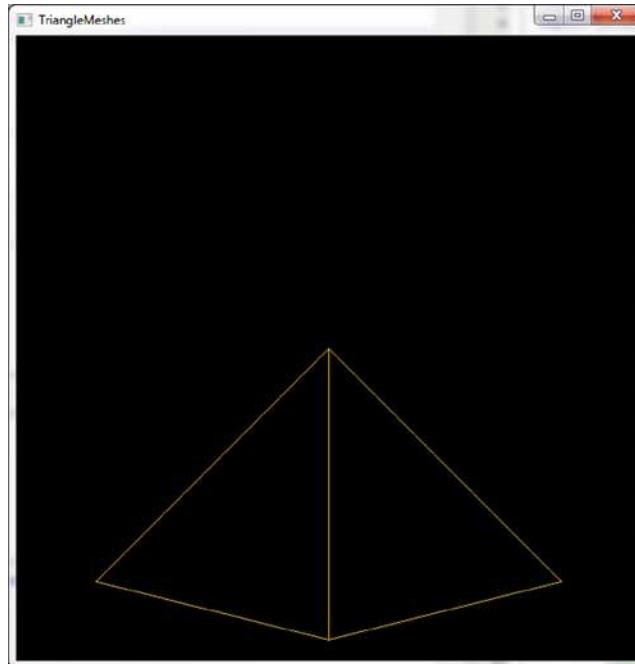


Figure 9-8. TriangleMesh translated with ambient light

Placing TriangleMesh objects within MeshView objects and Group containers makes basic transforms simple. A rotation used for a primitive, like the following:

```
pyramid1.setRotationAxis(Rotate.Y_AXIS);
pyramid1.setRotate(45);
```

provides a view similar to Figure 9-9. To fill in the pyramid with colored sides, use the *fill* parameter in the buildPyramid() method:

```
Group pyramid1 = buildPyramid(100,200,Color.GOLDENROD,true,true);
```

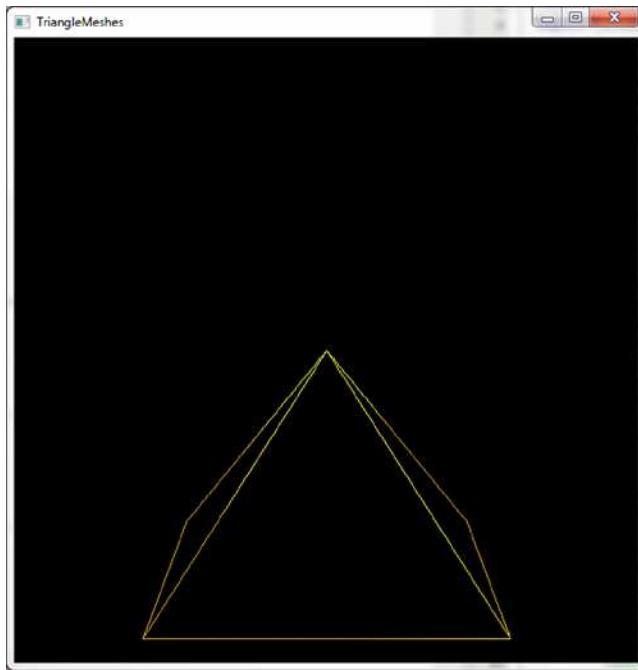


Figure 9-9. *TriangleMesh* rotated

This will provide a view similar to Figure 9-10.

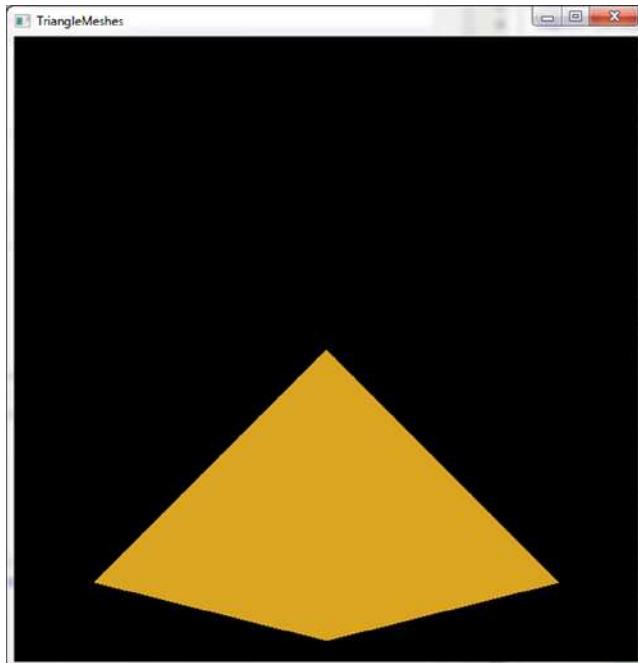


Figure 9-10. *TriangleMesh* with *DrawMode.FILL*

Listing 9-12 exercises different combinations of transformations to show the utility of the new `buildPyramid()` method.

Listing 9-12. Creating and Transforming a Pyramid using DrawMode.FILL

```
/* From TriangleMeshes.java */

//Step 3b: Create and transform a Pyramid using DrawMode FILL
Group pyramid2 = buildPyramid(100,200,Color.GOLDENROD,true,true);
//Since the pyramid is a group it can be translated and rotated like a primitive
pyramid2.setTranslateX(-100);
pyramid2.setTranslateY(-100);
pyramid2.setRotationAxis(Rotate.Z_AXIS);
pyramid2.setRotate(180);
Group pyramidGroup = new Group(pyramid1,pyramid2);
sceneRoot.getChildren().addAll(pyramidGroup);
```

Listing 9-13 exercises the ability of the `buildPyramid()` method to alter the appearance of each pyramid by adding pyramids of different material colors.

Listing 9-13. Creating and Transforming Pyramids of Different Colors

```
/* From TriangleMeshes.java */

//Step 3c: Add some more pyramids of a different color
Group pyramid3 = buildPyramid(100,200,Color.LAWNGREEN,true,true);
pyramid3.setTranslateX(100);
Group pyramid4 = buildPyramid(100,200,Color.LAWNGREEN,true,false);
pyramid4.setTranslateX(100);
pyramid4.setTranslateY(-100);
pyramid4.setRotationAxis(Rotate.Z_AXIS);
pyramid4.setRotate(180);
Group pyramidGroup = new Group(pyramid1,pyramid2,pyramid3,pyramid4);
sceneRoot.getChildren().addAll(pyramidGroup);
```

Listings 9-12 and 9-13 clearly demonstrate the flexibility of the `TriangleMesh` and `MeshView` classes within a 3D scene. Adding this code to our example will provide a view similar to Figure 9-11.

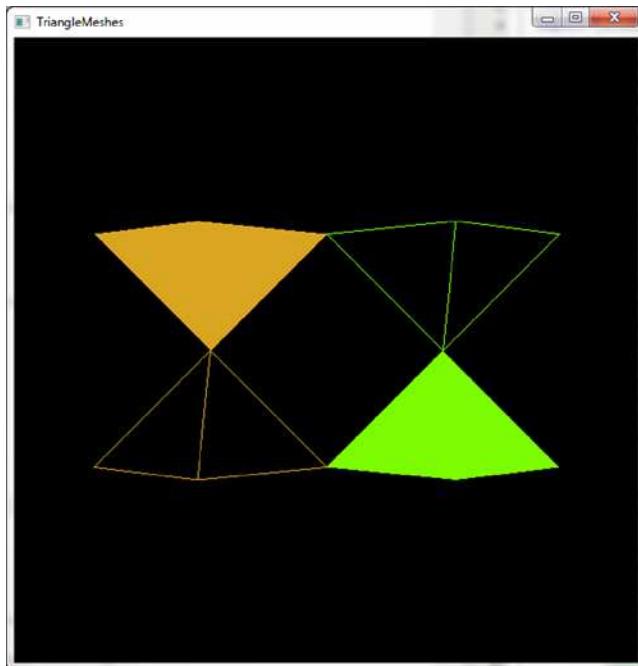


Figure 9-11. Multiple pyramids customized and transformed

Roll Camera!

Still, our new pyramids look a little flat. We need a way to roll the camera around the pyramids as if it was a satellite orbiting a point of origin. Listing 9-14 treats the `pyramidGroup` as an origin of sorts by adding a simple mouse handler and converting motion to a rotational transform of the `pyramidGroup` itself.

Listing 9-14. Origin Fixed Camera Rotation Mouse Handler

```
/* From TriangleMeshes.java */

//Step 4a: Add a Mouse Handler for Rotations
Rotate xRotate = new Rotate(0, Rotate.X_AXIS);
Rotate yRotate = new Rotate(0, Rotate.Y_AXIS);
pyramidGroup.getTransforms().addAll(xRotate,yRotate);
//Use Binding so your rotation doesn't have to be recreated
xRotate.angleProperty().bind(angleX);
yRotate.angleProperty().bind(angleY);
//Start Tracking mouse movements only when a button is pressed
scene.setOnMousePressed(event -> {
    scenex = event.getSceneX();
    sceney = event.getSceneY();
    anchorAngleX = angleX.get();
    anchorAngleY = angleY.get();
});
```

```
//Angle calculation will only change when the button has been pressed  
scene.setOnMouseDragged(event -> {  
    angleX.set(anchorAngleX - (scenex - event.getSceneY()));  
    angleY.set(anchorAngleY + sceney - event.getSceneX());  
});
```

This camera control is different from our earlier example in that it does not actually move or rotate the camera node but instead moves the objects in the scene. This gives the illusion that the viewer and camera are rotating around the scene and is an effective means for viewing scenes that are focused on a centralized object or rendering. The trick is to use JavaFX binding to update an existing set of Rotate objects. This saves the event handler from having to create and replace various transformation objects repeatedly. Adding the handler to our example and giving it a spin we can achieve viewing angles similar to Figure 9-12.

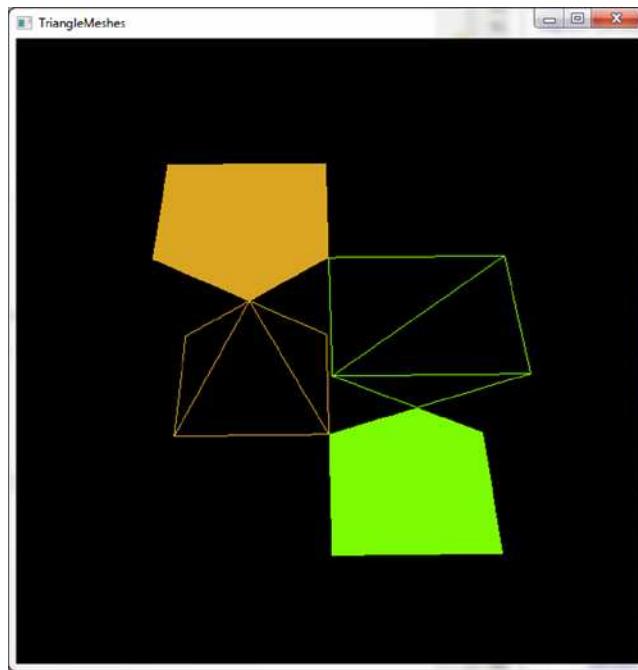


Figure 9-12. Rotating the Camera Around a Fixed Origin

The view captured in Figure 9-12 is particularly useful because it shows the various pyramids in their different forms. Of particular note is the bottom of the base of the pyramid, which is made up of two triangles using clockwise winding so that their “down” faces are facing outward toward the camera.

Hit the Lights

We have come so far with our custom Pyramid example, yet our 3D geometries still look a little cartoonish. This is because of the lighting, specifically the scene lighting. To truly show off the 3D surfaces, specular highlights, and the various shadings that come out of the box with the JavaFX 3D support, you must add a PointLight light to your scene. Listing 9-15 adds a simple PointLight to the scene in our example.

Listing 9-15. Adding a Point Light to a Scene

```
/* From TriangleMeshes.java */
```

```
//Step 4b: Add a Point light to show specular highlights
PointLight light = new PointLight(Color.WHITE);
sceneRoot.getChildren().add(light);
light.setTranslateZ(-sceneWidth/2);
light.setTranslateY(-sceneHeight/2);
```

Listing 9-15 includes some basic translations to move the light to a position other than our default camera position, thereby creating some nice specular highlights on the PhongMaterials of each filled pyramid. Adding Listing 9-15 to our example and rotating the camera provides nice views such as Figure 9-13.

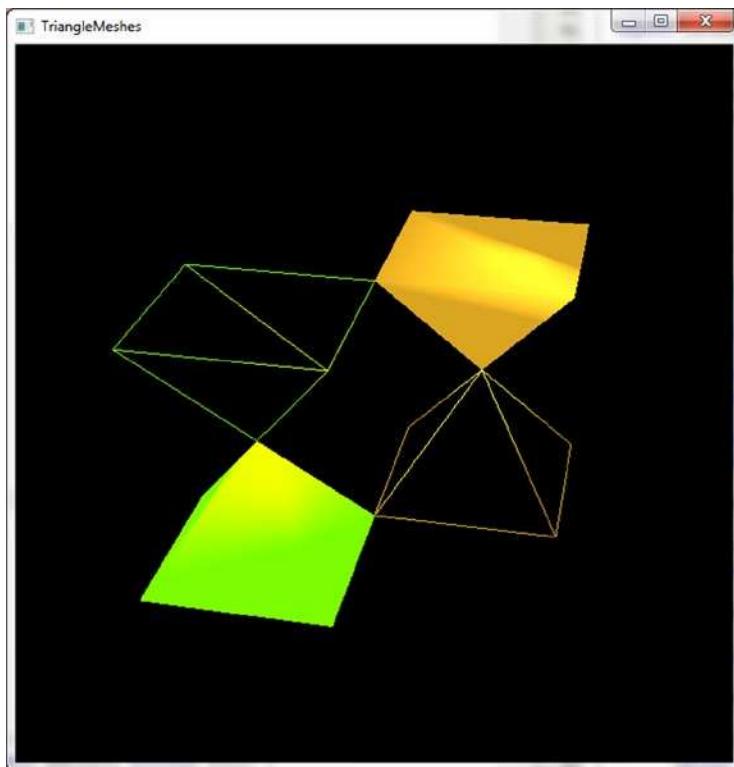


Figure 9-13. PointLight effects on various Pyramid TriangleMesh objects

Summary

As mentioned at the beginning of this chapter, a whole book could be dedicated to discussing the new JavaFX 8 3D support. There are many features that were alluded to but not discussed, as they would warrant whole chapters to explain. Whole texts are dedicated to the math and abstract concepts of 3D software development, and so those topics were not discussed in detail.

Hopefully, this chapter has scratched the surface of what is possible and also scratched your itch to learn about the new 3D support introduced in JavaFX 8. We used two separate running examples to discuss the usage of primitives, camera nodes, lights, and custom geometries and triangle meshes. Both examples featured event-driven interaction with different approaches to manipulating either the camera or the scene objects with the keyboard and mouse. You were introduced to a list of classes, some dedicated to 3D support, some versatile enough to be applied to either 3D or 2D scenes. Very useful and practical 3D tools can be developed from the building blocks provided in this chapter.

Chapter 10

JavaFX and Arduino

Arduino, an open-source electronics prototyping platform, is celebrating its tenth anniversary of availability in 2014—its first ten years of life. That means it's been here since the very beginning of the *Internet of things* concept, as we understand it now. In fact, the advent of low-cost prototyping platforms like Arduino is one of the cornerstones that have allowed this emerging revolution to happen, supporting both the do-it-yourself concept and the maker movement.

This chapter will give you a quick glance at how you can use JavaFX along with an Arduino board to develop desktop applications for monitoring data coming from the real world or controlling real devices.

We'll start by presenting briefly the available boards, followed by the software required to program the microcontroller and the procedure to connect it to a computer (without any WiFi or Ethernet shield solution); finally, we'll examine the code to acquire the data stream and represent the data and control the devices on the JavaFX thread.

The Arduino Board

Arduino is an open-source board with one microcontroller. The best-known version, the Arduino Uno rev 3, is based on the ATmega328 8-bit Atmel AVR microcontroller, while the new Arduino Due, based on the Atmel SAM3X8E ARM Cortex-M3 CPU, is the first Arduino board based on a 32-bit ARM core.

Thanks to a preprogrammed boot loader, programs can be easily uploaded into the microcontroller memory, directly from a connected computer.

The *Uno* provides 14 digital I/O pins and 6 analog inputs, as female headers, so plug-in shields can be mounted on top of them, or simple jumpers can be plugged to required pins. 32 KB are available as flash memory, and the clock operates at 16 MHz. Further details can be checked here: <http://arduino.cc/en/Main/ArduinoBoardUno>. Figure 10-1 shows front and back views of the Uno board.



Figure 10-1. Arduino Uno (front and back)

The Due, shown in Figure 10-2, increases the capacity to 54 digital I/O pins and 14 analog pins (12 inputs, 2 outputs). Flash memory is now 512 KB, while clock speed is 84 MHz. Further details can be checked at <http://arduino.cc/en/Main/ArduinoBoardDue>.

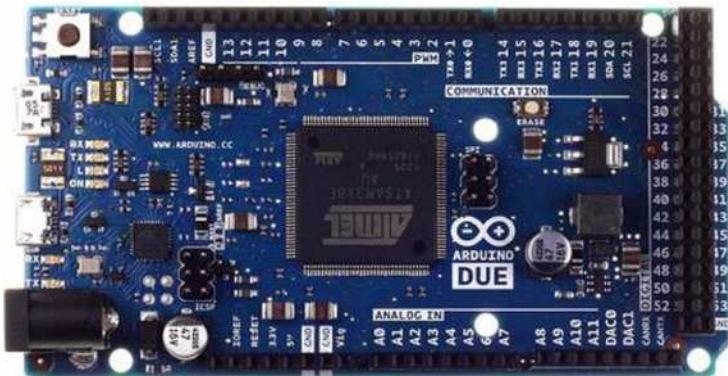


Figure 10-2. Arduino Due (front)

There are other boards with different specifications or components. For any concrete application, you should acquire the one that suits your needs, carefully reading the specifications at <http://arduino.cc/en/Main/Products>.

For the examples in this chapter you can use any Arduino board. If you don't have one, an Arduino Uno is recommended. There are plenty of sources where you may buy one online, as you can see at this link: <http://arduino.cc/en/Main/Buy>.

Once you've got the Arduino board, using a standard USB cable (type A plug to type B plug) you can attach it to your computer, as shown in Figure 10-3. This connection will provide 5V to power the board (check that the green power LED is illuminated) and a way to program the microcontroller from your computer.

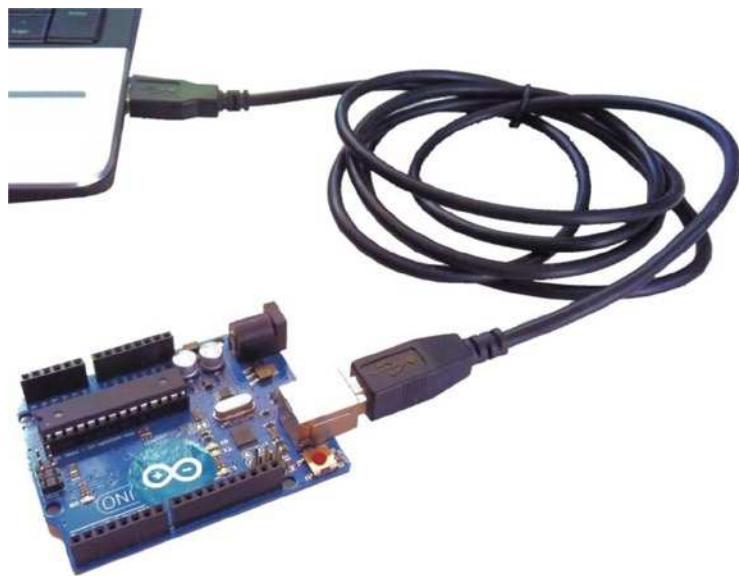


Figure 10-3. Arduino connected to the computer

Later, after you have programmed the microcontroller, you can use an external power supply, like a 9 V battery or a 9-to-12 V DC adapter, providing 250 mA up to 1000 mA depending on the devices connected, by plugging a 2.1 mm plug center pin positive into the board's power jack.

The Arduino IDE

To program the micro, first we need to install the drivers and the proper software. Go to the Downloads section at <http://arduino.cc/en/Main/Software>, select your OS, and download the latest release, 1.0.5 at the time of this writing; although, for Arduino Due or Yún you should use 1.5.5 beta.

The Arduino Software is distributed under the GNU Lesser General Public license. The open-source environment makes it easy to write code and upload it to the board. The Arduino Integrated Development Environment (IDE) runs on Windows, Mac OS X, or Linux. The environment is written in Java and is based on Processing, avr-gcc, and other open source software.

Windows Setup

Download the Windows Installer <http://arduino.googlecode.com/files/arduino-1.0.5-r2-windows.exe>, double click it, and proceed with the two steps shown in Figure 10-4.

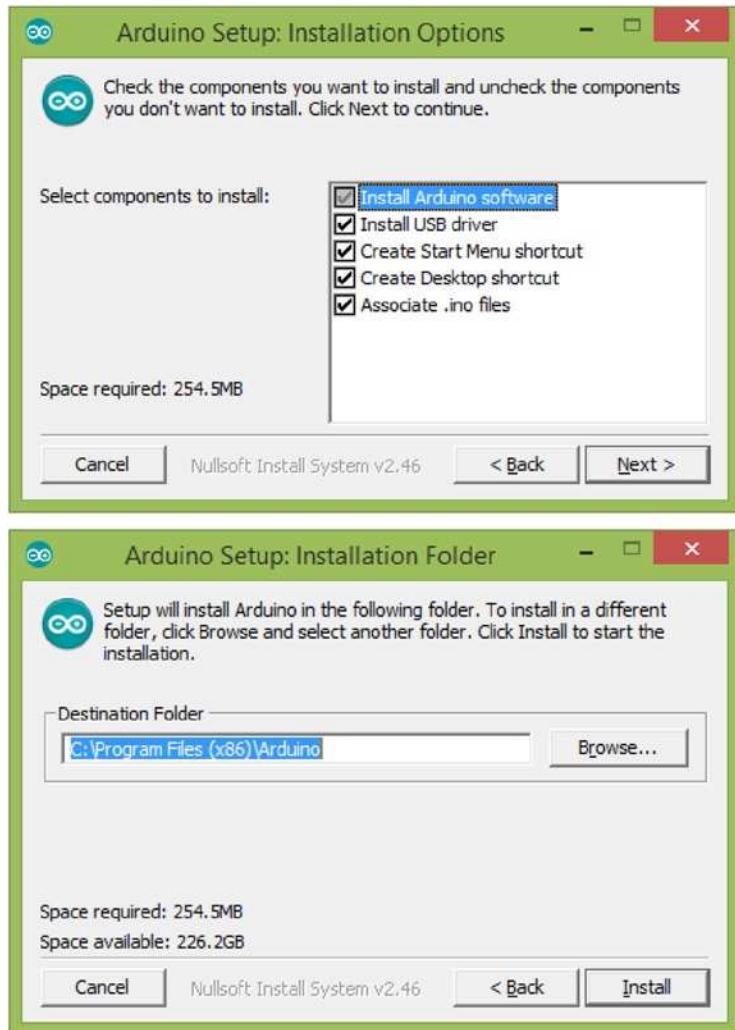


Figure 10-4. Steps during Arduino IDE installation

At the end of the process you'll be asked to install the driver. At this point connect the Arduino to your computer USB port, and a COM connection will be established. Check the Windows Device Manager to confirm that under Ports you see your Arduino board connected, as in Figure 10-5.

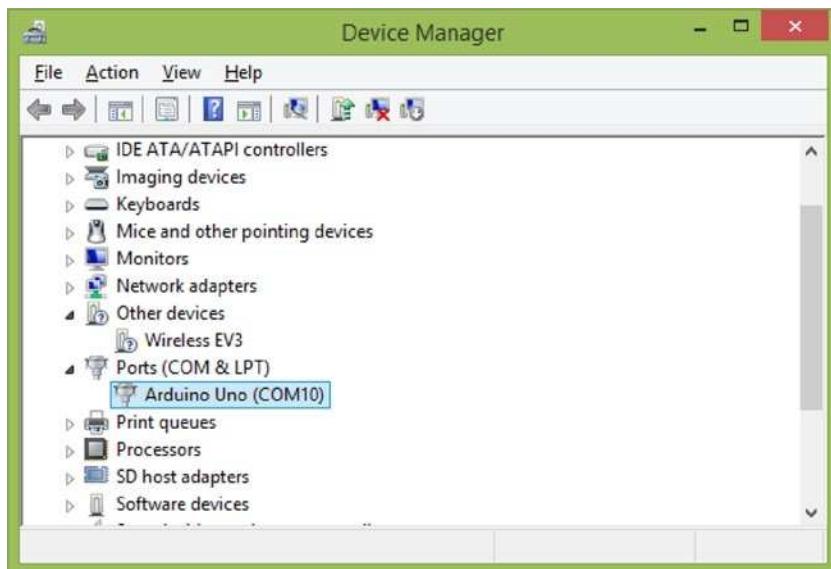


Figure 10-5. Arduino serial port in Windows

If you find a yellow triangle over Arduino under Other Devices, you will need to install the driver manually, following these steps (on a Windows 8 computer):

1. Disable driver signature enforcement:
 - Save and close all your work. On Settings (Windows key+I), click Power and then choose Shift+Restart. The computer will reboot immediately.
 - Select the following options: Troubleshoot > Advanced Options > Startup Settings and finally click Restart.
 - Once the Startup Settings shows up, select the 7 key, Disable Driver Signature Enforcement. Windows will reboot. Sign in normally.
2. Install the Arduino driver.
 - Open Device Manager, right-click under Arduino, and select Update Driver Software. Select Browse My Computer For Driver Software, and find the folder Drivers under the Arduino installation folder and click Next.
 - You will be prompted with the security window shown in Figure 10-6; select Install This Driver Software Anyway. After a few seconds, a success window will inform you the driver has been installed. Confirm that you can see it in the Device Manager, as in Figure 10-5.



Figure 10-6. Windows security warning

Mac OS X Setup

Download the file <http://arduino.googlecode.com/files/arduino-1.0.5-macosx.zip>, unzip it, and copy the folder to your Applications folder. No drivers are required.

When you connect the Arduino board, you should see it listed under /dev/tty.usbmodemXXXX or /dev/tty.usbserialXXXX.

Linux Setup

On Debian distributions (for example, Raspberry Pi) you can simply install it on the terminal:

```
$ sudo apt-get install arduino
```

For others distributions, please check here: <http://playground.arduino.cc/Learning/Linux>

You can also download the TAR file, unpack it, and install the required dependencies. Choose between 32 bit: <http://arduino.googlecode.com/files/arduino-1.0.5-linux32.tgz>, or 64-bit: <http://arduino.googlecode.com/files/arduino-1.0.5-linux64.tgz>.

When you connect the Arduino board, you should see it listed under /dev/ttyACMX or /dev/ttyUSBX.

Running the IDE

Assuming your installation ends successfully, launch the application; you'll see a screen like Figure 10-7.

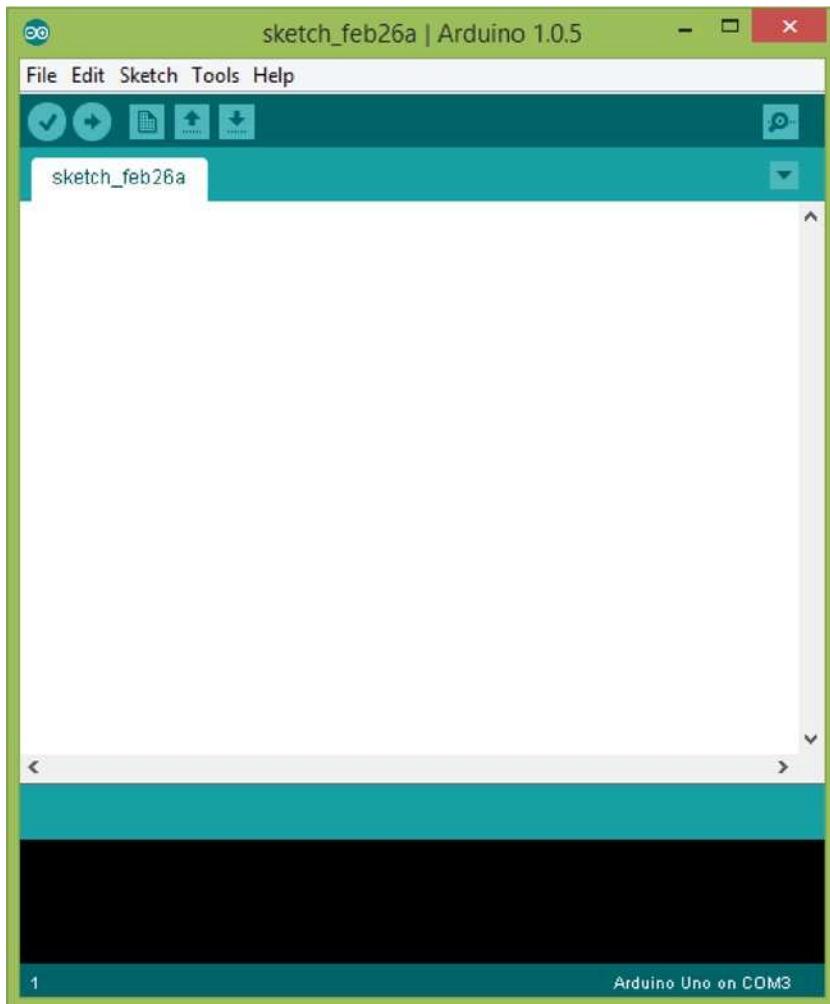


Figure 10-7. The Arduino IDE

The first thing to do is choose Tools > Board and select your own board from the list (Arduino Uno, for instance). Under Serial Port, select the port where the board is connected. In case of several options, an easy way to find out which port is the right one is to unplug the board, look for the missing port, plug it in again, and select that serial port.

The quickest test we can do, even without sensors (instrumentation or analog / digital signaling devices) is to open the Blink sample (Files > Examples > 01.Basics > Blink), as shown in Figure 10-8. The Arduino code files, called sketches, are saved with the extension .ino, under the Arduino/examples folders, in several categories.

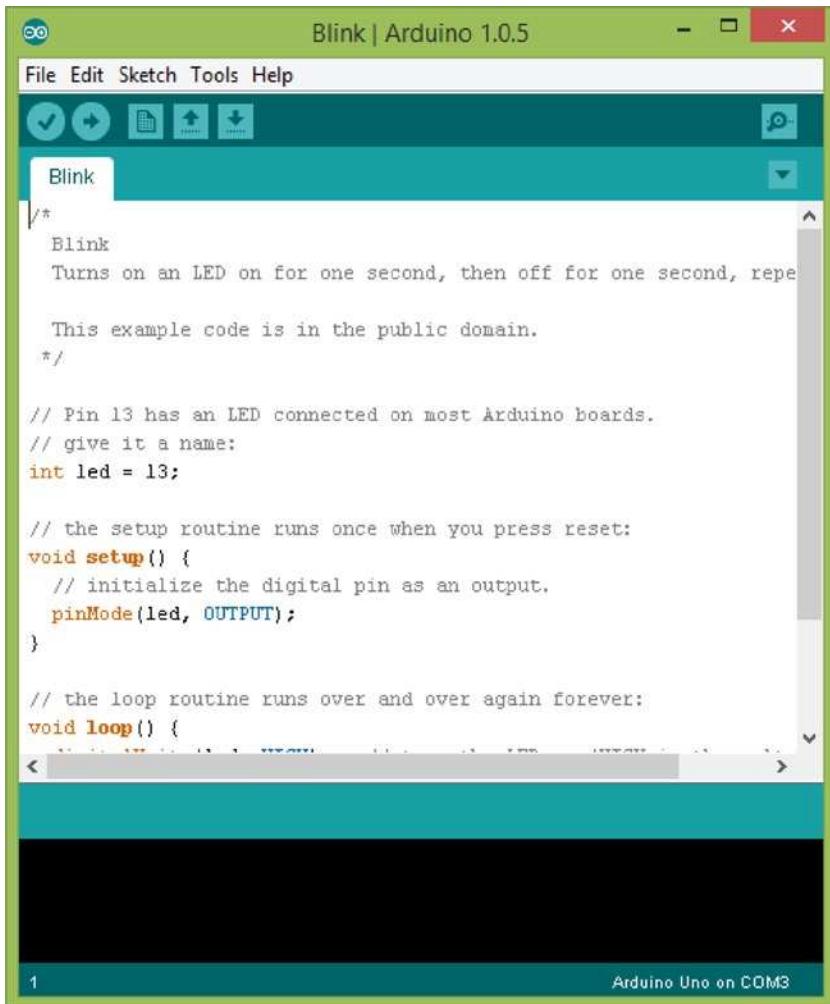


Figure 10-8. The Blink sketch

Click the Upload button (second from the left on the toolbar) and wait for the “Done uploading” message on the status bar. Then you should see the LED labeled with an L blinking on the board.

A really good source for example code to start playing with your Arduino board is at the Arduino Playground <http://playground.arduino.cc/>.

To try most of the preliminary tutorials or your own experiments, you’ll need a basic set of sensors and a small set of electronic components. It is a good idea to acquire your Arduino board bundled with electronic components, as a Starter Kit. In addition to the board, you will want a USB cable, a breadboard, jumper wires, LEDs of several colors, resistors, a temperature sensor, a potentiometer, push buttons, and the like.

Light Tracking

For our first example we'll use the components shown in Figure 10-9: an Arduino Uno, a breadboard, a photoresistor (like this one: <http://www.adafruit.com/products/161> or this one: <https://www.sparkfun.com/products/9088>), a 10 kohm resistor, plus several jumpers (wires).

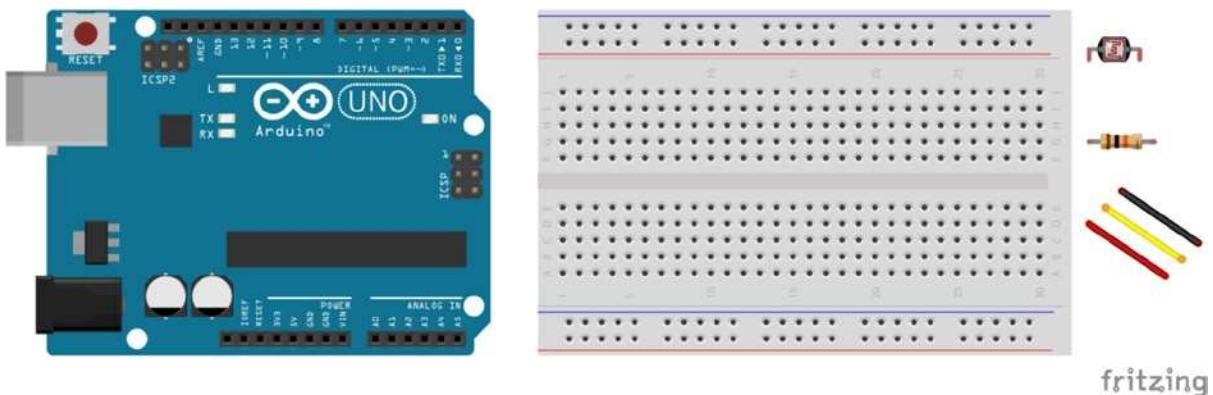


Figure 10-9. Bill of materials

Fritzing, an open source tool to document the Arduino-based prototype, can be very helpful in drawing the layout of our connections before we build them. It can be downloaded from here: <http://www.fritzing.org/>. Figure 10-10 shows our planned layout.

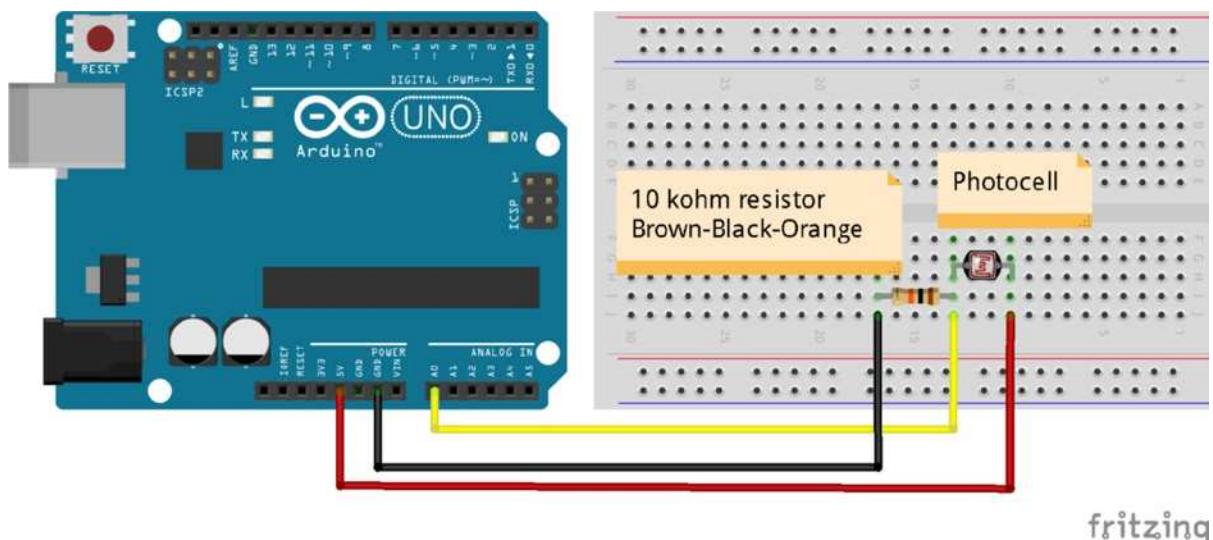


Figure 10-10. Circuit layout for the photocell example

Tinkering

The first thing we should do is convert our conceptual layout to a real one; so please grab your Arduino, the breadboard, the sensor, the resistor and the jumpers and build it! Don't be afraid, there's no risk at all!

The steps are really simple, assuming you use the same colors as shown in Figure 10-10:

1. Insert the wires from the ends of the components into the breadboard, as shown, aligning one end of the resistor and one end of the photocell in the same bank with the yellow jumper going to the Analog A0 pin of the Arduino board.
2. Run a black jumper from the remaining end of the resistor to Power ground (GND) on the Arduino.
3. Finally, run a red jumper from the remaining end of the photocell to the Power 5V pin on the Arduino.

Figure 10-11 shows the result.

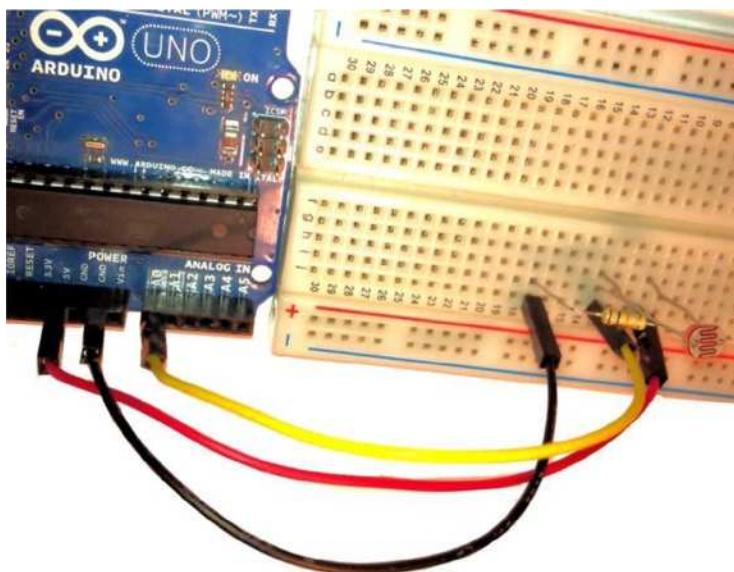


Figure 10-11. Arduino Uno with photocell

The Arduino Photocell Sketch

Now that we've got everything configured, we need to program the microcontroller. For that we'll use the code showed in Listing 10-1.

Listing 10-1. Arduino Sketch Photocell.ino

```
/*
Sketch for the analog read of a photocell, Light Dependent Resistor (LDR) or Photoresistor
 *One end of the photocell is connected to 5V
 *The other end to Analog 0 and to one end of a 10K resistor.
 *The other end of the resistor connected to ground
Available at:
 * http://www.adafruit.com/products/161
 * https://www.sparkfun.com/products/9088

Note:
 * Vout=Vin*R/(R+Rldr), where Vin=5V, R=10 kohm
 * Lux= Ea*(Ra/Rldr)^(1/sens)
   for a typical LDR, Ea=10 lux, Ra=10 kOhm, sens=0.8
   with Rldr in kohm. Check the datasheet.
 */

int pinLDR = A0; //analog pin 0

void setup() {
    Serial.begin(9600);
}

void loop() {
    // 1. Get the analog reading. Arduino Uno, Mega, between 0 and 1023
    int analogLDR = analogRead(pinLDR);

    // 2. Get the output voltage, if Vdd=5V
    float VOut = 5.0*analogLDR/1023.0;

    // 3. Get the LDR resistance (kohm)
    float resisLDR=10.0*(5.0/VOut-1.0);

    // 4. Get the amount of light (lux) assuming a typical LDR
    float lux=16.0*pow(10.0/resisLDR,1.0/0.8);

    // 5. Print to serial results
    Serial.println(lux);

    // Adjust delay in ms
    delay(100);
}
```

How It Works

As you can see in Listing 10-1, the sketch has two primary methods: `setup()` and `loop()`. In the former you initialize the required pins as input or output, or the serial port speed; in the latter you encode all the tasks that will be repeatedly executed. The analog readings from the sensors are just sent to the serial port, after they are converted to level of illumination, measured in lux.

Note Don't be confused with the calculations involved. They are specific to this sensor. For other sensors you'll need to perform similar transformations to convert the analog readings to the proper magnitude. Check the sensor datasheet for more details in each case.

The LDR is a light detector resistor, which means it changes its resistance with light: more light implies lower resistance. We use a 10 kohm fixed resistor to compare values. So we're splitting 5 V between the LDR and the resistor.

With the Arduino Uno the resolution of the analog readings is 10 bits, so there are 1024 possible values, between 0 and 5 V. An analog reading of 512 will indicate that the LDR has the same resistance as the resistor, and the output voltage will be 2.5 V.

Note With an Arduino Due, the resolution increases to 12 bits, giving 4096 values. If you use this board, be sure update the code accordingly.

Then, as indicated in the `loop()` method, the analog reading is first converted to volts:

$$V_{out} = V_{in} * \text{analogLDR}/1023$$

and then the resistance of the LDR is determined:

$$R_{LDR} = R * (V_{in} / V_{out} - 1)$$

Finally, taking into account the specific values of the photocell (sensitivity in ohm/lux and its value at 10 kohms), the level of illumination is determined. For 2.5 V, this level will be around 10 lux, corresponding to a dimly lit room. Readings close to 1023 will mean nearly 5 V and 10000-25000 lux, full daylight. By contrast, readings toward 0.1 V will equal 0.1 lux, like a moonless night. For further explanation, please check this link: <http://learn.adafruit.com/downloads/pdf/photocells.pdf>.

Testing the Sketch

Before going any further, please copy the listing to your Arduino IDE, and connect the Arduino (with the photocell connected as described) to the USB port of your computer. Check the name of the board and the port, and download the program. Figure 10-12 shows the IDE with the sketch, and on the bottom the name of the board and the port used. If you click Verify, the sketch will be compiled. Then click Upload, and the binary sketch will be compiled again and uploaded to the microcontroller. Click Save to store the sketch on your system, giving it a proper name, like `Photocell.ino`.

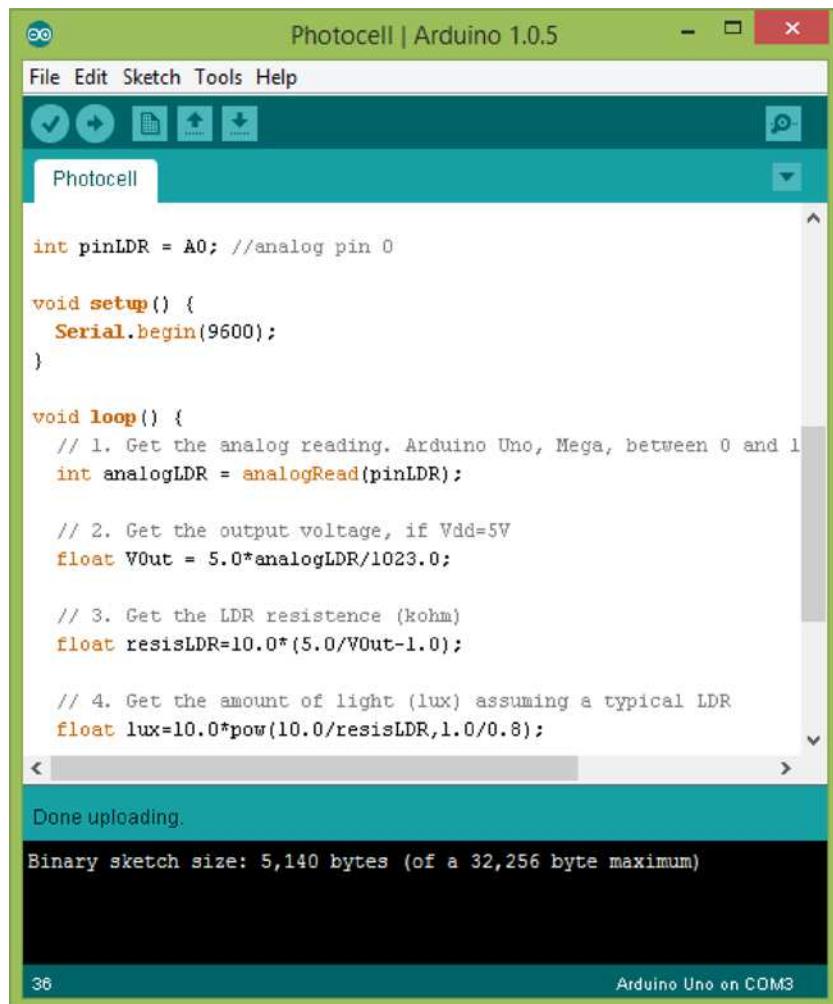


Figure 10-12. Arduino IDE with the photocell sketch

Without unplugging the USB cable, now open the Serial Monitor to check if the measures are showing up on your screen, as in Figure 10-13. If not, go back again and repeat the previous steps.

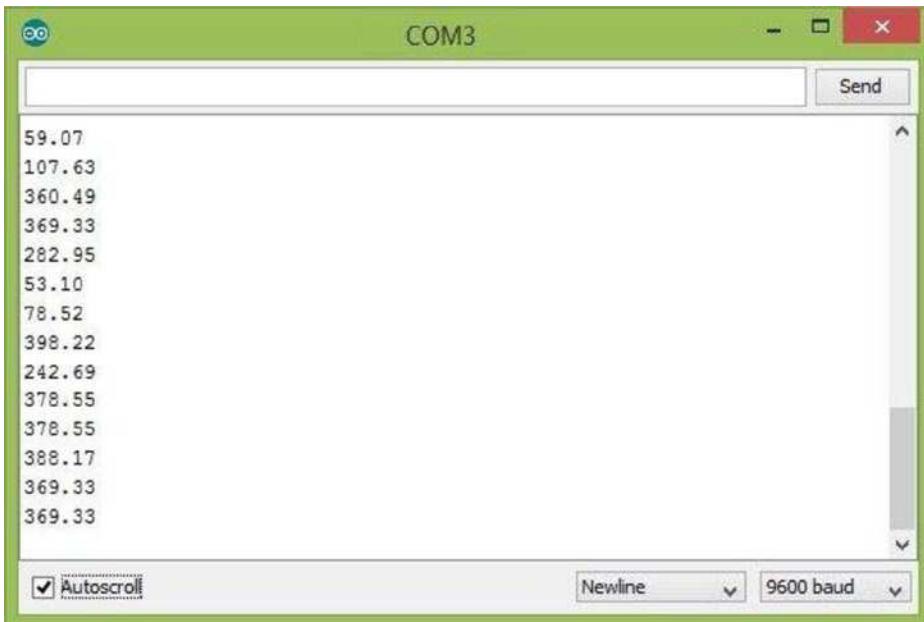


Figure 10-13. Serial Monitor

Note Keep in mind that with an inexpensive LDR, and without proper calibration or real values for its sensitivity, the light levels we'll be reading are an approximation.

If everything is working, it's time to go back to your computer, leave the Arduino working (or not; you can plug it in or unplug it whenever you want), and start thinking about how we are going to read the serial port to get the measures from the sensors and how we are going to display those readings in our JavaFX application.

Serial Reading

Although you can find third-party applications for serial reading, like the old HyperTerminal for Windows, Minicom for Linux, and Terminal for Mac, we need to have these readings in our application, and for that we need a Java API that performs that task.

Note here that serial reading is a highly hardware-specific task, and it breaks the Java multiplatform approach as we need native support for accessing the ports. Besides the old JavaComm API from Sun, there's no official API for communications bundled with the Java SE distribution, so we'll need a third-party library.

For several years, the preferred Java library for serial communication was RXTX. Originally from Trent Jarvi, and distributed under LGPL v 2.1+Linking Over Controlled Interface license, his last version (2.2pre2 from 2010) was accessible at <http://rxtx.qbang.org/>. In fact, you will see this library still distributed with the Arduino IDE 1.0.5, since it's used for communications between the IDE and the board.

But for a few months now, the site has been offline, and it seems there's no further support for this library.

So we need another alternative. So did the Arduino guys, who already are planning to migrate from RXTX to a new one.

Java Simple Serial Connector

Java Simple Serial Connector (jSSC), from Alexey Sokolov, is licensed under GNU Lesser GPL. The most recent version as of this writing (jSSC-2.8.0), released January 2014, is the library that was chosen to replace RXTX in the next Arduino IDE versions. You can try the current Arduino IDE nightly builds, beta versions (from 1.5.6), which have included this library.

jSSC is available for all platforms and operating systems. You'll notice it is faster than RXTX, and several bugs have been fixed. jSSC comes in a simple JAR file, with all the native required files included, so we don't need to deal with local installations, as it takes care of adding them to the classpath at runtime.

Just grab the latest available version. Go to <https://github.com/scream3r/java-simple-serial-connector/releases>, and download the JAR. If you add it as a library in NetBeans or open it, you will see its classes (the jssc package) and the native content (the libs package), as shown in Figure 10-14.

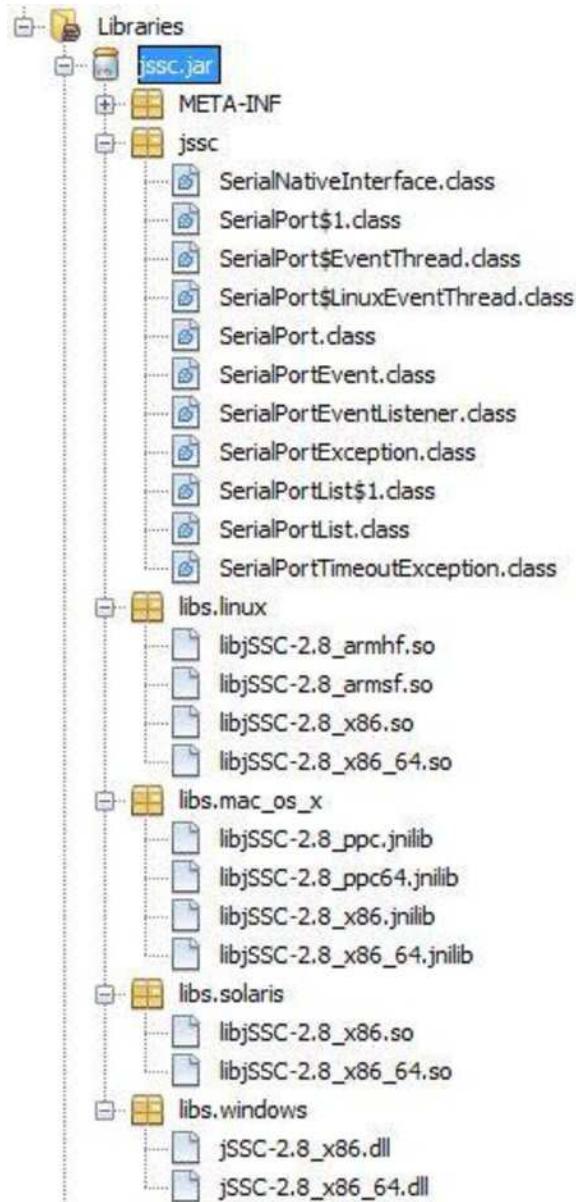


Figure 10-14. Contents of the jSSC library

JavaFX, the Charting API, and Sensor Readings

In the examples that follow, we will design a simple JavaFX application that takes the readings from the sensor and displays them in a chart. For the sake of clarity, we'll use two classes, one for the serial readings in Listing 10-2, and one for the chart and the JavaFX stage in Listing 10-3 later.

We'll bind these classes (`Serial` and `PhotocellFX`) by using a `StringProperty`, containing the last line read from the serial port. By listening to changes in this property in the JavaFX thread, we'll know when there's a new reading to add to the chart.

Serial Communications

Let's start with the `Serial` class, shown in Listing 10-2.

Listing 10-2. Source Code of the Serial Class

```
package jfx8ibe;

import java.util.Arrays;
import java.util.List;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import jssc.SerialPort;
import jssc.SerialPortException;
import jssc.SerialPortList;

/**
 * Chapter 10. JavaFX and Arduino
 * @author pereda
 */
public class Serial {
    /* List of usual serial ports. Add more or remove those you don't need */
    private static final List<String> USUAL_PORTS = Arrays.asList(
        "/dev/tty.usbmodem", "/dev/tty.usbserial", // Mac OS X
        "/dev/usbdev", "/dev/ttyUSB", "/dev/ttyACM", "/dev/serial", // Linux
        "COM3", "COM4", "COM5", "COM6" // Windows
    );
    private final String ardPort;
    private SerialPort serPort;

    public static final String SEPARATOR= ";";
    private StringBuilder sb=new StringBuilder();
    private final StringProperty line=new SimpleStringProperty("");

    public Serial(){
        ardPort="";
    }

    public Serial(String port){
        ardPort=port;
    }

    /*
     * connect() looks for a valid serial port with an Arduino board connected.
     * If it is found, it's opened and a listener is added, so every time
     * a line is returned, the stringProperty is set with that line.
    
```

```

* For that, a StringBuilder is used to store the chars and extract the line
* content whenever '\r\n' is found.
*/
public boolean connect(){
    Arrays.asList(SerialPortList.getPortNames())
        .stream()
        .filter(name->
            ((!ardPort.isEmpty() && name.equals(ardPort)) ||
            (ardPort.isEmpty() &&
            USUAL_PORTS.stream()
                .anyMatch(p -> name.startsWith(p)))))
        .findFirst()
        .ifPresent(name -> {
            try {
                serPort = new SerialPort(name);
                System.out.println("Connecting to "+serPort.getPortName());
                if(serPort.openPort()){
                    serPort.setParams(SerialPort.BAUDRATE_9600,
                        SerialPort.DATABITS_8,
                        SerialPort.STOPBITS_1,
                        SerialPort.PARITY_NONE);
                    serPort.setEventsMask(SerialPort.MASK_RXCHAR);
                    serPort.addEventListener(event -> {
                        if(event.isRXCHAR()){
                            try {
                                sb.append(serPort.readString(event.getEventValue()));
                                String ch=sb.toString();
                                if(ch.endsWith("\r\n")){
                                    // add timestamp
                                    line.set(Long.toString(System.currentTimeMillis()))
                                        .concat(SEPARATOR)
                                        .concat(ch.substring(0,
                                            ch.indexOf("\r\n"))));
                                    sb=new StringBuilder();
                                }
                            } catch (SerialPortException e) {
                                System.out.println("SerialEvent error:"+ e.toString());
                            }
                        });
                    });
                }
            } catch (SerialPortException ex) {
                System.out.println("ERROR: Port '" + name + "' : " +ex.toString());
            }
        });
    return serPort!=null;
}

```

```
public void disconnect(){
    if (serPort != null) {
        try {
            serPort.removeEventlistener();

            if(serPort.isOpened()){
                serPort.closePort();
            }
        } catch (SerialPortException ex) {
            System.out.println("ERROR closing port exception: "+ex.toString());
        }
        System.out.println("Disconnecting: comm port closed.");
    }
}

public StringProperty getLine(){
    return line;
}

public String getPortName(){
    return serPort!=null?serPort.getPortName():"";
}
}
```

How It Works

First of all, notice the jSSC imports:

```
import jssc.SerialPort;
import jssc.SerialPortException;
import jssc.SerialPortList;
```

You must include jSSC.jar in your project. This can be done by adding it to <JAVA_HOME>/jre/lib/ext on Linux or Windows (or /Library/Java/Extensions on Mac). Or you can just add it to the project as a dependency, as shown in Figure 10-14. The latter option is recommended if you intend to distribute your application.

Next, we set a list of suitable port names where the Arduino board may be connected. This list can be extended to other ports in your system or you can delete the ones from other platforms. The ardPort variable can be set with a port name through the constructor, and the list won't be used.

```
private static final List<String> USUAL_PORTS = Arrays.asList(
    "/dev/tty.usbmodem", "/dev/tty.usbserial", // Mac OS X
    "/dev/usbdev","/dev/ttyUSB","/dev/ttyACM", "/dev/serial", // Linux
    "COM3","COM4","COM5","COM6" // Windows
);

private final String ardPor;

public Serial(String port){
    ardPor=port;
}
```

In the connect() method we go through the list of valid serial ports found in your system:

```
public boolean connect(){
    Arrays.asList(SerialPortList.getPortNames())
        .stream()
```

We use lambdas to filter this collection so we get an item matching the ardPort value, if it is set, or one of the USUAL_PORTS list, otherwise:

```
.filter(name->
    (!ardPort.isEmpty() && name.equals(ardPort)) ||
    (ardPort.isEmpty() &&
    USUAL_PORTS.stream()
        .anyMatch(p -> name.startsWith(p))))
.findFirst()
```

If we find a valid serial port, we try to open it, and we set several common parameters for the serial communication.

```
.ifPresent(name -> {
    try {
        serPort = new SerialPort(name);
        System.out.println("Connecting to "+serPort.getPortName());
        if(serPort.openPort()){
            serPort.setParams(SerialPort.BAUDRATE_9600,
                SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE);
            serPort.setEventsMask(SerialPort.MASK_RXCHAR);
```

Because we'll be listening to the serial port, we add a listener that reacts to every `SerialPortEvent`, which it detects by the presence of bytes on the input buffer. The number of bytes is determined by `event.getEventValue()`. The `readString(int byteCount)` method returns whenever there are bytes on the buffer. We use a `StringBuilder` to store all the readings, and we evaluate the characters for a carriage return and line feed ("`\r\n`") sequence, and if they're found, to extract the line of text and reset the builder. A `StringProperty` is used to store the line, so the JavaFX thread can listen to changes in that observable object (exposed with the `getLine()` method; see Listing 10-3 later in the chapter). We add a timestamp to the string, to keep track of the instant it was read. Tracking over time also helps when the measures are the same, so they can be shown sequentially on the chart.

```
serPort.addEventListener(event -> {
    if(event.isRXCHAR()){
        try {
            sb.append(serPort.readString(event.getEventValue()));
            String ch=sb.toString();
            if(ch.endsWith("\r\n")){
                // add timestamp
                line.set(Long.toString(System.currentTimeMillis()))
```

```

        .concat(SEPARATOR)
        .concat(ch.substring(0,
                           ch.indexOf("\r\n"))));
    sb=new StringBuilder();
}
} catch (SerialPortException e) {
    System.out.println("SerialEvent error:"+ e.toString());
}
}
}

```

Note that in cases where you have to read several measures at a time, you need to use the same separator character in your Arduino sketch, to split them properly into individual readings.

Finally, the disconnect() method takes care of removing the listener and closing the port:

```

try {
    serPort.removeEventListerner();
    if(serPort.isOpened()){
        serPort.closePort();
    }
} catch (SerialPortException ex) {
    System.out.println("ERROR closing port exception: "+ex.toString());
}
}

```

The JavaFX Charts API

Charts with two axes (bars, areas, lines, bubble or scatter) or pie charts without an axis have been available in JavaFX since version 2.0. Each chart is a node, so it can be added to a scene like any other node.

In the JavaFX scene of our example, Listing 10-3, we'll add a LineChart to plot the light level from the photocell. This requires a Series to plot the readings, with XYChart.Data as pair of values for every point, one for each axis.

Two more Series will be added to our chart with the moving averages of every 50 values plotted (5 seconds) and 100 values (10 seconds). We want to show only the last 30 seconds on the chart, so taking into account the 100 ms delay from the board, whenever the size of the Series is greater than 300 points, the first values will be removed.

Listing 10-3. Source Code of the JavaFX Application Class

```

package jfx8ibe;

import java.text.SimpleDateFormat;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.beans.property.BooleanProperty;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.value.ChangeListener;
import javafx.geometry.Insets;

```

```

import javafx.scene.Scene;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart.Data;
import javafx.scene.chart.XYChart.Series;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.util.StringConverter;

/**
 * Chapter 10. JavaFX and Arduino
 * @author pereda
 */
public class PhotocellFX extends Application {

    private LineChart<Number,Number> chart;
    private Series<Number, Number> luxSeries;
    private Series<Number, Number> aveLuxSeries5;
    private Series<Number, Number> aveLuxSeries10;
    private NumberAxis xAxis;
    private NumberAxis yAxis;
    private final int tamMax=300;      // 30'' / 0.1''/reading = 300 readings
    private final int tamAve5=50;      // 5''

    private final int tamAve10=100;    // 10''

    /* Check in the Arduino IDE
     the port used by Arduino or leave it empty if you don't know it: */
    private final Serial serial=new Serial();
    private ChangeListener<String> listener;
    private final BooleanProperty connection=new SimpleBooleanProperty(false);

    @Override
    public void start(Stage primaryStage) {
        xAxis=new NumberAxis();
        xAxis.setLabel("Time");
        xAxis.setAutoRanging(true);
        xAxis.setForceZeroInRange(false);
        xAxis.setTickLabelFormatter(new StringConverter<Number>(){
            @Override
            public String toString(Number t) {
                return new SimpleDateFormat("HH:mm:ss").format(new Date(t.longValue()));
            }
            @Override
            public Number fromString(String string) {
                throw new UnsupportedOperationException("Not supported yet.");
            }
        });
        yAxis=new NumberAxis();
        yAxis.setLabel("Light (lux)");
    }
}

```

```

chart=new LineChart<>(xAxis,yAxis);
chart.setCreateSymbols(false);
chart.setAnimated(false);
chart.setLegendVisible(true);
chart.setTitle("Photocell Readings");
luxSeries = new Series<>();
luxSeries.setName("Light Level (lux)");
aveLuxSeries5 = new Series<>();
aveLuxSeries5.setName("Moving Average 5'' (lux)");
aveLuxSeries10 = new Series<>();
aveLuxSeries10.setName("Moving Average 10'' (lux)");
chart.getData().addAll(luxSeries,aveLuxSeries5,aveLuxSeries10);
listener=(ov, t, t1) -> {
    Platform.runLater(()->{
        try{
            String[] data=t1.split(Serial.SEPARATOR);
            if(data.length==2){
                long time=new Long(data[0]);
                luxSeries.getData().add(new Data<>(time,new Double(data[1])));
                if(luxSeries.getData().size()>tamMax){
                    luxSeries.getData().remove(0);
                }
                if(luxSeries.getData().size()>tamAve5){
                    aveLuxSeries5.getData().add(new Data<>(time, average(tamAve5)));
                    if(aveLuxSeries5.getData().size()>tamMax){
                        aveLuxSeries5.getData().remove(0);
                    }
                }
                if(luxSeries.getData().size()>tamAve10){
                    aveLuxSeries10.getData().add(new Data<>(time, average(tamAve10)));
                    if(aveLuxSeries10.getData().size()>tamMax){
                        aveLuxSeries10.getData().remove(0);
                    }
                }
            }
        } catch(NumberFormatException nfe){
            System.out.println("NFE: "+t1+ " "+nfe.toString());
        }
    });
    serial.getLine().addListener(listener);
}

BorderPane root = new BorderPane();
StackPane stack =new StackPane();
stack.setPadding(new Insets(10,0,10,0));
stack.getChildren().add(chart);
root.setCenter(stack);

```

```
Label lbl=new Label("Not connected");
connection.addListener((ov, b, b1)->lbl.setText(b1?
    "Connected to: "+serial.getPortName()."Not connected"));
root.setBottom(lbl);
root.setStyle("-fx-background-color: derive(goldenrod,60%); "
    + "-fx-font: 16 \'Courier New\';");

Scene scene = new Scene(root, 800, 600);
primaryStage.setTitle("Arduino Test. Photocell");
primaryStage.setScene(scene);
primaryStage.show();

startSerial();
}

@Override
public void stop(){
    System.out.println("Closing serial port");
    serial.getLine().removeListener(listener);
    stopSerial();
}

private void startSerial(){
    serial.connect();

    connection.set(!serial.getPortName().isEmpty());
}

private void stopSerial(){
    if(connection.get()){
        serial.disconnect();
        connection.set(false);
    }
}

private double average(int tam){
    return luxSeries.getData()
        .stream()
        .skip(luxSeries.getData().size()-tam)
        .mapToDouble(d->d.getYValue().doubleValue())
        .sum()/tam;
}

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

How It Works

First of all, note the fields declared. We have a `LineChart`, with three `Series` that take pairs of numbers to be plotted against each axis, which are `NumberAxis` instances. We also define several limits for these `Series`: the maximum number of points in each of them will be 300, relative to the last 30 seconds. For the moving averages, they will be calculated based on periods of 5 seconds and 10 seconds.

```
private LineChart<Number,Number> chart;
private XYChart.Series<Number, Number> luxSeries;
private XYChart.Series<Number, Number> aveLuxSeries5;
private XYChart.Series<Number, Number> aveLuxSeries10;
private NumberAxis xAxis;
private NumberAxis yAxis;
private final int tamMax =300; // 30'' / 0.1''/reading = 300 readings
private final int tamAve5 = 50; // 5'
private final int tamAve10=100; // 10''
```

Additionally, we have an instance of the `Serial` class from Listing 10-2, with the default constructor. If you know the serial port where the Arduino board is connected, you can use the new `Serial ("<port name>")` constructor instead. A `BooleanProperty` will be used to keep track of the connection status.

```
/* Check in the Arduino IDE the port used by Arduino or leave it empty
   if you don't know it: */
private final Serial serial=new Serial();
private ChangeListener<String> listener;
private final BooleanProperty connection=new SimpleBooleanProperty(false);
```

Next, we'll create the chart. First, we'll start by creating both axes. For the `xAxis` to show formatted data, we'll override the `toString()` method, so every long value in milliseconds will be written in `HH:mm:ss` format:

```
@Override
public void start(Stage primaryStage) {
    xAxis=new NumberAxis();
    xAxis.setLabel("Time");
    xAxis.setAutoRanging(true);
    xAxis.setForceZeroInRange(false);
    xAxis.setTickLabelFormatter(new StringConverter<Number>(){
        @Override
        public String toString(Number t) {
            return new SimpleDateFormat("HH:mm:ss").format(new Date(t.longValue()));
        }
        @Override
        public Number fromString(String string) {
            throw new UnsupportedOperationException("Not supported yet.");
        }
    });
    yAxis=new NumberAxis();
    yAxis.setLabel("Light (lux)");
```

Then we'll create the chart by specifying its axes. We set a few properties: `setCreateSymbols(true)` will plot a big white circle on every point of the series. For now we set it to false, so it won't create a node on the data. `setAnimated(false)` avoids the animation of the chart after adding any new point. Because we are plotting points at high frequency, it's better this way. For lower frequencies, on the other hand, we will set it to true to create a smooth transition effect. `setLegendVisible(true)` shows the legend with the series names.

```
chart=new LineChart<>(xAxis,yAxis);
chart.setCreateSymbols(false);
chart.setAnimated(false);
chart.setLegendVisible(true);
chart.setTitle("Photocell Readings");
```

Then, we'll create the three series, set names for them (which will be displayed in the legend box), and add them to the chart Data, which is an `ObservableList<Series>`:

```
luxSeries = new XYChart.Series<>();
luxSeries.setName("Light Level (lux)");
aveLuxSeries5 = new XYChart.Series<>();
aveLuxSeries5.setName("Moving Average 5'' (lux)");
aveLuxSeries10 = new XYChart.Series<>();
aveLuxSeries10.setName("Moving Average 10'' (lux)");

chart.getData().addAll(luxSeries,aveLuxSeries5,aveLuxSeries10);
```

Next, a listener will be created to bind the presence of any new line on the serial port (corresponding to a reading of the sensor coming from the Arduino) with the action of adding this value as a new pair to the series. For that we set the x coordinate value as the time when we add the reading in milliseconds (on the chart it will be formatted to HH:mm:ss) and the y coordinate value is a double measurement of the level of light reported by the Arduino in the String `t1`. As the series data could grow indefinitely, to avoid memory issues and to keep track only of the data of the last 30 seconds, we set a `tamMax` value of 300 points and remove the first values of the series if we exceed that limit.

```
listener=(ov, t, t1) -> {
Platform.runLater(()->{
    try{
        String[] data=t1.split(Serial.SEPARATOR);
        if(data.length==2){
            long time=new Long(data[0]);
            luxSeries.getData().add(new Data(time,new Double(data[1])));
            if(luxSeries.getData().size()>tamMax){
                luxSeries.getData().remove(0);
            }
        }
    }
});
```

Note Observe that by using `Platform.runlater()`, not only do we place the task of filling the series with the incoming data in the JavaFX thread, but also we give to the Scene graph the required time to render the chart.

Every time we add data to the series, we have to perform the calculations for the moving averages, as we'll see later. We also check to confirm that the size of each series is within the tamMax limit.

```

if(luxSeries.getData().size()>=tamAve5){
    aveLuxSeries5.getData().add(new Data<>(time, average(tamAve5)));
    if(aveLuxSeries5.getData().size()>tamMax){
        aveLuxSeries5.getData().remove(0);
    }
}
if(luxSeries.getData().size()>=tamAve10){
    aveLuxSeries10.getData().add(new Data<>(time, average(tamAve10)));
    if(aveLuxSeries10.getData().size()>tamMax){
        aveLuxSeries10.getData().remove(0);
    }
}

```

And the listener will be added to serial.getLine():

```
serial.getLine().addListener(listener);
```

Once we've got the chart ready, we'll create the scene, add it to the stage, and finally display it. We create a BorderPane, so in the center we add the chart, and on the bottom we add a Label to show the connection status including the port name. Note the styling done to the root pane.

```

BorderPane root = new BorderPane();
StackPane stack =new StackPane();
stack.setPadding(new Insets(10,0,10,0));
stack.getChildren().add(chart);
root.setCenter(stack);

Label lbl=new Label(serial.getPortName().isEmpty()?"Not connected":
    "Connected to: "+serial.getPortName());
root.setBottom(lbl);
root.setStyle("-fx-background-color: derive(goldenrod,60%); "
    + "-fx-font: 16 \"Courier New\";");

Scene scene = new Scene(root, 800, 600);
primaryStage.setTitle("Arduino Test. Photocell");
primaryStage.setScene(scene);
primaryStage.show();

```

Finally, in the start method we try to open the serial port and start reading from it; connection will be set to true in that case.

```

        startSerial();
    }

private void startSerial(){
    serial.connect();

    connection.set(!serial.getPortName().isEmpty());
}

```

Whenever the application is closed, we remove the listener and call the `serial.disconnect()` method, to shut down the serial connection properly (see Listing 10-2).

```
@Override  
public void stop(){  
    System.out.println("Closing serial port");  
    serial.getLine().removeListener(listener);  
    stopSerial();  
}  
  
private void stopSerial(){  
    if(connection.get()){  
        serial.disconnect();  
        connection.set(false);  
    }  
}
```

To obtain the moving average we'll benefit from the new features in Java 8, such as lambda expressions and bulk data operations for collections. For any given data in the main series `luxSeries`, the moving average is a list of the means of its elements taken in blocks of length `tam`. With `skip()` we go to the last block of the list, with `mapToDouble()` we take from each data its `y` double value, and with `sum()` we get the result of adding all the values in the subsequent collection.

```
private double average(int tam){  
    return luxSeries.getData()  
        .stream()  
        .skip(luxSeries.getData().size()-tam)  
        .mapToDouble(d->d.getYValue().doubleValue())  
        .sum()/tam;  
}
```

To put it simply, when the size of `luxSeries` is `tamAve5`, `average` returns the mean of all the `y` values of the list, and this is the first data point added to `aveLuxSeries5`. When a new reading is added to `luxSeries`, a second data point is added to `aveLuxSeries5`, with the mean of the last `tamAve5` values, and so on.

Running the Example

Now it's time for you to create a new JavaFX project, add the two classes from Listings 10-2 and 10-3, add the `jSSC.jar` file (in case it's not already in the classpath), and build and run the project, with your Arduino board connected with the photocell.

If everything is in place, you should see a moving red line showing up on the chart, and after a while the yellow and green lines of the moving averages, as in Figure 10-15. You can move your hand over the photocell to change the light conditions, and see the effect immediately reflected on the chart.



Figure 10-15. Running the Photocell example

Note You could adjust the 100 ms of delay between readings. If you increase it to 600 ms, for instance, change the labels accordingly, as the first moving average will be from the last thirty seconds, and the second moving average from the last minute.

Adding More Functionality

If you have come this far, you will want to continue to upgrade the application with a few extra features. For that, we'll add at the top of the pane a toolbar with a few buttons to start and stop reading the board, to create a snapshot of the chart, and to switch on or off a Light Emitting Diode (LED) on the circuit board. And at the bottom, another toolbar acting as a status bar will indicate the name of the port, the connection status, and the last value of the readings. This will replace the actual label. We will style the chart and the scene with CSS.

The next listings show only the additional code for the new features. For the full code of this example, please download it from <http://www.apress.com>.

Let's start by adding an LED and a resistor (220 ohm) to the board. We'll use any digital pin on the Arduino, like number 13, to connect one end of the resistor, the other end should be connected to the anode of the LED (long lead), and we have to ground the cathode (short lead, flat side). Note in Figure 10-16 that both photocell and LED are grounded through the blue line of the breadboard (ground rail, shown at the bottom of the breadboard).

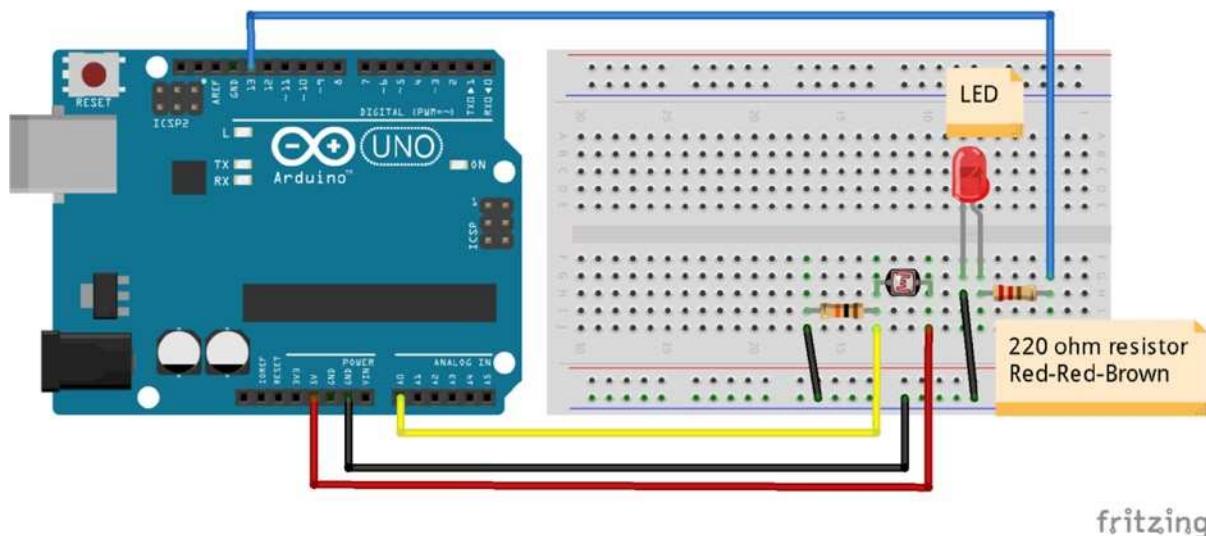


Figure 10-16. LED added to the circuit

Listing 10-4 shows the modifications we have to make to the sketch from Listing 10-1. On every iteration, we listen for any byte on the serial port. If there is any, we read it and turn on or off the LED if it's H (high) or L (low), respectively. Modify your sketch accordingly and upload the changes to the board.

Listing 10-4. Modification in Photocell.ino

```
int pinLED = 13;

void setup() {
  Serial.begin(9600);
  pinMode(pinLED, OUTPUT);
  digitalWrite(pinLED,LOW);
}

void loop() {
...
// 6. listen for commands
if(Serial.available()>0){
  int in=Serial.read();
  if(in=='H'){
    digitalWrite(pinLED,HIGH);
```

```

    } else if(in=='L') {
        digitalWrite(pinLED,LOW);
    }
}

// Adjust delay in ms
delay(100);
}

```

Now we have to allow writing to the serial port in `Serial.java`. The new public method shown in Listing 10-5 will be called from the scene whenever the user clicks a button:

Listing 10-5. Modification in Serial.java

```

public void write(String text){
    try {
        serPort.writeBytes(text.getBytes());
    } catch (SerialPortException ex) {
        System.out.println("ERROR: writing '" + text + "'：" + ex.toString());
    }
}

```

Next we will create a CSS file, named `photocell.css`, with the contents shown in Listing 10-6. The style settings will create this appearance:

- The main series `luxSeries` is plotted as dots, while the moving averages series are shown as lines without symbols. Each series has a custom symbol and custom color, different from the default one. To hide the symbols, a small scale is applied to the shape on the chart, but not on the legend.
- The chart plot area and the legend box have custom background color. The grid lines and the axis lines and ticks have also custom color.

Listing 10-6. Source Code of the CSS File photocell.css

```

.root {
    -fx-background-color: derive(goldenrod,60%);
    -fx-font: 16px "Tahoma";
}

.chart {
    -fx-padding: 0px;
}

.chart-plot-background {
    -fx-background-color: linear-gradient(to bottom,
                                            derive(goldenrod,40%),derive(goldenrod,20%));
}

.axis:bottom {
    -fx-border-color: derive(goldenrod,-20%) transparent transparent transparent;
}

.axis:left {
    -fx-border-color: transparent derive(goldenrod,-20%) transparent transparent;
}

```

```
.axis-tick-mark, .axis-minor-tick-mark,  
.chart-vertical-grid-lines, .chart-horizontal-grid-lines,  
.chart-vertical-zero-line, .chart-horizontal-zero-line {  
    -fx-stroke: derive(goldenrod,-2%);  
}  
.chart-legend {  
    -fx-background-color: linear-gradient(to bottom,  
                                         derive(goldenrod,40%),derive(goldenrod,20%));  
    -fx-padding: 10px;  
}  
.chart-series-line {  
    -fx-stroke-width: 3px;  
}  
.default-color0.chart-series-line { -fx-stroke: transparent; }  
.default-color1.chart-series-line { -fx-stroke: orangered; }  
.default-color2.chart-series-line { -fx-stroke: firebrick; }  
.default-color0.chart-line-symbol {  
    -fx-background-color: transparent, olive;  
    -fx-background-radius: 3px;  
}  
.default-color1.chart-line-symbol {  
    -fx-background-color: orangered, orangered;  
    -fx-background-radius: 2;  
    -fx-scale-x: 0.2;  
    -fx-scale-y: 0.2;  
}  
.default-color2.chart-line-symbol {  
    -fx-background-color: firebrick, firebrick;  
    -fx-background-radius: 2;  
    -fx-scale-x: 0.2;  
    -fx-scale-y: 0.2;  
}  
.default-color1.chart-legend-item-symbol, .default-color2.chart-legend-item-symbol{  
    -fx-scale-x: 0.6;  
    -fx-scale-y: 0.2;  
}  
.tool-bar:horizontal {  
    -fx-background-color: linear-gradient(to bottom,  
                                         derive(goldenrod,40%),derive(goldenrod,20%));  
    -fx-font: 12px "Tahoma";  
}
```

Finally, we have to modify PhotocellFX.java from Listing 10-3. These changes and additions are required:

- Add two global strings to write on the serial port, and three labels for the status bar:

```
private final static String LED_ON = "H";  
private final static String LED_OFF = "L";
```

```
private final Label lblLux=new Label();
private final Label lblLuxAve5=new Label();
private final Label lblLuxAve10=new Label();
```

- Make `setCreateSymbols(true)` to create the symbols on the chart:

```
chart.setCreateSymbols(true);
```

- Add the last value of each series to the labels in the status bar:

```
long time=new Long(data[0]);
luxSeries.getData().add(new Data(time,new Double(data[1])));
lblLux.setText("Light Level: "
    +String.format("%.2f",new Double(data[1])));
...
double av=average(tamAve5);
aveLuxSeries5.getData().add(new Data<>(time, av));
lblLuxAve5.setText("Light Level (Average 5'): "
    +String.format("%.2f",av));
...
double av=average(tamAve10);
aveLuxSeries10.getData().add( new Data<>(time, av));
lblLuxAve10.setText("Light Level (Average 10'): "
    +String.format("%.2f",av));
...
```

- Add a toolbar at the top of the scene, with buttons to start and stop the serial communication, a button to take a snapshot of the chart and a toggle button to switch on or off the LED.

```
BorderPane root = new BorderPane();

Button btnStart=new Button("Start");
btnStart.setOnAction(t->startSerial());
btnStart.disableProperty().bind(connection);

Button btnStop=new Button("Stop");
btnStop.setOnAction(t->stopSerial());
btnStop.disableProperty().bind(connection.not());

Button btnSnap=new Button("Snap");
btnSnap.setOnAction(t-> saveSnapshot() );

ToggleButton tglLed=new ToggleButton("Led");
tglLed.disableProperty().bind(connection.not());
tglLed.selectedProperty().addListener((ov, t, t1) -> serial.write(t1?LED_ON:LED_OFF));

ToolBar tUp=new ToolBar();
tUp.getItems().addAll(btnStart,btnStop,new Separator(),btnSnap,new Separator(),tglLed);
root.setTop(tUp);
```

- Add a toolbar at the bottom with labels to show the status and the values of the readings.

```

Label lbl=new Label("Not connected");
connection.addListener((ov, b, b1)->lbl.setText(b1?"Connected to: "
+serial.getPortName():"Not connected"));
lbl.setPrefWidth(250);

lblLux.setPrefWidth(120);
lblLuxAve5.setPrefWidth(180);
lblLuxAve10.setPrefWidth(180);

ToolBar tBottom=new ToolBar();
tBottom.getItems().addAll(lbl,new Separator(),lblLux,new Separator(),
lblLuxAve5,new Separator(),lblLuxAve10);

root.setBottom(tBottom);

```

- Remove the styling from root and load the CSS file from Listing 10-6:

```

Scene scene = new Scene(root, 800, 600);
scene.getStylesheets().add(getClass().getResource("photocell.css").toExternalForm());

```

- Modify the title, and remove the startSerial() call:

```

primaryStage.setTitle("Arduino Test 2. Photocell&LED");
primaryStage.setScene(scene);
primaryStage.show();
}

```

- Finally, add a method to capture a snapshot of the chart, and provide a way to save it to a file in the same path of the application:

```

private void saveSnapshot(){
    System.out.println("Saving snapshot... ");
    final WritableImage image = chart.snapshot(null, null);
    final BufferedImage fromFXImage = SwingFXUtils.fromFXImage(image,null);
    if(fromFXImage!=null){
        try {
            ImageIO.write(fromFXImage, "png", new File("image.png"));
        } catch (IOException ex) {
            System.out.println("error img: "+ex.toString());
        }
    }
}

```

After you've applied all changes, build and run the project, and you will see the new features in the application, as shown in Figure 10-17. Try to start and stop the serial communication, and observe the status bar. Capture an image of the chart and try toggling the LED on and off.



Figure 10-17. Upgraded example running

Now, What?

There are plenty of resources out there to demonstrate or utilize the Arduino, and you can experiment whatever you have in mind! Arduino and the required equipment are inexpensive and safe (in most cases), and they give you the chance to improve your do-it-yourself (DIY) skills.

So look for an affordable project, buy what you need, build it, test it first on the Arduino IDE, and then design a JavaFX application to monitor and control your devices.

Share your developments and look for help when you need it; there are great communities of people willing to help, in both Arduino development and JavaFX.

Summary

In this chapter you've discovered what can be done by combining Arduino and JavaFX. You began by learning about Arduino, its different boards and main specifications. Next, you saw how to communicate with it from the Arduino IDE. After learning about loading examples in the IDE, you got the chance to try it for yourself building a simple circuit with an Arduino Uno and a photocell, creating a sketch and reading the light levels in the Serial Monitor.

Then you learned about the different ways to read a serial port from your computer and found about the Java Simple Serial Connector library. Installing this third-party component allowed you to perform serial port operations such as locating, opening, reading, writing and closing.

To monitor and display the Arduino readings, you learned how to use the JavaFX Charts API. Then you saw an example in which a JavaFX chart is plotted with a series of data coming from the serial port, using an Arduino board to measure the level of light with a photocell. Lambdas and bulk data operations for collections were employed to perform statistical analysis. Finally, you discovered how to enhance the application by adding toolbars with buttons and labels, and styling the scene with CSS.

JavaFX on the Raspberry Pi

The Internet of things (IoT) has become a hot topic of late, and for good reason. Some estimates put the number of connected small devices somewhere around 9 billion currently, jumping to 24 billion devices by 2020. And while projections vary, the assessment really doesn't: in terms of sheer numbers, the Internet of things is going to dwarf any computing models ever seen.

The devices themselves are often quite small and much less powerful than recent examples of the processing "arms race" to which many of us have grown accustomed (numb?) over the past several years. Even when embedded within a larger appliance or piece of equipment, the tiny computer that controls and communicates may be no larger than your hand... and perhaps as small as your fingernail. These tiny devices are *resource-constrained* due largely to space, power, and/or connectivity requirements and restrictions, necessitating a different approach to developing and deploying software and often different tools and tool chains than are available for larger, more powerful platforms.

The power of the IoT isn't in each device as much as it is in pervasiveness and connectedness.

Java was *made* for the Internet of things—literally. The vision of its creators was for Java to control small devices like television set-top boxes, and with the explosion of the Internet of things, Java returns to its roots. You may think that JavaFX, being a platform for rich-client development, would miss the IoT party... but not so! According to the JavaFX Overview page on the Oracle Technology Network, "It is designed to provide a lightweight, hardware-accelerated Java UI platform..."

(<http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html>).

That statement holds the key point for the graphically rich-and-powerful JavaFX: hardware acceleration. How large would a computer have to be to accommodate JavaFX's power and might? As it turns out, about the size of a deck of playing cards!

The Raspberry Pi has been called the "Commodore 64 of our era." Back in the 1980s and 1990s, many aspiring developers cut their programming teeth on the C64 because it was widely available and inexpensive. As a result, it had excellent software support—including programming tools/languages—and a vibrant do-it-yourself community. Many industry leaders have lamented over the years that the current generation of budding computer scientists have lacked inexpensive and widely available platforms with which to play, experiment, and innovate. There are many answers to that complaint, but the Raspberry Pi Foundation may have provided the best ones: the Raspberry Pi Model A sells for around \$25 (USD) and the Model B for approximately \$35, and both are extremely capable Linux computers. Figure 11-1 will give you an idea of what a Raspberry Pi in the wild looks like.



Figure 11-1. Raspberry Pi Models A (left) and B (right)

Let's take a look at the technical specifications and consider why such an unassuming device merits a chapter in this book. Table 11-1 shows both models' key features for comparison.

Table 11-1. Raspberry Pi Specifications

Raspberry Pi	Model A	Model B
Chip	Broadcom BCM2835 SoC full HD multimedia applications processor	
CPU	700 MHz Low Power ARM1176JZ-F Applications Processor	
GPU	Dual Core VideoCore IV® Multimedia Co-processor	
Memory	256M SDRAM	512M SDRAM
Ethernet	None	Onboard 10/100 Ethernet RJ45 jack
USB 2.0	1 port	2 ports
Video Output	HDMI, Composite RCA (PAL & NTSC)	
Audio Output	3.5mm jack, HDMI	
Onboard Storage	SD, MMC, SDIO card slot	
Operating System	Linux (various)	
Dimensions	8.6cm x 5.4cm x 1.5cm	8.6cm x 5.4cm x 1.7cm

Based upon the same foundation, both boards are very similar, with the key differences highlighted in Table 11-1. For only \$10 more, the Model B offers a great deal more than the Model A: accessory/network connectivity out of the box and perhaps most importantly, twice the memory! From here forward, we will focus upon the Model B, as setup is dramatically simplified by its onboard Ethernet... and the extra memory is never a bad thing!

The deeper back story of the Raspberry Pi makes for interesting reading, but for now, let's focus upon what makes the Raspberry Pi such a perfect fit for Java and JavaFX:

- It is faster than the specifications seem to indicate. As noted, the default clock speed for the Pi is 700MHz, but the Raspberry Pi Foundation has released guidance for safely overclocking to speeds up to 1GHz.
- Very wisely, the Foundation chose a CPU with floating-point support, often referred to as “hard float,” which offers greater performance than so-called “soft float only” chips, and operating systems available for the Pi take advantage of that extra power and speed.
- Finally, the Pi has onboard a fairly beefy Graphics Processing Unit (GPU) with a fast 3D core that is “capable of BluRay quality playback, using H.264 at 40MBits/s” (<http://www.raspberrypi.org/faqs#generalSoCUsed>).

Clearly, there is more to the Pi than meets the eye.

Preparing the Raspberry Pi for JavaFX

Before we can begin developing JavaFX applications for the Raspberry Pi and deploying them, we must first configure the Pi. For the purposes of this exercise, you'll need a Raspberry Pi, a suitable power adapter (I recommend 1.2A, 5v), an 8+ Gigabyte SD card (used by the Pi for storage; use a Class 10 SD card for best results), a USB keyboard and mouse, and an HDMI cable and HDMI-capable TV/monitor. Optional: an RJ45 Ethernet cable, if you'd like to connect the Pi to your network via a wired connection. Here are the steps to follow to get your Raspberry Pi “JavaFX-ready.”

Create a Bootable SD Card

The Pi uses the SD card as a hard drive, loading the operating system from it and reading and writing files from/to it. This configuration is conceptually similar to a larger Linux computer with a Solid State Drive (SSD), and it serves the Pi well. To create a bootable SD card, we must download a Pi-compatible Linux distribution (distro) and install it to the card. The following steps detail the easiest way to accomplish that.

1. Download to your development machine the SD Association Formatter tool at https://www.sdcard.org/downloads/formatter_4/. We'll use this to prepare the SD card (Mac and Windows; if using a Linux machine for development, use parted/gparted).
2. Install the SD Association Formatter tool to your development machine.
3. Format the SD card using the SD Association Formatter tool (Mac/Windows). To do this, simply insert the SD card into a reader attached to your computer, run the SD Association Formatter, select the SD card to format (be sure to select the correct one!), and click the Format button. Figure 11-2 shows an example of the SD Association Formatter tool ready to format the SD card mapped to Drive F.

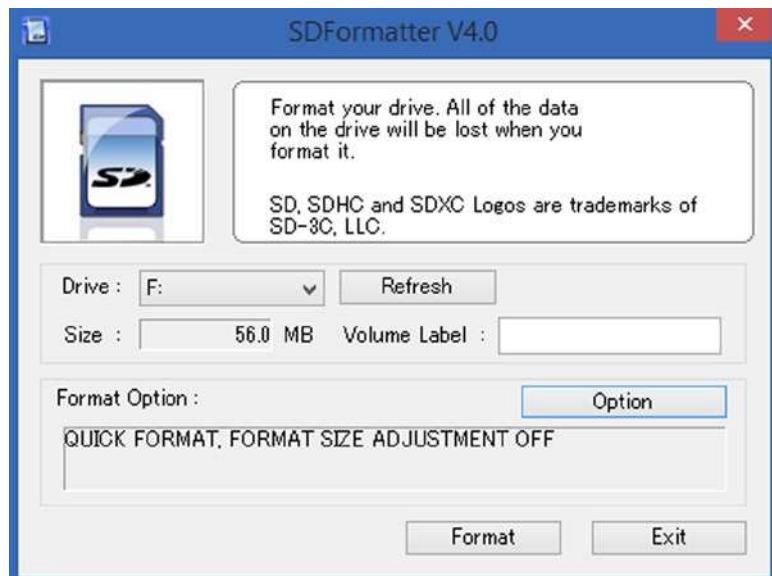


Figure 11-2. The SD Association Formatter tool

4. Download the New Out Of Box Software, or NOOBS (offline and network install), from <http://www.raspberrypi.org/downloads> as shown in Figure 11-3. This will enable us to install our choice of operating system with the least fuss.

Image	NOOBS_v1_3_4.zip
Torrent	NOOBS_v1_3_4.zip.torrent
SHA-1 Checksum	96b7a28e7b3bc7d73c93ff2207ede0582fe29525
Release Date	2014-01-09
Version	1.3.4
Documentation	Link

Figure 11-3. NOOBS download

5. Unzip the downloaded NOOBS file.
6. Copy the extracted NOOBS files to the SD card.

7. Eject the SD card from your dev machine and place it in your Raspberry Pi's card slot.
8. With keyboard, mouse, and HDMI cable (to monitor or TV) attached (Ethernet cable optional), connect the power cable to your Raspberry Pi to boot NOOBS.
9. When prompted for which operating system to install, choose Raspbian.
10. Go get some coffee and let NOOBS work its magic!

When finished, your Raspberry Pi will reboot into Raspbian Linux, a Debian-based Linux tailored for the Pi. Next, we'll want to make a few adjustments to the base configuration to tailor it for our purposes.

Configuring the Raspberry Pi

When Raspbian first boots, it presents the `raspi-config` tool, allowing us to make changes to the basic Raspbian installation. We'll want to do a few things to optimize for JavaFX.

1. First, we make the Raspberry Pi feel at home in its new home. Select option 4, Internationalisation Options, then option I2, Change Timezone, and then follow the prompts to choose your local timezone.

Note The Raspberry Pi Foundation is a UK-based organization, so those accustomed to the American spelling of certain words—like “Internationalization”—may occasionally see something that looks a bit unusual in `raspi-config`. Feel free to “autocorrect” mentally whilst/while reading.

2. As noted earlier, it's possible—and even encouraged—to overclock the Pi for better performance. Experience indicates that there is a limit to the pro/con balance, and that limit lies just short of 1000MHz (turbo). Even the best SD cards seem prone to corruption at that setting. . . but at the 950 MHz (high) setting, corruption is rare when using a quality SD card under normal conditions. To overclock, select option 7, Overclock, choose OK to accept the warning, and then 950 MHz (high). Choose OK to continue.
3. In order to access the Pi remotely—not absolutely necessary, but strongly recommended—we need to select option 8, Advanced Options, then option A4, SSH, and select the Enable button. Choose OK to continue.
4. The `raspi-config` tool also provides an easy way to adjust the memory split between the CPU and the GPU—a good thing indeed, since the default split is a bit miserly when allocating memory to the GPU! In order to run our JavaFX applications well, simply select option 8, Advanced Options, and then option A3, Memory Split. Minimum recommended GPU allocation is 128M, but since we're focusing upon graphics, let's make that 256M for an even CPU/GPU split.

5. (Optional) You can change the name your Raspberry Pi uses by selecting option 8, Advanced Options, and then A2, Hostname. This is convenient when you have only one Pi on the network, but it's a near-necessity when you add a second... and third...
6. Choose the Finish button to exit the raspi-config tool, and Yes when asked if you would like to reboot now.
7. Once the Pi has rebooted, if you did connect an Ethernet cable earlier, make a note of the number that shows on the line saying "My IP address is" just before the login prompt; you'll need this for subsequent sections. To log into the Pi, just type "pi" (with the quotes) at the raspberrypi login prompt, and then "raspberry" (again, with quotes) for the password. You're in!

Configuring a Fixed IP Address

Giving your Raspberry Pi a fixed network address isn't required, but it is strongly recommended. Doing so means you always connect to your Pi using the same address (or name, if you create an entry in your hosts file), and it thus removes one potential variable from your development process.

It's also a good idea to update your network DHCP device/router with the Pi's IP address so that it doesn't try to assign it to another device on your network. The steps required to do so will vary by switch/router manufacturer.

For Wired Networking (Optional, but STRONGLY RECOMMENDED)

If you plan to continue using your wired Ethernet connection to connect your Pi to your network, simply follow these steps to give it a permanent address:

1. From your Raspberry Pi's command prompt, type `ifconfig -a` and `netstat -nr` to gather the following information: current IP (if you want to keep it), netmask, gateway, destination, and broadcast. Jot these down, you'll need them!
2. On the Pi, make a backup of `/etc/network/interfaces` by running

```
sudo cp /etc/network/interfaces /etc/network/interfaces.orig
```

This command will allow you to revert easily to the previous settings if needed.

Note The sudo command, or Super User "do," executes the command immediately following it with Super User (or "root") privileges. In the preceding step, we ran the `cp` (copy) command as the Super User to make a backup copy of our network interfaces configuration.

3. Using your favorite text editor, modify the `interfaces` file. For `vi`, the syntax is `sudo vi /etc/network/interfaces`, but `nano`, `pico`, and so on are also available options. Use whichever editor you're comfortable with, and if you aren't (yet) comfortable with one, this is an excellent opportunity to learn!

4. Change the line that reads

```
iface eth0 inet dhcp
```

to

```
iface eth0 inet static
```

and add the following lines:

`address 192.168.1.nnn` (this is the IP address you want the Pi to have and keep)

`netmask 255.255.255.0` (or whatever was shown by `ifconfig -a` in step 1)

`network 192.168.1.0` (or whatever was in the Destination column in the second line of output from `netstat -nr` in step 1)

`broadcast 192.168.1.255` (as shown in `ifconfig -a`)

`gateway 192.168.1.1` (or what was shown in either `ifconfig` or `netstat`)

5. Save the file.

6. Reboot the Pi by typing `sudo reboot` at the command prompt.

For Wireless Networking (Optional)

If you buy a WiFi USB adapter for your Raspberry Pi and want to configure it with a fixed IP address, the steps are similar to those just shown for wired networking.

Note For best results, please refer to http://elinux.org/RPi_USB_Wi-Fi_Adapters for WiFi adapters that have been tested and found to work best with the Raspberry Pi. Personal favorite: Edimax EW-7811un. It's detected automatically by current Raspbian builds and just works.

Once you've installed the WiFi adapter and confirmed that it works using DHCP, you can assign it a fixed IP address by completing the steps in the "For Wired Networking" section earlier, with the following modifications:

1. In step 4, either `iface eth0` entry listed in the `/etc/network/interfaces` file is fine; those affect only the Ethernet (`eth0`) port and not the WiFi adapter.
2. Change the line that reads

- ```
allow-hotplug wlan0
to
auto wlan0
```
3. Change the line that reads
- ```
iface wlan0 inet manual
to
iface wlan0 inet static
```
4. Comment out the following entry by placing a pound sign/hash (#) at the beginning of the line:
- ```
wpa-roam /etc/wpa-supplicant/wpa_supplicant.conf
```
5. Below the entries for address, network, netmask, broadcast, and gateway you made already (as noted in step 4 in the “For Wired Networking” section), add the following lines:
- ```
wpa-ssid "MyNetworkSSID"
wpa-psk "MySecretKey123"
```

Note VERY IMPORTANT: When specifying the SSID and password in step 5, be sure to include the quotation marks (double quotes).

6. Save the file.
7. Reboot the Pi by typing `sudo reboot` at the command prompt.

Note If you have configured a WiFi adapter, you’ve already either decided to remove the mouse to free a USB port or have added a USB hub to the mix. Either option is fine as long as your power adapter provides 1–1.2A of current. If your adapter supplies less than 1A, your WiFi connection will likely be intermittent, at best. To fix (or avoid) this, simply upgrade the power adapter or add a powered USB hub.

At this point, you no longer require the keyboard, mouse, and display on a consistent basis. It’s always good to be able to connect them if desired—and you may want (or need) to have them connected to support a future JavaFX Pi program you create—but from this point forward, we’ll be doing most of our work remotely. If you wish to unplug them and go “headless,” simply power down the Pi by typing `sudo shutdown -h now` and, once shutdown is complete, unplug the HDMI, keyboard, and mouse cables; remove the Pi’s power plug; and reconnect the power.

Connecting to the Pi Remotely

Earlier, we used `raspi-config` to enable an SSH server on the Pi so we could connect to it remotely. Configuring your development machine to do so with an SSH client is fairly painless and will make your life a lot easier, even allowing you to do some cool things that wouldn't be possible otherwise. So let's dive in!

Setting up Secure Shell (SSH)

If you're using a Mac or Linux machine for your dev box, there really is very little you must do to connect to your Raspberry Pi via SSH; simply open a terminal window and you're off and running. Windows requires just a bit more effort. I'll point out the differences as we go.

1. (Windows) Windows doesn't include native SSH or SCP (Secure Copy) utilities, but PuTTY—a free utility—fills that gap nicely. Just download the "Windows installer for everything except PuTTYtel" from <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html> and install it to your system.
2. (Windows) Run PuTTY (`putty.exe`). In the PuTTY Configuration window (see Figure 11-4), do the following to create and save a connection (for repeated use) to your Pi:
 - a. Enter the Pi's IP address (or hostname if you've entered that in your dev machine's `hosts` file) in the Host Name (or IP Address) textbox.
 - b. Enter **Raspberry Pi** (or whatever you like) in the Saved Sessions textbox.
 - c. Click the Save button so you can easily reload the connection information and connect in the future.
 - d. Click the Open button to connect via SSH to your Pi!

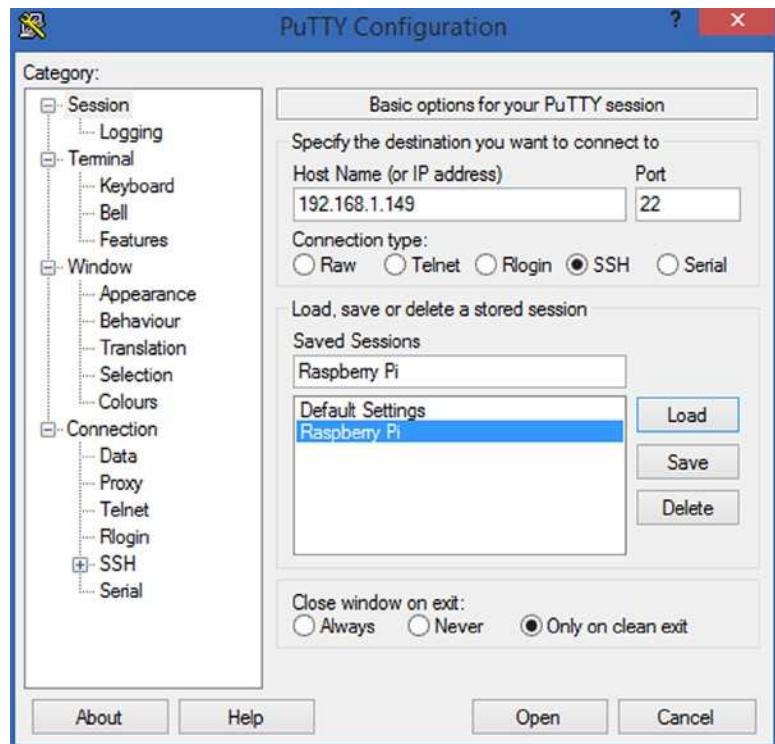


Figure 11-4. PuTTY Configuration window

1. (Mac/Linux) Open a terminal window and type the following at a command prompt:

```
ssh 192.168.1.nnn (using the IP address of your Pi if you don't have an entry  
for it in your dev machine's hosts file) or
```

```
ssh raspberrypi (or however you referred to it in your local hosts file)
```

2. (All platforms) If presented with a warning about a new host key, choose Yes to update your SSH keys with one for your Pi. An example of this warning is shown in Figure 11-5.

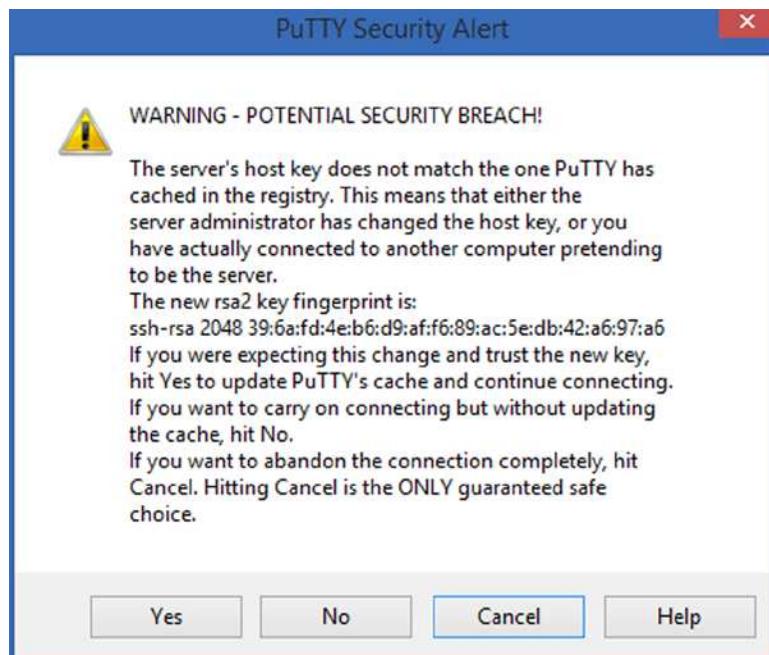


Figure 11-5. Host key warning

- When prompted, provide your Pi user name (*pi*) and password (*raspberry*). You should now be connected remotely, via SSH, to your Raspberry Pi and see something similar to the terminal window shown in Figure 11-6.

A screenshot of a terminal window titled "pi@pifx: ~". The window displays a standard Debian Linux login sequence:

```
login as: pi
pi@192.168.1.149's password:
Linux pifx 3.10.25+ #622 PREEMPT Fri Jan 3 18:41:00 GMT 2014 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Mar  4 00:13:14 2014
pi@pifx ~
```

The terminal prompt "pi@pifx ~" is visible at the bottom.

Figure 11-6. Connected!

Installing JDK8

By the time you read this, it's very possible that JDK8 will ship preinstalled with Raspbian, and if you've loaded a version of Raspbian in which that is the case, congratulations! You can skip this section and get right into the code. To see if that is the case, simply type the following at your Pi command prompt:

```
pi@raspberrypi ~ $ java -version
```

You will see something like this, depending upon the version that is currently installed and accessible:

```
java version "1.8.0"
Java(TM) SE Runtime Environment (build 1.8.0-b129)
Java HotSpot(TM) Client VM (build 25.0-b69, mixed mode)
```

The important bit is the first line: if it doesn't say "1.8.n," you'll need to install JDK8.

Installing JDK8 on Your Development Machine

It's important to install the same JDK build on your development machine *and* your Raspberry Pi to avoid library/versioning issues. We'll start with your dev box and then move to the Pi.

To download JDK8, point your browser to <https://jdk8.java.net/download.html>, review the linked license agreement, click the radio button to Accept License Agreement, and choose the JDK download that matches the operating system on your dev box.

Once the download finishes, run the installer (Mac, Windows) to install JDK8 to your machine. If you're running Linux, you'll need to follow a process very similar to the one described next for installing JDK8 to your Raspberry Pi.

Installing JDK8 on the Raspberry Pi

There are a few different ways to install a new JDK to your Raspberry Pi, but this is perhaps the fastest and simplest way to get to code...and even it requires several steps. So let's begin!

1. On your development machine, again point your browser to <https://jdk8.java.net/download.html>, click the radio button to Accept License Agreement (you *did* read the license agreement earlier, right?), and choose the JDK download labeled Linux ARMv6/7 VFP, HardFP ABI (translation: ARM chips versions 6 or 7, Vector Floating Point, Hard Floating Point Application Binary Interface). Download the file to a directory on your dev box.

-
2. Open a terminal (Mac, Linux) or command/cmd (Windows) window and cd to the directory containing the downloaded JDK8 file, as in the following:

(Mac, Linux)

```
Marks-MacBook-Pro:~ mark$ cd Downloads/  
Marks-MacBook-Pro:Downloads mark$
```

(Windows)

```
C:\Users\Mark>cd Downloads  
C:\Users\Mark\Downloads>
```

3. Confirm that the file is there, as expected; your display should be similar to one of the following:

(Mac, Linux)

```
Marks-MacBook-Pro:Downloads mark$ ls jdk-8*  
jdk-8-fcs-b129-linux-arm-vfp-hf1t-06_feb_2014.tar.gz
```

(Windows)

```
C:\Users\Mark\Downloads>dir jdk-8*  
Volume in drive C is XXnnnnnnnnX  
Volume Serial Number is XXnn-nnXX
```

```
Directory of C:\Users\Mark\Downloads\
```

```
03/01/2014 12:01 AM 87,454,973 jdk-8-fcs-b129-linux-arm-vfp-hf1t-06_  
feb_2014.tar.gz
```

4. Transfer the file to the Pi using the Secure Copy (scp) utility:

(Mac, Linux)

```
Marks-MacBook-Pro:Downloads mark$ scp jdk-8-fcs-b129-linux-arm-vfp-hf1t-06_  
feb_2014.tar.gz pi@raspberrypi:.
```

(Windows, using PuTTY's scp utility pscp)

```
C:\Users\Mark\Downloads>"C:\Program Files (x86)\PuTTY\pscp.exe"  
jdk-8-fcs-linux-arm-vfp-hf1t-06_feb_2014.tar.gz pi@raspberrypi:.
```

When prompted, enter **raspberry** for the password. This securely copies the file to the pi account's home directory on the Raspberry Pi, as represented by the “.” character.

5. Now that we've copied our new JDK8 to the Pi, we must install it. SSH to the Pi from your dev box as before (from a terminal window on Mac/Linux or via PuTTY on Windows) to complete the remaining steps.

6. Extract the JDK to a directory of your choosing, using sudo to create a new directory if necessary:

```
sudo tar -zxvf jdk-8-fcs* -C /opt
```

7. Using your favorite editor once more, you'll need to edit your `.bashrc` file to properly set the pi user's environment for JDK8. If you are still in your pi user's home directory (and if not, simply type `cd` and press Enter to return there), you can make the required changes by using a command similar to `vi .bashrc` (using `vi` as an example), add the following lines to the bottom of the file:

```
export JAVA_HOME=/opt/jdk1.8.0  
export PATH=$JAVA_HOME/bin:$PATH
```

8. Save the file.
9. Reboot the Pi by typing `sudo reboot` at the command prompt.
10. Once you've reconnected to the Pi via SSH, verify the new environment by typing the following command:

```
pi@raspberrypi ~ $ java -version
```

You will see something like this, depending upon the version that is currently installed and accessible:

```
java version "1.8.0"  
Java(TM) SE Runtime Environment (build 1.8.0-b129)  
Java HotSpot(TM) Client VM (build 25.0-b69, mixed mode)
```

With JDK8 in place, we're nearly ready! The last thing we must do to take full advantage of the power of JavaFX 8 is install an Integrated Development Environment (IDE) that is JDK8-aware on our development machine.

Installing and Configuring NetBeans 8 for Raspberry Pi Development

As mentioned earlier in this book, an IDE is not, strictly speaking, required to write code. But an IDE—and especially one that fully supports JDK8 and is as feature-rich as NetBeans 8—enables you to do some things that are simply impossible from the command line, as you will soon see.

Installing NetBeans 8

If you've already installed NetBeans 8, please skip this section. If not, here are the steps you'll need to take:

1. Point your browser to <https://netbeans.org/downloads/> and select version 8.0 in the top-right portion of the screen. Choose any NetBeans download bundle that includes JavaFX (Java SE, Java EE, and All are valid options) and click the Download button below it, saving it to the directory of choice on your development machine.
2. Run the installer, providing the requested information when prompted.
3. NetBeans itself is a Java application, and NetBeans 8 should run on JDK 8 for best results. To verify, load NetBeans and choose the menu item titled NetBeans, and then About NetBeans. The entries for Java and Runtime should indicate version 1.8.0.

Configuring NetBeans 8 for the Pi

NetBeans 8 adds the capability to point to a remote JDK and use it to debug and execute programs remotely that you write locally, on your development machine. It even deploys your applications automatically and seamlessly. As documented by copresenter and coauthor José Pereda in his article at <http://netbeans.dzone.com/articles/nb-8-raspberry-pi-end2end>, you can enable this capability by doing the following:

1. If it is not already running, start NetBeans on your development machine.
2. Choose Tools from the menu bar, and then Java Platforms.
3. Click the Add Platform button.
4. Select the Remote Java Standard Edition radio button and click Next.
5. Provide the following entries (see Figure 11-7 for an example):

Platform Name: Raspberry Pi JDK 8

Host: IP address or hostname (if listed in your dev box's /etc/hosts file) of your Raspberry Pi

Username: pi

Password: raspberry

Remote JRE Path: /opt/jdk1.8.0/jre

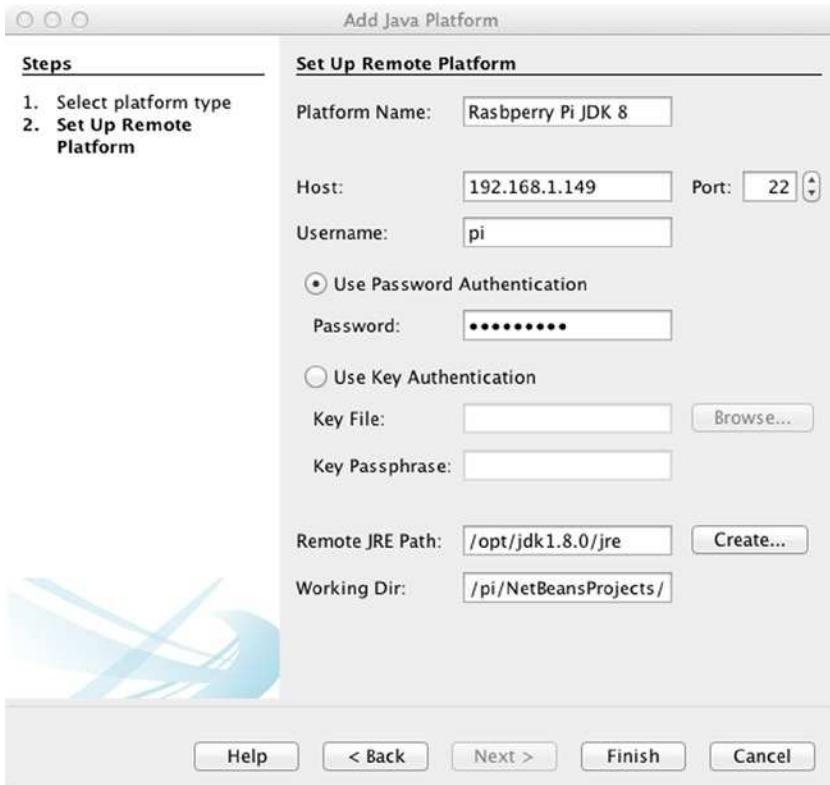


Figure 11-7. Configuring the remote JDK platform in NetBeans

6. Click the Finish button and wait for NetBeans to establish and configure the remote JDK connection.
7. Click the Close button once the remote JDK is in place.

Now that we've completed the setup, you should have a development environment that is among the best available to develop JavaFX 8 applications for the Raspberry Pi. So let's get started!

RaspiCycle: The Code

To demonstrate the “ins and outs” of developing JavaFX 8 code for and deploying it to the Raspberry Pi, I’ve created a fairly simple—but visually and programmatically interesting, I hope!—example program. Listing 11-1 contains the entirety of a very basic Pi-focused game that pays homage to the groundbreaking movie and videogame *TRON*, in particular the light cycles that characters rode.

This game will allow you to control your yellow (as in the movie, not the video game) light cycle and move it about the Pi’s display—the “Game Grid”—until you crash into one of your cycle’s lingering light trails or an outer wall. The name TRON is actually an old BASIC command formed by combining the words “TRace ON,” which prompted the program to display line numbers as it ran to aid in

debugging. (Bonus tidbit: the command to disable the trace was `TR0FF`.) Rather than line numbers, I've placed a couple of console output statements to show current coordinates and the result of a collision.

Ready to start with the code? Well. . . Greetings, Program, and welcome to the Game Grid!

Listing 11-1. Source code for RaspiCycle.java

```
package jfx8ibe;

import java.util.ArrayList;
import java.util.List;
import javafx.animation.AnimationTimer;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.geometry.Point2D;
import javafx.geometry.Rectangle2D;
import javafx.scene.Cursor;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.stage.Screen;
import javafx.stage.Stage;

/**
 *
 * @author Mark Heckler, @MkHeck
 */
public class RaspiCycle extends Application {
    private int SPEED = 1;
    private double SCREENHEIGHT;
    private double SCREENWIDTH;
    private Canvas canvas;
    private GraphicsContext gc;
    private Point2D startPos;
    private Point2D curPos;
    private Point2D newPos;
    private final List<Line2D> walls = new ArrayList<>();

    public enum Direction {
        LEFT, RIGHT, UP, DOWN
    };
    private Direction curDir = Direction.UP;

    private AnimationTimer animTimer;
```

```
@Override
public void start(Stage primaryStage) {
    StackPane root = new StackPane();

    Rectangle2D bounds = Screen.getPrimary().getVisualBounds();
    SCREENHEIGHT = bounds.getHeight();
    SCREENWIDTH = bounds.getWidth();

    // Add screen boundaries to list of walls (coordinates)
    // to check for collisions

    // Top wall
    walls.add(getLine(bounds.getMinX(), bounds.getMinY(),
        bounds.getMaxX(), bounds.getMinY()));

    // Right wall
    walls.add(getLine(bounds.getMaxX(), bounds.getMinY(),
        bounds.getMaxX(), bounds.getMaxY()));

    // Bottom wall
    walls.add(getLine(bounds.getMinX(), bounds.getMaxY(),
        bounds.getMaxX(), bounds.getMaxY()));

    // Left wall
    walls.add(getLine(bounds.getMinX(), bounds.getMinY(),
        bounds.getMinX(), bounds.getMaxY()));

    // Define starting point for our Light Cycle (bottom center)
    startPos = curPos = new Point2D(SCRENNWIDTH / 2, SCREENHEIGHT - 10);

    // Prepare the Game Grid (screen)
    canvas = new Canvas(SCRENNWIDTH, SCREENHEIGHT);
    gc = canvas.getGraphicsContext2D();

    gc.setFill(Color.rgb(2, 2, 47));
    gc.fillRect(0, 0, SCREENWIDTH, SCREENHEIGHT);
    drawGameGrid();
    drawWalls(bounds);
    root.getChildren().add(canvas);

    Scene scene = new Scene(root, SCREENWIDTH, SCREENHEIGHT);
    scene.setCursor(Cursor.NONE);

    scene.setOnKeyPressed( (KeyEvent event) -> {
        handleKeyPress(event);
    });

    primaryStage.setTitle("RaspiCycle");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

```
animTimer = new AnimationTimer() {
    @Override
    public void handle(long now) {
        runLightCycle();
    }
};

animTimer.start();

}

/** 
 * The main() method is ignored in correctly deployed JavaFX application.
 * main() serves only as fallback in case the application can not be
 * launched through deployment artifacts, e.g., in IDEs with limited FX
 * support. NetBeans ignores main().
 *
 * @param args the command line arguments
 */
public static void main(String[] args) {
    launch(args);
}

private void drawWalls(Rectangle2D bounds) {
    gc.setStroke(Color.LIGHTBLUE);
    gc.setLineWidth(2);

    // Top wall
    gc.strokeLine(bounds.getMinX(), bounds.getMinY(),
                  bounds.getMaxX(), bounds.getMinY());

    // Right wall
    gc.strokeLine(bounds.getMaxX(), bounds.getMinY(),
                  bounds.getMaxX(), bounds.getMaxY());

    // Bottom wall
    gc.strokeLine(bounds.getMinX(), bounds.getMaxY(),
                  bounds.getMaxX(), bounds.getMaxY());

    // Left wall
    gc.strokeLine(bounds.getMinX(), bounds.getMinY(),
                  bounds.getMinX(), bounds.getMaxY());
}

private void drawGameGrid() {
    int boxSize = (int) (Math.min(SCREENWIDTH, SCREENHEIGHT) / 10);
    int i;
    gc.setStroke(Color.BLACK);
    gc.setLineWidth(5);
```

```
// Draw horizontal lines
i = boxSize;
while (i < SCREENHEIGHT) {
    gc.strokeLine(0, i, SCREENWIDTH, i);
    i += boxSize;
}

// Draw vertical lines
i = boxSize;
while (i < SCREENWIDTH) {
    gc.strokeLine(i, 0, i, SCREENHEIGHT);
    i += boxSize;
}
}

private Line getLine(double x1, double y1, double x2, double y2) {
    return new Line(Math.min(x1, x2), Math.min(y1, y2),
                    Math.max(x1, x2), Math.max(y1, y2));
}

public void handleKeyPress(KeyEvent event) {
    boolean isNewDir = false;

    if (event.getCode() == KeyCode.LEFT &&
        curDir != Direction.LEFT) {
        curDir = Direction.LEFT;
        isNewDir = true;
    }
    else if (event.getCode() == KeyCode.RIGHT &&
              curDir != Direction.RIGHT) {
        curDir = Direction.RIGHT;
        isNewDir = true;
    }
    else if (event.getCode() == KeyCode.UP &&
              curDir != Direction.UP) {
        curDir = Direction.UP;
        isNewDir = true;
    }
    else if (event.getCode() == KeyCode.DOWN &&
              curDir != Direction.DOWN) {
        curDir = Direction.DOWN;
        isNewDir = true;
    }
    else if (event.getCode() == KeyCode.DIGIT1) {
        SPEED = 1;
    }
    else if (event.getCode() == KeyCode.DIGIT2) {
        SPEED = 2;
    }
    else if (event.getCode() == KeyCode.DIGIT3) {
        SPEED = 3;
    }
}
```

```
        else if (event.getCode() == KeyCode.ESCAPE) {
            animTimer.stop();
            Platform.exit();
        }

        if (isNewDir) {
            // User sent Light Cycle in a new direction...
            // add the wall (light trail) to the list & start a new one
            walls.add(getLine(startPos.getX(), startPos.getY(),
                curPos.getX(), curPos.getY()));
            startPos = curPos;
        }
    }

    private void runLightCycle() {
        calculateDestination();
        gc.setStroke(Color.YELLOW);
        gc.setLineWidth(5);
        gc.strokeLine(curPos.getX(), curPos.getY(), newPos.getX(), newPos.getY());
        curPos = newPos;
        checkCollision();
    }

    private void calculateDestination() {
        double newX = 0;
        double newY = 0;

        switch (curDir) {
            case UP:
                // X doesn't change, only Y
                newX = curPos.getX();
                newY = curPos.getY() - SPEED;
                break;
            case DOWN:
                // X doesn't change, only Y
                newX = curPos.getX();
                newY = curPos.getY() + SPEED;
                break;
            case LEFT:
                // Y doesn't change, only X
                newY = curPos.getY();
                newX = curPos.getX() - SPEED;
                break;
            case RIGHT:
                // Y doesn't change, only X
                newY = curPos.getY();
                newX = curPos.getX() + SPEED;
                break;
        }
        newPos = new Point2D(newX, newY);
    }
}
```

```

private void checkCollision() {
    System.out.println("Current Position: (" + curPos.getX() + ", " + curPos.getY() + ")");

    walls.stream().filter((line) -> (
        (line.getStartX() <= curPos.getX() && curPos.getX() <= line.getEndX()) &&
        (line.getStartY() <= curPos.getY() && curPos.getY() <= line.getEndY())))
    .forEach((line) -> {
        animTimer.stop();
        System.out.println("COLLISION!");
        Platform.exit();
    });
}
}

```

Explanation of the Code

Let's break the code down into sections to take a closer look, starting with the member variables as shown in Listing 11-2.

Listing 11-2. RaspiCycle Member Variables

```

private int SPEED = 1;
private double SCREENHEIGHT;
private double SCREENWIDTH;
private Canvas canvas;
private GraphicsContext gc;
private Point2D startPos;
private Point2D curPos;
private Point2D newPos;
private final List<Line> walls = new ArrayList<>();

public enum Direction {
    LEFT, RIGHT, UP, DOWN
};
private Direction curDir = Direction.UP;

private AnimationTimer animTimer;

```

The first variable in this code is an `int` named `SPEED`. We use this variable to enable the user to change the pace of the game. We start at a speed of 1; if the person driving the light cycle is feeling the “Power of the Users”, s/he can increase it! They can also decrease it, as we see later in the code.

The next two variables, `SCREENHEIGHT` and `SCREENWIDTH`, are pseudo-constants that we use to keep track of the screen’s visual boundaries.

The next two, `canvas` and `gc`, provide the visual foundation of our game. The JavaFX documentation about the `Canvas` class describes it this way: “`Canvas` is an image that can be drawn on using a set of graphics commands provided by a `GraphicsContext`.”

`Canvas` extends `Node`, providing a backdrop—similar to an artist’s canvas in the physical world—upon which you can (programmatically) write letters and numbers, draw points, lines, and geometric shapes, change colors, and much more.

Each Canvas object has an associated `GraphicsContext` that is used to issue draw calls on the canvas. In the physical world, the `GraphicsContext` corresponds to the artist's supplies: brushes and sponges, paints and more, anything used to create something visual on the canvas.

The next three variables are two-dimensional (2D) points. In this example, we use the points' x and y coordinates to track the current position (`curPos`) of the cycle, the new position (`newPos`) to which the cycle is moving, and the starting position (`startPos`) of the current light trail. As you will soon see, all three of these positions are important for managing the game rendering and player positioning.

The List of Line objects allows us to track every boundary wall and light trail that spells disaster for our light cycle if we collide with it. Line objects serve our purposes well, as we need an easy way to compare coordinates with our cycle's projected position, and a List (in this case, an `ArrayList`) provides an easy way to manage those walls.

Our game allows the user to move in one of four directions: UP, DOWN, LEFT, or RIGHT. We create an enum variable and its members to facilitate our handling of the directional guidance provided by the user and the cycle's movements, assigning it a starting direction of UP, similar to the original *TRON* Light Cycles game.

And finally, the `AnimationTimer` gives us a means to set and keep the game in motion and to stop it when a collision occurs or the user requests to end the game.

Let's skip ahead just a bit in the code. In Listings 11-3 through 11-5, we take a closer look at how easy it is to set up the canvas background for our `RaspiCycle` program. Listing 11-3 contains a bit of code from the `start()` method in which we first set the `GraphicsContext`'s fill color to a deep blue using RGB values to get just the right hue, and then use `fillRect()` with the primary screen's dimensions to paint the entire canvas with it. We then call a method to draw a criss-cross of black lines to form the background grid, and finally a method to draw some walls, establishing the game's visual boundaries.

Listing 11-3. Creating the Game Grid

```
gc.setFill(Color.rgb(2, 2, 47));
gc.fillRect(0, 0, SCREENWIDTH, SCREENHEIGHT);
drawGameGrid();
drawWalls(bounds);
```

The `drawGameGrid()` method shown in Listing 11-4 lays down the expected background of gridwork. To do so, we determine an appropriately-sized box pattern based upon a quick calculation we perform by dividing the the smaller screen dimension (width or height) by 10. We then loop horizontally and vertically through the screen real estate to place black lines into a uniform grid pattern.

Listing 11-4. The `drawGameGrid()` Method

```
private void drawGameGrid() {
    int boxSize = (int) (Math.min(SCREENWIDTH, SCREENHEIGHT) / 10);
    int i;
    gc.setStroke(Color.BLACK);
    gc.setLineWidth(5);
```

```

// Draw horizontal lines
i = boxSize;
while (i < SCREENHEIGHT) {
    gc.strokeLine(0, i, SCREENWIDTH, i);
    i += boxSize;
}

// Draw vertical lines
i = boxSize;
while (i < SCREENWIDTH) {
    gc.strokeLine(i, 0, i, SCREENHEIGHT);
    i += boxSize;
}
}

```

Note Although the screen of the Raspberry Pi (like most computers) is wider than it is tall, it is better to allow the code to accommodate other screen configurations automatically. This makes it easier to adapt and deploy applications on mobile platforms or component screens, for example, without significant—or perhaps any—changes to the presentation code.

In Listing 11-5, the `drawWalls()` method adjusts the color and width of the virtual pen and draws the outer boundaries, completing the Game Grid and readying it for play.

Listing 11-5. The `drawWalls()` Method

```

private void drawWalls(Rectangle2D bounds) {
    gc.setStroke(Color.LIGHTBLUE);
    gc.setLineWidth(2);

    // Top wall
    gc.strokeLine(bounds.getMinX(), bounds.getMinY(),
                  bounds.getMaxX(), bounds.getMinY());

    // Right wall
    gc.strokeLine(bounds.getMaxX(), bounds.getMinY(),
                  bounds.getMaxX(), bounds.getMaxY());

    // Bottom wall
    gc.strokeLine(bounds.getMinX(), bounds.getMaxY(),
                  bounds.getMaxX(), bounds.getMaxY());

    // Left wall
    gc.strokeLine(bounds.getMinX(), bounds.getMinY(),
                  bounds.getMinX(), bounds.getMaxY());
}

```

Listing 11-6 concisely demonstrates the Lambda approach to handling key presses. Rather than the anonymous inner class that would have been required prior to Java 8, we simply allow the compiler to infer the method based upon the expected class/parameter and provide the method's functionality—in this case, by defining and calling RaspiCycle's handleKeyPress() method with the KeyEvent as a parameter, as shown in Listing 11-7.

Listing 11-6. Capturing User Input

```
scene.setOnKeyPressed( (KeyEvent event) -> {
    handleKeyPress(event);
});
```

Listing 11-7. The handleKeyPress() Method

```
public void handleKeyPress(KeyEvent event) {
    boolean isNewDir = false;

    if (event.getCode() == KeyCode.LEFT &&
        curDir != Direction.LEFT) {
        curDir = Direction.LEFT;
        isNewDir = true;
    }
    else if (event.getCode() == KeyCode.RIGHT &&
              curDir != Direction.RIGHT) {
        curDir = Direction.RIGHT;
        isNewDir = true;
    }
    else if (event.getCode() == KeyCode.UP &&
              curDir != Direction.UP) {
        curDir = Direction.UP;
        isNewDir = true;
    }
    else if (event.getCode() == KeyCode.DOWN &&
              curDir != Direction.DOWN) {
        curDir = Direction.DOWN;
        isNewDir = true;
    }
    else if (event.getCode() == KeyCode.DIGIT1) {
        SPEED = 1;
    }
    else if (event.getCode() == KeyCode.DIGIT2) {
        SPEED = 2;
    }
    else if (event.getCode() == KeyCode.DIGIT3) {
        SPEED = 3;
    }
    else if (event.getCode() == KeyCode.ESCAPE) {
        animTimer.stop();
        Platform.exit();
    }
}
```

```
if (isNewDir) {
    // User sent Light Cycle in a new direction...
    // add the wall (light trail) to the list & start a new one
    walls.add(getLine(startPos.getX(), startPos.getY(),
                      curPos.getX(), curPos.getY()));
    startPos = curPos;
}
}
```

Our game requires very few input options from the user to make it playable. If the user presses the up, down, left, or right arrow keys—and if this guidance indicates a change of direction—we update the current direction indicator (`curDir`) and a change-of-direction flag (`isNewDir`) so that the game engine can respond. Likewise, if the user presses the 1, 2, or 3 keys, we assign the value input to the `SPEED` variable to alter, if necessary, the pace of the game. If the user wants to end the game (without the indignity of driving his light cycle into a wall!), s/he simply presses the Escape key; the program stops the timer loop and exits.

If there is a change of direction, this means that the current light trail ends. . . and a new one begins in a new direction. The position of the current trail is recorded in the list of “walls” to check for collisions and the `startPos` variable is updated to indicate that it’s time to “lay down light” in a new direction!

Taken together, Listings 11-8 through 11-11 demonstrate the heart of the game—the game’s engine, if you will. Listing 11-8 establishes the control loop by defining an object extending the abstract `AnimationTimer` class, overriding its abstract `handle()` method. The `handle()` method is called in every frame, providing the mechanism needed to actively loop through the game’s code until the game ends. In this example, we capture the central organization of the game engine in the `runLightCycle()` method and place a call to it within the `handle()` method, then start the timer to kick things off.

Listing 11-8. Draw and Play

```
animTimer = new AnimationTimer() {
    @Override
    public void handle(long now) {
        runLightCycle();
    }
};
animTimer.start();
```

Listing 11-9: The `runLightCycle()` Method

```
private void runLightCycle() {
    calculateDestination();
    gc.setStroke(Color.YELLOW);
    gc.setLineWidth(5);
    gc.strokeLine(curPos.getX(), curPos.getY(), newPos.getX(), newPos.getY());
    curPos = newPos;
    checkCollision();
}
```

The objective of our game is to avoid *crashing your light cycle for as long as possible*. The original *TRON* Light Cycles game had other light cycles also trying to force you to crash, but we'll leave that for a follow-on exercise, as this short program both demonstrates the basic concepts *and* makes for a challenging enough user experience. . .especially if run with SPEED set to 3.

Within the `runLightCycle()` method, we perform the following tasks:

1. Calculate the next position based upon user input and existing direction/speed.
2. Set the graphics context to draw the correct line color and width.

Note As mentioned earlier, this is the correct color per the original *TRON* movie; in the video game, the antagonists drove the yellow light cycles. However, as with most things, prior art prevails!

3. Move the cycle by drawing the light trail from the current position to the newly calculated one.
4. Store the new position as current.
5. Check for a collision. Did the cycle hit a boundary wall or light trail?

Steps 1 and 5 are accomplished by calling the `calculateDestination()` and `checkCollision()` methods, respectively. We'll examine those next.

In Listing 11-10, which shows the `calculateDestination()` method, we examine the current direction in which the cycle is going or the user wishes it to go next (`curDir`), factor in the `SPEED` setting, and calculate the next set of coordinates. We then assign the point created from these coordinates to the light cycle's new position (`newPos`) and return control to the `runLightCycle()` method.

Listing 11-10. The calculateDestination() Method

```
private void calculateDestination() {
    double newX = 0;
    double newY = 0;

    switch (curDir) {
        case UP:
            // X doesn't change, only Y
            newX = curPos.getX();
            newY = curPos.getY() - SPEED;
            break;
        case DOWN:
            // X doesn't change, only Y
            newX = curPos.getX();
            newY = curPos.getY() + SPEED;
            break;
    }
}
```

```
        case LEFT:
            // Y doesn't change, only X
            newY = curPos.getY();
            newX = curPos.getX() - SPEED;
            break;
        case RIGHT:
            // Y doesn't change, only X
            newY = curPos.getY();
            newX = curPos.getX() + SPEED;
            break;
    }
    newPos = new Point2D(newX, newY);
}
```

Listing 11-11 contains the `checkCollision()` method.

Listing 11-11. The `checkCollision()` Method

```
private void checkCollision() {
    System.out.println("Current Position: (" + curPos.getX() + ", " + curPos.getY() + ")");

    walls.stream().filter((line) -> (
        (line.get startX() <= curPos.getX() && curPos.getX() <= line.getEndX()) &&
        (line.get startY() <= curPos.getY() && curPos.getY() <= line.getEndY())))
        .forEach((line) -> {
            animTimer.stop();
            System.out.println("COLLISION!");
            Platform.exit();
        });
}
```

The first thing we do is to print the current position to standard output. This is entirely optional but is included as both homage to the original *TRON* (remember TRace ON discussed earlier?) and for a demonstration of remote execution, shown in the next section.

The heart of this method is the second statement in `checkCollision()`, and it provides a more extensive example of “*Lambdas in action*” when used with the new Java 8 Stream API (<http://download.java.net/jdk8/docs/api/java/util/stream/package-summary.html>). Let’s take it apart to get a better idea of how this statement checks for a collision and if appropriate, stops the game.

The new Stream API enables us to create a stream from a collection—in this case, our list of `walls`—and perform certain operations on it. We do so by creating a *stream pipeline*, passing the manipulated stream from one operation to the next.

We start with `filter()`. In this example, we’ve provided a Lambda expression that can be used to filter “in or out” a particular entry, and the Stream API does the work of applying the filter to each element. Only entries that match will continue through the pipeline and be evaluated further.

There are cases where additional manipulation may be required or desired, and the `map()` and `reduce()` methods (and variations) provide powerful capabilities in several contexts; please see the official JDK8 documentation for Stream (<http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>) for more information. For this program, however, all we need to do is

determine if our current position has placed us in the same coordinate space as one of our walls. The `forEach()` method will take each item that is passed to it via the stream and execute the supplied Lambda expression; and since that filter only passes a wall through the *stream pipeline* if we've crashed our light cycle into it, it stops the timer, prints the sad outcome, and shuts down the program.

Using NetBeans with the Raspberry Pi

If you're reading this, you've made it through the discussion and code and are now ready for the best part: building your Light Cycle game and running it on your Raspberry Pi! Here are the steps you'll need to take to do that:

1. Download the RaspiCycle project from the Apress media site for this book.
2. Start NetBeans 8.
3. From the NetBeans File menu, select Open Project and navigate to the RaspiCycle project directory. Select the project and click the Open Project button.
4. Once the project is loaded, right-click on it in NetBeans' Projects tab and select Properties.
5. From the Project Properties box, select Run from the Categories menu to the left. You will see a window similar to that in Figure 11-8.

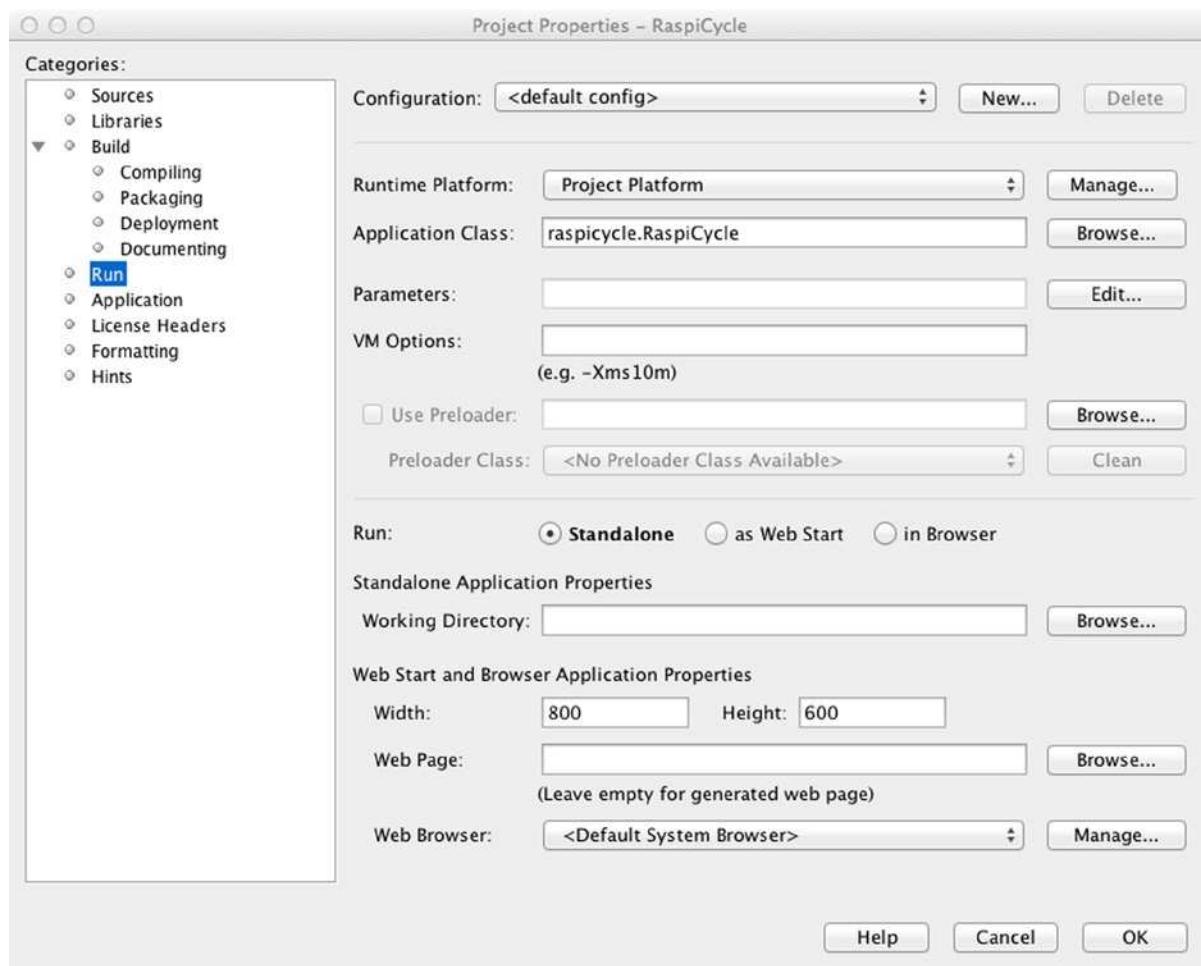


Figure 11-8. Selecting Run from the NetBeans Project Properties window

6. Click the New button to the right of the selected Configuration. Provide a name for the New Configuration (“remote config” or something similar) and click the OK button (see Figure 11-9).

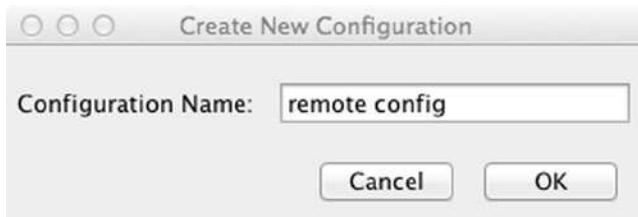


Figure 11-9. Create New Configuration

7. Now we must associate a remote JDK with our remote configuration. To do so, click the combobox labeled Runtime Platform and select the remote platform that you configured earlier (named *Raspberry Pi JDK 8*). The Project Properties should look something like Figure 11-10 at this point. Click the OK button to save and enable this configuration.

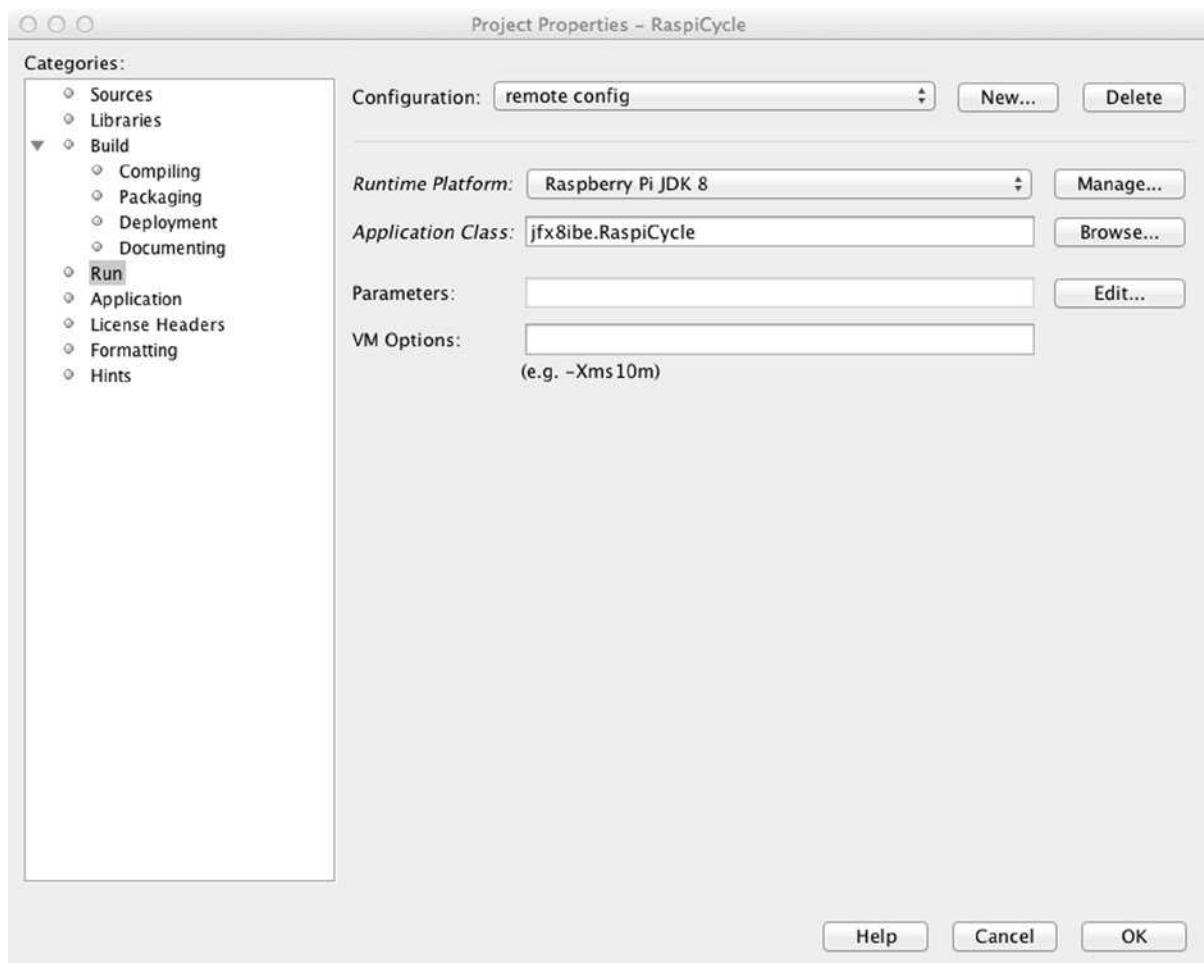


Figure 11-10. Run with Remote Configuration

8. The next and final step is to build and deploy the game! To do so, go to the Run menu and select Run Project and watch NetBeans' Output window/ tab. . . while keeping an eye on the Pi's screen. Figures 11-11/11-12 and Figures 11-13/14 show what you can expect to see in the NetBeans Output window and on the Raspberry Pi, respectively.

```
Detected JavaFX Ant API version 1.3
jfx-deployment:
jar:
Connecting to raspicam:22
cmd : mkdir -p '/home/pi/NetBeansProjects//RaspiCycle/dist'
Connecting to raspicam:22
done.
Connecting to raspicam:22
cmd : cd '/home/pi/NetBeansProjects//RaspiCycle'; '/opt/jdk1.8.0/jre/bin/java' -Dfile.encoding=UTF-8
```

Figure 11-11. Deploying to the Raspberry Pi directly from NetBeans

```
Current Position: (683.0, 730.0)
Current Position: (683.0, 729.0)
Current Position: (683.0, 728.0)
Current Position: (683.0, 727.0)
Current Position: (683.0, 726.0)
Current Position: (683.0, 725.0)
Current Position: (683.0, 724.0)
Current Position: (683.0, 723.0)
Current Position: (683.0, 722.0)
Current Position: (683.0, 721.0)
```

Figure 11-12. TRace ON!

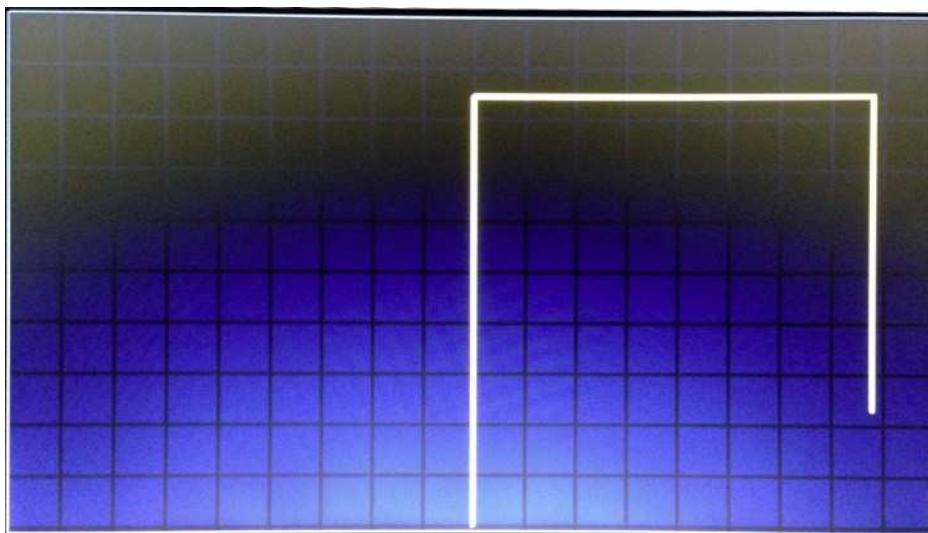


Figure 11-13. Light Cycle running!

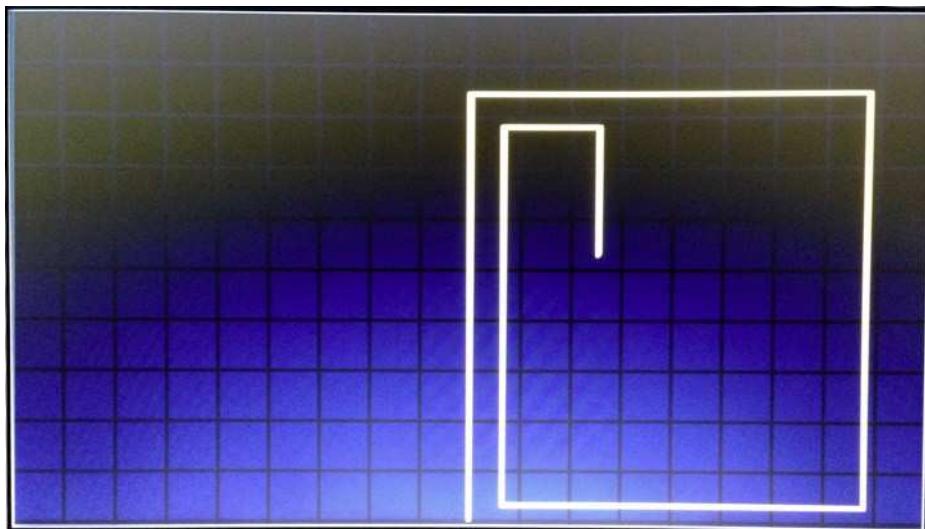


Figure 11-14. And running...

Tip Once you get to this stage, you'll want to again plug in the USB keyboard and HDMI TV/monitor for the easiest—and easily most impressive—results!

Figures 11-13 and 11-14 are photos taken of the Raspberry Pi screen in action, and the clarity of these images would certainly improve with a good screen-capture utility for the Pi! To further whet your appetite—and to demonstrate that you can easily run the game on your dev machine as well by simply changing the Configuration under Project Properties ► Run, we will close this chapter with Figure 11-15, an example of the same program running on a MacBook Pro. Developing for the Pi using JavaFX doesn't mean it can *only* run on the Pi, after all!

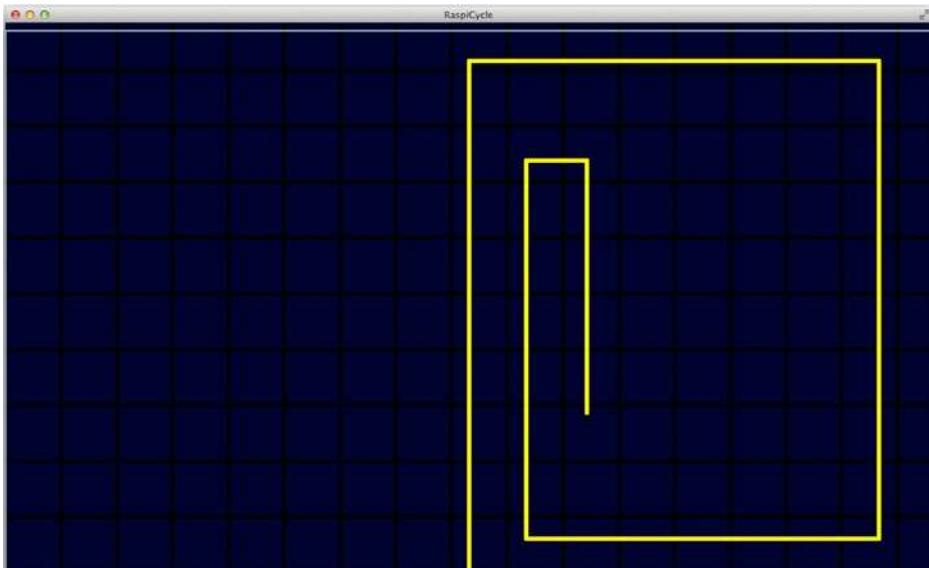


Figure 11-15. The picture may be clearer, but in reality, it looks the same. Careful with that Light Cycle!

Summary

In this chapter, we've taken a tiny computer no larger than a deck of playing cards and transformed it into a JavaFX gaming machine. Along the way, you've gotten acquainted with the Raspberry Pi; and you've learned how to create a bootable SD card and install an operating system to it, how to configure it for best performance and remote connectivity and give it a fixed network (IP) address, and how to connect to the Pi from your development machine. You've revisited how to install JDK8/JavaFX8 to both the Raspberry Pi and your dev box and learned how to install and configure NetBeans on your dev box so it can use the JDK on the Pi as a remote platform for debugging and execution.

With both your development machine and Raspberry Pi ready for action, you were introduced to the principles needed to develop a simple but compelling game for the Raspberry Pi using JavaFX and a few select tools/APIs, including Canvas, GraphicsContext, AnimationTimer, lambdas, and the new Stream API.

To end the chapter, you saw how to remotely deploy your application from NetBeans to your Raspberry Pi, all with a click. Now it's your turn. What will you do next?

JavaFX and Gestures

User interfaces are becoming increasingly less mouse-centric, in favor of multi-touch and even touchless input. Gestures are one of the ways humans can communicate with machines naturally, as the latter begin to understand human body language, thanks to complex mathematical algorithms for gesture recognition.

In this chapter you will learn about two new approaches to gesture recognition: In the first part of the chapter you will learn about developing multi-touch JavaFX applications using a touch-enabled device such as the Windows 8 Surface, while in the second part of this chapter you will discover an awesome gadget, the Leap Motion device, that will allow a touchless approach to develop enhanced JavaFX applications.

In this chapter the following examples will demonstrate JavaFX's touch events API and the Leap Motion Gestures API.

- Animating a shape along a path using touch events (Windows 8)
- Touch, rotate and zoom in JavaFX 3D (Windows 8)
- A shooting star using hand and palm gestures (Leap Motion)
- Water ripple effect using fingers and swipe gestures (Leap Motion)

Before getting into the examples, let's look at JavaFX's touch events.

Recognizing Gestures in Your Application

Touch-enabled devices are currently part of our culture. From smart phones to tablets, we find ourselves immersed in software. In this section you will learn the JavaFX `TouchEvent` API for touch-enabled devices. The examples in this section are executed on the Windows 8 surface.

If you are familiar with gestures and gesture programming in general, then you can skip to Table 12-1, which lists all the gesture events currently supported by JavaFX 8. If you are new to gesture programming, then first let's discuss the differences between a `MouseEvent` and `GestureEvent`.

Traditional MouseEvent programming is based on a simple 2D coordinate system aligned with the pixels of the screen. Typically mouse event handling is oriented toward wherever the mouse is at the moment of the event. Gestures are slightly different in that they are oriented toward touch points and their relative motion. A TouchPoint is the location on the screen or device that detects interaction through physical contact. There can be multiple touch points simultaneously, and each TouchPoint instance contains information such as X and Y coordinates, GUI component picking, and grouping of related touch points. Generally touch points are treated in a similar fashion to mouse clicks.

The code for the first example of this chapter provided by Listing 12-1, shows both mouse and touch point event handling side by side. The example explains what the gesture TouchPoint equivalents are to various mouse events such as MousePressed, MouseDragged, and MouseReleased.

Touch points are the basis of the more useful gestures such as rotations, scrolls, swipes, and zooms. These more complex gestures involve multiple touch points and movement or inertia associated with the touch points. For instance, touching and dragging your finger across a touch screen will simply register multiple TouchPoint events in a similar fashion to MouseEvent.MOUSE_DRAGGED. However, a registered TouchPoint followed by a quick swipe to the edge of the device translates to a swipe gesture. This expands the toolset of the developer by providing a multitude of additional interface options that generally don't exist within a standard mouse and keyboard GUI.

Gesture events are enabled for your application in the same manner as all event processing: via event handlers. Table 12-1 provides a listing of the available Gesture events.

Table 12-1. Gesture Events

Gesture Event	Description
ROTATION_STARTED	Two touch points recognized, with one touch point initiating a circular movement.
ROTATE	Two touch points recognized, with one or both touch points performing a circular movement.
ROTATION_FINISHED	Two touch points recognized, with both touch points halted after performing a circular movement.
SCROLL_STARTED	One or more touch points that have begun to move in a slow horizontal or vertical sliding movement.
SCROLL	One or more touch points that are moving in a slow horizontal or vertical sliding movement.
SCROLL_FINISHED	One or more touch points that have ceased moving in a slow horizontal or vertical sliding movement.
SWIPE_LEFT	One or more touch points that are moving in a quick sliding movement to the left edge of the screen.
SWIPE_DOWN	One or more touch points that are moving in a quick sliding movement to the bottom edge of the screen.
SWIPE_RIGHT	One or more touch points that are moving in a quick sliding movement to the right edge of the screen.

(continued)

Table 12-1. (continued)

Gesture Event	Description
SWIPE_UP	One or more touch points that are moving in a quick sliding movement to the top edge of the screen.
TOUCH_PRESSED	A single touch point recognized.
TOUCH_STATIONARY	A single touch point that has remained in the same relative screen location without movement.
TOUCH_MOVED	A single touch point that has moved from a previously recognized TOUCH_PRESSED or TOUCH_STATIONARY location.
TOUCH_RELEASED	A single touch point that has been lifted from the screen.
ZOOM_STARTED	Two touch points recognized, initiating a pinch or stretch motion.
ZOOM	Two touch points recognized, performing a pinch or stretch motion.
ZOOM_FINISHED	Two touch points recognized, with both touch points halted after performing a pinch or stretch motion.

Example: Animating Shapes along a Path using Touch Events

Let's say you want to allow the user of your application to trace a path within your visual application. Tracing through a visual presentation is a very natural interaction for humans. A simple example might be to record the movement of a mouse on the screen and then "play back" the movement through a smooth JavaFX animation. Listing 12-1 is a complete example of this.

Listing 12-1. Animating a Circle using Touch Events

```
package jfx8ibe;

import javafx.animation.PathTransition;
import javafx.application.Application;
import javafx.geometry.Point2D;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.*;
import javafx.scene.shape.*;
import javafx.stage.Stage;
import javafx.util.Duration;

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

```
Path onePath = new Path();
Point2D anchorPt;
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Chapter 12 Touch Events with Paths");

    Group root = new Group();
    root.getChildren().add(onePath); // add the path to record events
    Scene scene = new Scene(root, 600, 800, Color.WHITE);
    //Nice colorized gradient for the circle
    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 circle to be animated
    Circle circle = new Circle(100, 100, 20, gradient1);
    // add circle to the scene
    root.getChildren().add(circle);
    // animate circle by following the path
    PathTransition pathTransition =
        new PathTransition(Duration.millis(4000), onePath, circle);
    pathTransition.setCycleCount(1);
    pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);

    // once finished clear path
    pathTransition.setOnFinished(actionEvent -> onePath.getElements().clear());

    // starting initial path
    scene.setOnMousePressed((MouseEvent) ->
        startPath(mouseEvent.getX(), mouseEvent.getY())
    );

    // dragging creates lineTo objects added to the path
    scene.setOnMouseDragged((MouseEvent) ->
        drawPath(mouseEvent.getX(), mouseEvent.getY()));
}

// end the path when mouse released
scene.setOnMouseReleased(MouseEvent -> endPath(pathTransition));

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

```
private void startPath(double x, double y) {  
    onePath.getElements().clear();  
    anchorPt = new Point2D(x, y);      // start point in path  
    onePath.setStrokeWidth(3);  
    onePath.setStroke(Color.BLACK);  
    onePath.getElements()  
        .add(new MoveTo(anchorPt.getX(), anchorPt.getY()));  
}  
  
private void drawPath(double x, double y) {  
    onePath.getElements().add(new LineTo(x, y));  
}  
  
private void endPath(PathTransition pathTransition) {  
    onePath.setStrokeWidth(0);  
    if (onePath.getElements().size() > 1) {  
        pathTransition.stop();  
        pathTransition.playFromStart();  
    }  
}  
}  
} //End Listing 12-1
```

Running Listing 12-1 will draw and animate a Circle object based on the movement of the mouse. This will provide a view similar to Figure 12-1.

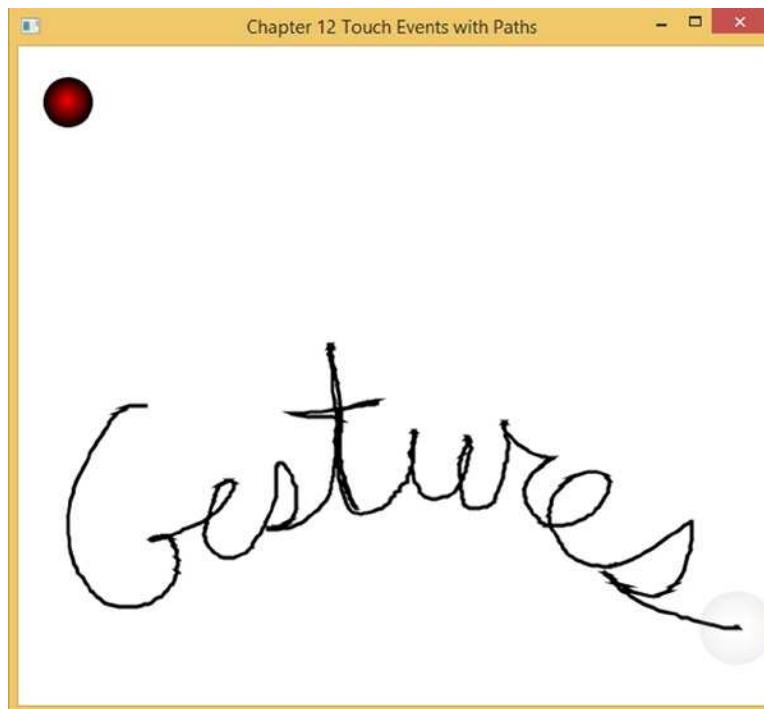


Figure 12-1. Using mouse or touch gestures to animate a shape along a path

This circle object is animated along the onePath path using a PathTransition with a 4000 millisecond transition time. The onePath point element list is defined using LineTo objects, which are built from the x and y coordinates captured from simple mouse event handlers. In this situation the capture is initiated with the OnMousePressed event, collected by the OnMouseDragged event, and completed using the OnMouseReleased event.

However, as we mentioned earlier, tracing is a very natural human interaction and is appropriately done with the hands. If you are running this on a touch-compliant screen device, you can easily use the JavaFX touch event support to enhance your tracing application. Touch events, like all Gesture events, have first-class event handling in JavaFX 8. If a GUI component supports Mouse events, then it will also support Gesture events. The best part is that Mouse and Gesture events coexist nicely within a single application. Rather than replace the current Mouse event handlers, we will match each Mouse event handler with its equivalent Touch event. The equivalent to MousePressed is TouchPressed, so add the following code to Listing 12-1. Add it in the start() method just after the existing scene.setOnMousePressed() handler.

```
scene.setOnTouchPressed((touchEvent) ->
    startPath(touchEvent.getTouchPoint().getX(),
              touchEvent.getTouchPoint().getY()));
```

This snippet demonstrates that the location information for all Gesture events is accessible through the TouchPoint object.

The equivalent to MouseDragged is TouchMoved, so add the following code to Listing 12-1 in the start() method just after the existing scene.setOnMouseDragged() handler:

```
scene.setOnTouchMoved((touchEvent) ->
    drawPath(touchEvent.getTouchPoint().getX(),
             touchEvent.getTouchPoint().getY()));
```

The equivalent to MouseReleased is TouchReleased, so add the following code to Listing 12-1. Place it in the start() method just after the existing scene.setOnMouseReleased() handler.

```
scene.setOnTouchReleased(touchEvent -> endPath(pathTransition));
```

As you can see, great care was taken to ensure that usage of Gesture event handling would be as seamless and natural as using traditional Mouse event handling.

Touch, Rotate, and Zoom in 3D

When we're manipulating a 3D scene, the most important interactions are typically rotations and zooming, which are most concerned with moving and changing the orientation of objects or the camera. Integrating Gesture support into your 3D application is a natural enhancement over the traditional mouse and keyboard. JavaFX Shape3D objects, whether primitives or custom triangle meshes, support all Gesture events such as Rotate and Zoom by default.

To demonstrate using gesture events within a JavaFX 3D scene, we will take our TriangleMesh example from Chapter 9 and extend it with the touchStationary, Rotate, and Zoom gesture events.

Our base TriangleMesh example creates four different pyramid TriangleMesh objects, each added to a MeshView container to be visible. Each MeshView container is added to a Group node, which is then added to the 3D scene graph. So let's say we want to enhance our scene by adding a Rotate gesture event to each pyramid. A user should be able to touch a pyramid and then rotate it about the z axis using a rotation gesture. If you read the previous example in this chapter you know that we need to add an event handler for the Rotate gesture, in this case OnRotate. However, you must first provide a means to notify your application which onscreen object is intended to rotate, or else we will rotate the camera perspective. If we stay within the multi-touch paradigm, we will first need to add an event handler for OnTouchStationary. The OnTouchStationary event indicates to the application that a single touch has occurred and has remained in the same relative location. This differentiates the event from OnTouchPressed and OnTouchMoved, which occur immediately when any touch occurs on the device, whether it is part of a movement gesture or not. Add a private variable:

```
private Node activeNode = null;
```

Our strategy is to select an onscreen object as the active Node to be the focus of the subsequent gestures. The OnTouchStationary event will occur before any OnRotate or OnZoom events, so the activeNode will be available to those following gesture events. We can use this event to select the active node by adding the following event handler:

```
scene.setOnTouchStationary(event -> {
    Node picked = event.getTouchPoint()
        .getPickResult()
        .getIntersectedNode();
    if (null != picked) {
        activeNode = picked;
    } else {
        activeNode = camera;
    }
    event.consume();
}); //End OnTouchStationary
```

As in our previous touch example, we access the location data via the TouchPoint object. The TouchPoint object has picking support built, in which automatically gives us access to the intersected Node, which is to say the on screen object picked. If no object is picked, that is, the touch was not over an onscreen object, then the camera is set as the activeNode. Now we can add our OnRotate event handler:

```
scene.setOnRotate(event -> {
    if(null == activeNode)
        activeNode = camera;
    activeNode.setRotationAxis(Rotate.Z_AXIS);
    activeNode.setRotate(activeNode.getRotate() + event.getAngle());
    event.consume();
}); //End OnRotate
```

Access the rotation value provided by the Rotate gesture using event.getAngle() and add it to whatever the previous rotation value was for the current Node. This will spin the object on screen. If the user is touching nothing, then the camera itself is rotated, which will cause the entire scene to appear to rotate. Figure 12-2 is an example of what your scene may look like when you apply a rotation gesture to several of the scene objects.

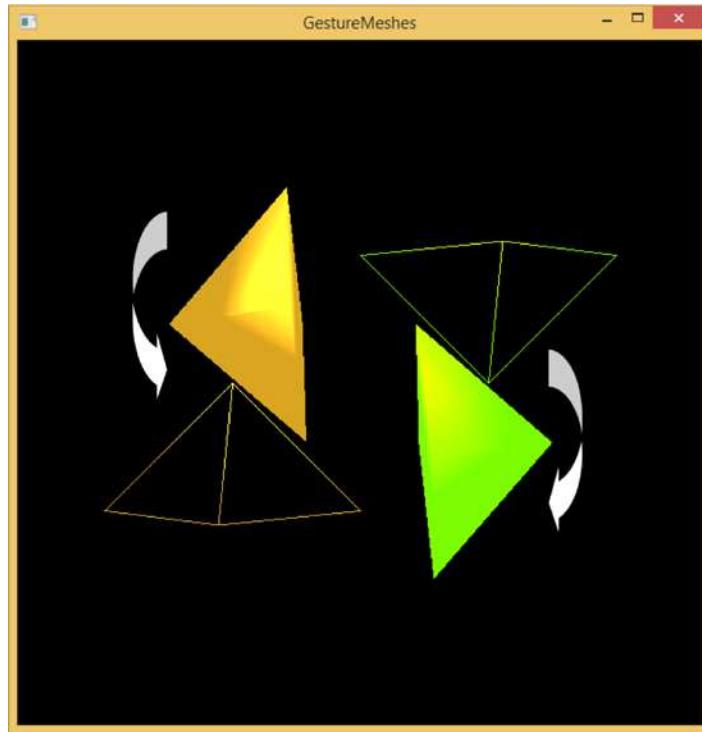


Figure 12-2. Rotating 3D pyramids using gestures

Note All objects in any scene are technically nodes. However, when picking objects, JavaFX returns a reference to the MeshView object. Any Group containers are ignored, as a Group container will present its contained Node objects to the Picking API first, and the Grabbing API will only deliver the TouchPoint event to the topmost Node (in terms of z-order) on screen.

Now let's follow the same pattern and give the user the ability to scale the activeNode. This will provide an effect of enlarging or shrinking the selected node. For this we will leverage the OnZoom event:

```
scene.setOnZoom(event -> {
    if(null == activeNode)
        activeNode = camera;
    double zoomFactor = event.getZoomFactor();
    zoomFactor *= zoomFactor < 1.0 ? -1 : 1;
    activeNode.setScaleX(activeNode.getScaleX() + zoomFactor/50);
    activeNode.setScaleY(activeNode.getScaleY() + zoomFactor/50);
    activeNode.setScaleZ(activeNode.getScaleZ() + zoomFactor/50);
    event.consume();
});
```

The Zoom event provides a double ZoomFactor, which is a positive value from 0.0 and up. Values returned between 0.0 and 1.0 indicate a “pinch” zoom gesture. Values returned higher than 1.0 indicate a “spread” zoom gesture. Here we can use the “pinch” values to negatively scale our object while the “spread” values can positively scale our object. Figure 12-3 is an example of what your scene may look like when you apply a zoom gesture to several of the scene objects.

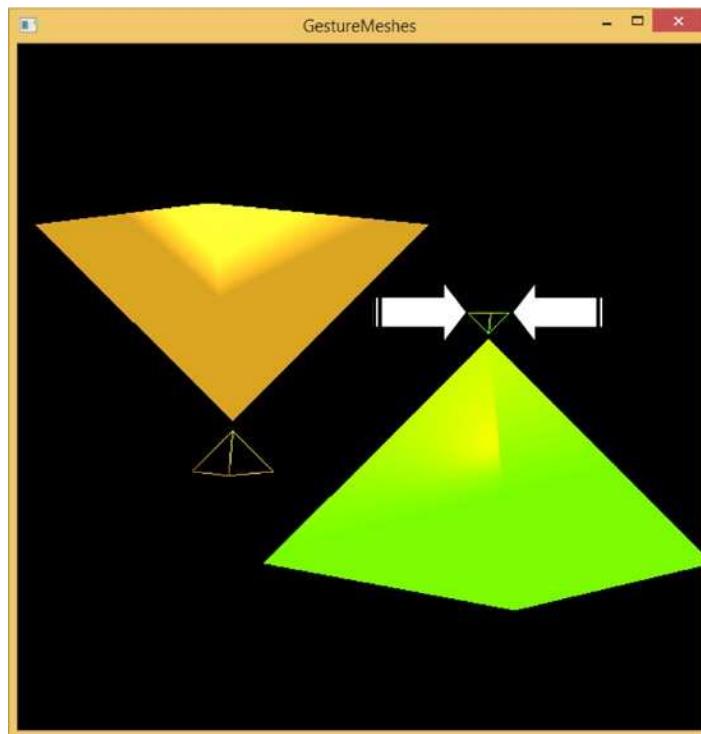


Figure 12-3. Zooming 3D pyramids using gestures

These three event handlers `OnTouchStationary`, `OnRotate` and `OnZoom` have enabled multi-touch Gesture support in your 3D scene. The effects stack upon each other, as users can rotate and zoom simultaneously using a combination of “pinch” and “spin” movements with their fingers. Not only is this type of interaction quite natural to a human but is typically easier than with a mouse and keyboard solution.

Note Here we are “scaling” our zoom with a fixed integer of 50. This is needed to compensate for the difference between onscreen coordinates and scaling coordinates. It is common practice when dealing with 3D scenes and can be thought of as a sensitivity variable.

The Leap Motion Controller

The Leap Motion Controller (Figure 12-4), also known as the “Leap”, is a small device, 80x30x12 mm, that is simply connected to your computer through the USB port. It doesn’t require external power.



Figure 12-4. *Leap Motion device and its box*

The Leap software works in Windows, Mac OS X and Linux, and with an initial price of \$79.99, was launched in July 2013 with an initial batch of six hundred thousand units sold around the world.

The company has some introductory videos here: <https://www.leapmotion.com/>. After you watch a few of them you will be surprised by the apparently magic performance of the device.

Basically, the Leap will scan your hands, at a very fast rate, and translate your movements and gestures to the computer in a very precise way, so any Leap-enabled application can be interfaced directly, without the need of using a mouse, keyboard, or any other physical device.

As a new era of interaction with machines is beginning, many challenges are appearing at the same time. As users, we’ll need to learn how to interact in 3D environments, performing a whole new repertoire of gestures, while as developers, we will face the challenge of providing the software capable of translating accurately those gestures, so new applications will respond as we expect.

Let’s start by describing the hardware and software provided by Leap Motion, and then we will talk about the API for developing JavaFX programs with the device, and present two sample applications to show what is possible to achieve once you have a Leap device in your hands.

How It Works

Basically, the main hardware of the device consists of three infrared LEDs combined with two monochromatic infrared (IR) cameras. While the LEDs generate a 3D pattern of dots of IR light, the cameras scan the reflected data at nearly 300 fps. Everything within a radius of 50 cm will be scanned and processed. With a resolution of 0.01 mm, data will be analyzed in the host computer by the Leap Motion software, using a proprietary motion detection algorithm.

Depending on your CPU capabilities and the data analyzed, the range of processing latency goes from 2 ms to 33 ms.

The best way to find out about the Leap *magic* is just plugging it on the USB port and installing the software from <https://www.leapmotion.com/setup>. A small Leap icon should appear on the taskbar notification area (Windows) or menu bar (Mac), and it should be green, like the lateral LED on the device (which should be facing you). Click on it and select the Visualizer for an incredible visualization of your fingers and hands.

A less fancy application, but more useful for understanding how it works, can be seen by choosing Settings > Control Panel > Troubleshooting > Diagnostic Visualizer. Once you see a blue grid over a black background, move your hands and click H to see the menu, N to visualize the hand, T the fingers, O to draw the gestures, or L to see the latency, and you'll see displayed something like Figure 12-5.



Figure 12-5. Leap Motion Diagnostic Visualizer application

As you can see, the device detects hands, fingers and sharp tools within a 60 cm range and a 120-150 view angle. For these objects, 3D vectors provide position, direction and velocity, all referred to a right-handed Cartesian coordinate system, whose origin is at the center of the device, as shown in Figure 12-6. Every time the device scans and processes the available data, a Frame object is generated, containing lists with the tracked data in that instant (hands, fingers, tools) as well as a set of basic motion gestures found in the analysis of the last frames (swipe, tap, circles).

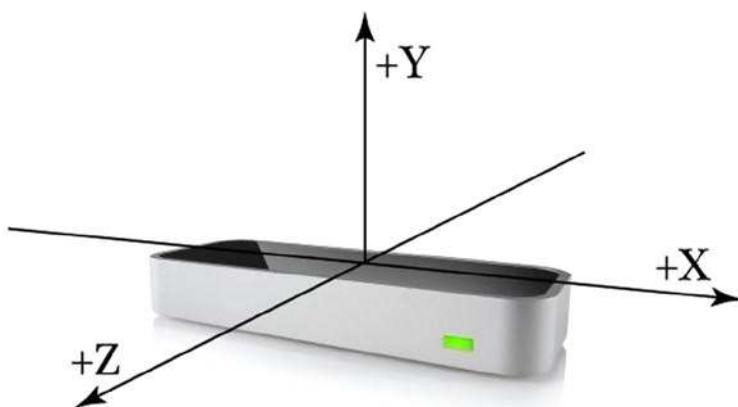


Figure 12-6. Coordinate system centered in the device

As you may have noticed, the y -axis positive direction is the opposite of the downward orientation in most computer graphics systems, including JavaFX. But the fact that the data is referred to the device position, and not to the screen as we are accustomed to with mouse and touch events, changes dramatically the way we need to think. Fortunately, the API provides several useful methods to find where our hands and fingers are pointing at any time.

More complex tasks require an understanding of spatial geometry, including knowledge of terms like vectors, vector-cross-product and matrix operations.

Getting Started with the Leap SDK

In order to integrate the Leap device events and interact with software applications, Leap Motion provides an SDK for many programming languages, including Java, in their developer section: <https://developer.leapmotion.com/downloads>. Download the SDK for your operating system, and you'll get the necessary libraries to get started. These consist of basically the LeapJava.jar and a bunch of native libraries. The SDK also includes a HelloWorld.java example.

One easy way to integrate these files on your system is by adding the jar to `<JAVA_HOME>/jre/lib/ext` on Linux or Windows (or `/Library/Java/Extensions` on Mac). Then copy the native libraries (`LeapJava.dll`, `Leap.dll` and `Leapd.dll` for Windows, `libLeapJava.dylib` and `libLeap.dylib` for Mac, `libLeapJava.so` and `libLeap.so` for Linux) to the `<JAVA_HOME>/bin` folder.

Or you can just add the jar to every project as a dependency, and load the native libraries as a VM argument: `-Djava.library.path=<native library path>`.

Before going any further, please look at the documentation, at:

<https://developer.leapmotion.com/documentation/java/index.html>, and try for yourself the `HelloWorld.java` example. You'll notice the following:

- We need a `Controller` object, allowing the connection between the Leap device and the application.
- We need a `Listener` subclass to handle events from the controller.
- Gesture tracking is enabled in the `onConnect()` method.
- The main method in this class is `onFrame()`, a callback method dispatched when a new `Frame` with motion tracking data is available. Here we can get lists of hands, fingers or tools, and several vectors with their position, orientation and velocity.
- If gestures are enabled, we'll also get a list of the gestures found, based on the analysis of the last frames. Also, you will know the status of the gesture: whether it has just begun, is progressing, or has ended.

Creating a Shooting Star

One thing you may have discovered throughout this book is the care we must put in mixing the so-called JavaFX thread with other non-JavaFX threads. At this point, it will be evident for you that the `Leap Listener` subclass is a non-JavaFX thread, handling events at a very high rate.

In order to bring these events to a JavaFX thread, we will use `ObjectProperty<T>` objects in the `Listener` subclass to store the desired values on every frame, and then we'll implement their `ObservableValue<T>` methods to expose the values. In the JavaFX class, we'll implement a `ChangeListener<T>` to listen to any change, and by using `Platform.runLater()` we will put the task of rendering these changes on the JavaFX scene graph on a JavaFX thread.

Listing 12-2 shows a basic Leap Listener subclass, with an observable object (`point`) to track where a hand is pointing on the screen. If the `hands()` list is not empty (there is one or more) and we have a valid screen, with the first hand and the first screen, we get a vector from the `intersect()` method, with the intersection between the screen and a ray projecting from the specified palm position along the specified hand direction. Coordinates are scaled to screen resolution, in pixels.

Listing 12-2. Simple Listener Subclass

```
package jfx8ibe;

import com.leapmotion.leap.*;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.value.ObservableValue;
import javafx.geometry.Point2D;
```

```
/*
 * Chapter 12. JavaFX and Leap Motion
 * @author jperedra
 */
public class SimpleLeapListener extends Listener {

    private final ObjectProperty<Point2D> point=new SimpleObjectProperty<>();
    public ObservableValue<Point2D> pointProperty(){ return point; }

    @Override
    public void onFrame(Controller controller) {
        Frame frame = controller.frame();
        if (!frame.hands().isEmpty()) {
            Screen screen = controller.locatedScreens().get(0);
            if (screen != null && screen.isValid()){

                Hand hand = frame.hands().get(0);
                if(hand.isValid()){
                    Vector intersect = screen.intersect(hand.palmPosition(),
                        hand.direction(), true);
                    point.setValue(new Point2D(
                        screen.widthPixels()*Math.min(1d,Math.max(0d,intersect.getX())),
                        screen.heightPixels()*Math.min(1d,Math.max(0d,(1d-intersect.getY())))));
                }
            }
        }
    }
}
```

In Listing 12-3 we have the JavaFX class, which uses an instance of our SimpleLeapListener shown in Listing 12-2. We use a SVGPath node, with a star-like path, to represent a shooting star, helped by the radial gradient. We listen to changes to the location of the screen where our hand is pointing, moving the star there, and also rotating it so the black vertex is always pointing to the direction of movement, as if it was flying.

Listing 12-3. Source Code of the JavaFX Class

```
package jfx8ibe;

import com.leapmotion.leap.Controller;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.geometry.Point2D;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.AnchorPane;
import javafx.scene.shape.SVGPath;
import javafx.stage.Stage;

/*
 * Chapter 12. JavaFX and Leap Motion
 * @author jperedra
 */
```

```

public class StarLeapFX extends Application {
    private final SimpleLeapListener listener = new SimpleLeapListener();
    private final Controller leapController = new Controller();

    @Override
    public void start(Stage primaryStage) {

        leapController.addListener(listener);
        SVGPath sv=new SVGPath();
        sv.setContent("M 0,44.9 H 123.5 L 23.6,117.5 61.8,0 100,117.5 z");
        sv.setEffect(new DropShadow());
        sv.setStyle("-fx-fill:radial-gradient(center 50% 20%, radius 90%, "
                + "black 0%, black 30%, red 70%, yellow 85%, red 93%, yellow 100%);");
        sv.setTranslateY(302.5);
        final AnchorPane root = new AnchorPane();
        root.setStyle("-fx-background-color:linear-gradient(from 0% 0% to 0% 100%, "
                + "#99ffff3 0%, #00ccb4 70%, #10cc00 71%, #0c9900 100%);");
        root.getChildren().add(sv);
        Scene scene = new Scene(root, 800, 600);

        listener.pointProperty().addListener((ov, t, t1) -> {
            Platform.runLater(() -> {
                Point2D d=root.sceneToLocal(t1.getX()-scene.getX()-scene.getWindow().getX(),
                        t1.getY()-scene.getY()-scene.getWindow().getY());
                double dx=d.getX(), dy=d.getY();
                if(dx>=0d && dx<=root.getWidth()-sv.getBoundsInParent().getWidth() &&
                    dy>=0d && dy<=root.getHeight()-sv.getBoundsInParent().getHeight()){
                    double ix=(dx-sv.getTranslateX());
                    double iy=(dy-sv.getTranslateY());
                    if(ix*ix+iy*iy>50){
                        sv.setRotate(Math.toDegrees(Math.PI/2+Math.atan2(iy, ix)));
                    }
                    sv.setTranslateX(dx);
                    sv.setTranslateY(dy);
                }
            });
        });
        primaryStage.setTitle("My flying Shooting Star!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    @Override
    public void stop(){
        leapController.removeListener(listener);
    }

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

```

Note the change of coordinates: on every frame we detect the screen coordinates, but our star is a node on the scene, so we have to find its coordinates before any translation.

Running the Example

Now create a JavaFX project with these two classes, add the Leap dependencies properly, and build it. Check that you have connected the Leap Motion device (green light status) and run it. Move your hand, and the “shooting star” should follow the path you describe, rotating accordingly to point to the direction of movement. Figure 12-7 shows an instant of this flight.

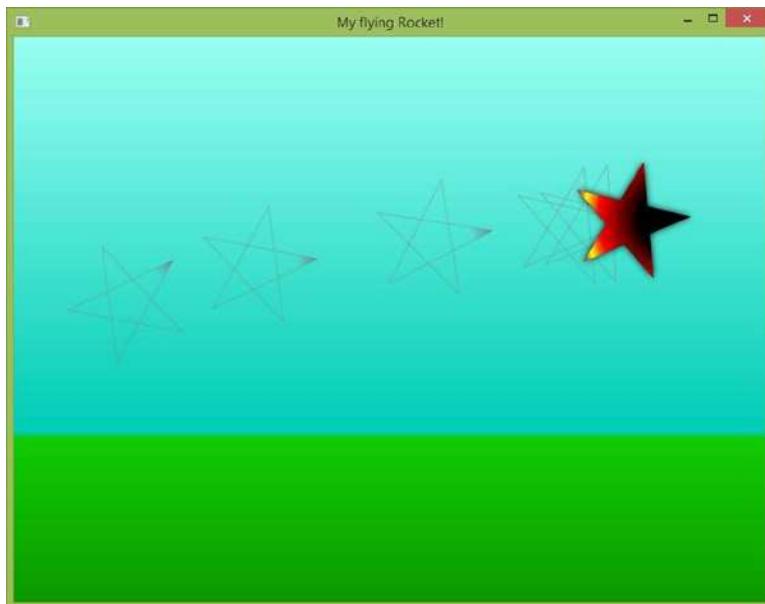


Figure 12-7. Running the shooting star example

Without going into much detail, just take a look at the way both threads work. From Listing 12-2, point is set with the exact position where the hand is pointing at the screen on every frame. For that, the screen.intersect() method provides a way to return a vector with these coordinates, based on the hand position and direction, and the proper scaling to the screen size in pixels:

```
private final ObjectProperty<Point2D> point=new SimpleObjectProperty<>();
public ObservableValue<Point2D> pointProperty(){ return point; }

@Override
public void onFrame(Controller controller) {
...
    Vector intersect = screen.intersect(hand.palmPosition(),
                                         hand.direction(), true);
    point.setValue(new Point2D(
        screen.widthPixels()*Math.min(1d,Math.max(0d,intersect.getX())),
        screen.heightPixels()*Math.min(1d,Math.max(0d,(1d-intersect.getY())))
    ));
...
}
```

On the other side, as we can see in Listing 12-3, we listen to any change in point, so by converting screen coordinates first to scene coordinates and then to local coordinates of the node, we can translate it properly, on the JavaFX thread.

```
listener.pointProperty().addListener((ov, t, t1) -> {
    Platform.runLater(() -> {
        Point2D d=root.sceneToLocal(t1.getX()-scene.getX()-scene.getWindow().getX(),
                                     t1.getY()-scene.getY()-scene.getWindow().getY());
        double dx=d.getX(), dy=d.getY();
        ...
        sv.setTranslateX(dx);
        sv.setTranslateY(dy);
    });
});
```

Note The transparent stars have been added to the image in Figure 12-7 to show the described trajectory of the shooting star, but they are not generated by the ShootingStar example.

Creating a Water Ripple Effect

For the next example, we'll add the ability to listen to the tip position of any number of fingers. In Listing 12-4 we have the source code of one single file but two classes, the JavaFX class WaterLeapFX and the Leap Listener subclass SimpleLeapListener, defined here as a private inner class of WaterLeapFX.

Listing 12-4. Source Code of the WaterLeapFX Application Class

```
package jfx8ibe;

import com.leapmotion.leap.*;
import java.awt.AWTException;
import java.awt.GraphicsEnvironment;
import java.awt.Robot;
import java.awt.image.BufferedImage;
import javafx.animation.AnimationTimer;
import javafx.application.*;
import javafx.beans.property.BooleanProperty;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.embed.swing.SwingFXUtils;
```

```
import javafx.geometry.Point2D;
import javafx.geometry.Point3D;
import javafx.scene.Scene;
import javafx.scene.image.*;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.stage.StageStyle;

/**
 * Chapter 12. JavaFX and Leap Motion
 * @author jpered
 */
public class WaterLeapFX extends Application {

    private int original[], water[];
    private short waterMap[];
    private int width, height, halfWidth, halfHeight, size;
    private int oldInd, newInd, mapInd;

    private AnimationTimer timer;
    private long lastTimerCall;

    private final SimpleLeapListener listener = new SimpleLeapListener();
    private final Controller leapController = new Controller();

    @Override
    public void start(Stage primaryStage) {
        Image img=screenSnapshot();
        final ImageView imageView = new ImageView(img);
        width = (int)imageView.getImage().getWidth();
        height = (int)imageView.getImage().getHeight();
        halfWidth=width>>1;
        halfHeight=height>>1;
        size = width * (height+2)*2;
        waterMap = new short[size];
        water = new int[width*height];
        original = new int[width*height];
        oldInd = width;
        newInd = width * (height+3);
        PixelReader pixelReader = imageView.getImage().getPixelReader();
        pixelReader.getPixels(0,0,width,height,
            WritablePixelFormat.getIntArgbInstance(),original, 0,width);

        StackPane root = new StackPane();
        root.getChildren().add(imageView);
        Scene scene = new Scene(root, width, height);
        scene.addEventHandler(KeyEvent.KEY_PRESSED, t1 -> {
            if(t1.getCode() == KeyCode.ESCAPE){ Platform.exit(); }
        });
    }
}
```

```

lastTimerCall = System.nanoTime();
timer = new AnimationTimer() {
    @Override public void handle(long now) {
        if (now > lastTimerCall + 30_000_000L) {
            imageView.setImage(applyEffect());
            lastTimerCall = now;
        }
    }
};

leapController.addListener(listener);
listener.isSwipe().addListener((ov, b, b1) -> {
    if(b1){ Platform.runLater(() -> waterMap = new short[size] ); }
});
listener.isDrop().addListener((ov, b, b1) -> {
    if(b1){
        Platform.runLater(() -> {
            for(int i=0; i<listener.points.size(); i++){
                Point3D t1=listener.points.get(i);
                Point2D d=root.sceneToLocal(
                    t1.getX()-scene.getX()-scene.getWindow().getX(),
                    t1.getY()-scene.getY()-scene.getWindow().getY());
                double dx=d.getX(), dy=d.getY(), dz = Math.abs(t1.getZ());
                int rad=(dz<50)?5:(dz<150)?4:3;
                if(dx>=0d && dx<=root.getWidth() && dy>=0d && dy<=root.getHeight()){
                    waterDrop((int)dx,(int)dy,rad);
                }
            }
        });
    }
});

primaryStage.initStyle(StageStyle.UNDECORATED);
primaryStage.setScene(scene);
primaryStage.show();
timer.start();
}

@Override
public void stop(){
    timer.stop();
    leapController.removeListener(listener);
}

public void waterDrop(int dx, int dy, int rad) {
    for (int j=dy-rad; j<dy+rad; j++) {
        for (int k=dx-rad; k<dx+rad; k++) {
            if (j>=0 && j<height && k>=0 && k<width) {
                waterMap[oldInd+(j*width)+k] += 128;
            }
        }
    }
}
}

```

```

private Image applyEffect() {
    int a,b, i=oldInd;
    oldInd=newInd;
    newInd=i;
    i=0;
    mapInd=oldInd;
    for (int y=0;y<height;y++) {
        for (int x=0;x<width;x++) {
            short data = (short)((waterMap[mapInd-width]+waterMap[mapInd+width] +
                waterMap[mapInd-1]+waterMap[mapInd+1])>>1);
            data -= waterMap[newInd+i];
            data >>= 4;
            waterMap[newInd+i]=data;
            data = (short)(1024-data);
            a=((x-halfWidth)*data/1024)+halfWidth;
            if (a>=width){ a=width-1; }
            if (a<0){ a=0; }
            b=((y-halfHeight)*data/1024)+halfHeight;
            if (b>=height){ b=height-1; }
            if (b<0){ b=0; }
            water[i++]=original[a+(b*width)];
            mapInd++;
        }
    }
    WritableImage raster = new WritableImage(width, height);
    PixelWriter pixelWriter = raster.getPixelWriter();
    pixelWriter.setPixels(0,0,width,height,
        PixelFormat.getIntArgbInstance(),water, 0,width);
    return raster;
}

private Image screenSnapshot() {
    GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
    try {
        Robot robot = new Robot(env.getScreenDevices()[0]);
        final BufferedImage bi = robot.createScreenCapture(env.getMaximumWindowBounds());
        return SwingFXUtils.toFXImage(bi, null);
    } catch (AWTException e) {}
    return null;
}

// We define SimpleLeapListener class, the Leap Listener subclass,
// as a private inner class of WaterLeapFX
private class SimpleLeapListener extends Listener {

    private final BooleanProperty drop=new SimpleBooleanProperty(false);
    protected final ObservableList<Point3D> points= FXCollections.observableArrayList();
    private final BooleanProperty swipe=new SimpleBooleanProperty(false);
}

```

```

@Override
public void onConnect(Controller controller){
    controller.enableGesture(Gesture.Type.TYPE_SWIPE);
}
@Override
public void onFrame(Controller controller) {
    Frame frame = controller.frame();
    Screen screen = controller.locatedScreens().get(0);
    if (screen != null && screen.isValid()){
        drop.set(false);
        points.clear();
        for(Finger finger : frame.fingers()){
            if(finger.isValid()){
                Vector inter = screen.intersect(finger.tipPosition(),
                                                finger.direction(), true);
                Point3D p = new Point3D(
                    screen.widthPixels()*Math.min(1d,Math.max(0d,inter.getX())),
                    screen.heightPixels()*Math.min(1d,Math.max(0d,(1d-inter.getY()))),
                    finger.tipPosition().getZ());
                points.add(p);
            }
        }
        drop.set(!points.isEmpty());
        swipe.set(false);
        for (Gesture gesture : frame.gestures()) {
            if(gesture.type()==Gesture.Type.TYPE_SWIPE &&
               gesture.state().equals(Gesture.State.STATE_STOP)){
                swipe.set(true);
                break;
            }
        }
    }
}

public ObservableValue<Boolean> isDrop() { return drop; }
public ObservableValue<Boolean> isSwipe() { return swipe; }
}

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

```

How It Works

Starting from the private Listener subclass, instead of an `ObjectProperty<Point2D>` from Listing 12-2, now we have an `ObservableList<Point3D>` collection of 3D points (shown in the following code snippet), to hold all the 2D points coordinates on the screen coming from the fingers detected, plus their z-distance to the device as a third coordinate. Notice there's also a `BooleanProperty`, set to true after the list has been completed.

```
private final BooleanProperty drop=new SimpleBooleanProperty(false);
protected final ObservableList<Point3D> points= FXCollections.observableArrayList();
```

Another `BooleanProperty` will be used to listen to swipe gestures. For that, we have to enable a listener for this kind of gesture in the `onConnect` method.

```
private final BooleanProperty swipe=new SimpleBooleanProperty(false);

@Override
public void onConnect(Controller controller){
    controller.enableGesture(Gesture.Type.TYPE_SWIPE);
}
```

Next, in the `onFrame` method, we track the fingers actually pointing at the screen, filling the list with the coordinates of the intersection point and the z-distance to the device, and updating the `BooleanProperty` at the end. This will serve to trigger a single event, instead of listening to every change event generated on the list for every finger added.

```
@Override
public void onFrame(Controller controller) {
...
    drop.set(false);
    points.clear();
    for(Finger finger : frame.fingers()){
        if(finger.isValid()){
            Vector inter = screen.intersect(finger.tipPosition(),
                                            finger.direction(), true);
            Point3D p = new Point3D(
                screen.widthPixels()*Math.min(1d,Math.max(0d,inter.getX())),
                screen.heightPixels()*Math.min(1d,Math.max(0d,(1d-inter.getY()))),
                finger.tipPosition().getZ());
            points.add(p);
        }
    }
    drop.set(!points.isEmpty());
```

Additionally, we listen for finished swipe gestures:

```
swipe.set(false);
for (Gesture gesture : frame.gestures()) {
    if(gesture.type()==Gesture.Type.TYPE_SWIPE &&
       gesture.state().equals(Gesture.State.STATE_STOP)){
```

```
        swipe.set(true);
        break;
    }
}
}
```

At the beginning of the JavaFX class, we use the old `java.awt.Robot` API to take a snapshot of the user desktop (see the `screenSnapshot()` method in Listing 12-4), before displaying the stage over it. With the image generated, we call `PixelReader` to get an array of colors for every pixel. We add the image to an `ImageViewer`. We define the scene with the dimensions of the image, and create an undecorated stage, so when the application is shown, the user sees an exact copy of the desktop. In order to allow him or her to quit the application with the Esc key, we enable a listener to key events:

```
Image img=screenSnapshot();
final ImageView imageView = new ImageView(img);
width = (int)imageView.getImage().getWidth();
height = (int)imageView.getImage().getHeight();
...
PixelReader pixelReader = imageView.getImage().getPixelReader();
pixelReader.getPixels(0,0,width,height,
        WritablePixelFormat.getIntArgbInstance(),original, 0,width);
...
StackPane root = new StackPane();
root.getChildren().add(imageView);
Scene scene = new Scene(root, width, height);
scene.addEventHandler(KeyEvent.KEY_PRESSED, t1 -> {
    if(t1.getCode() == KeyCode.ESCAPE){ Platform.exit(); }
});
```

Also, we define an `AnimationTimer` to call the `applyEffect` method every 30 ms. Note that the now timestamp is in nanoseconds. The modified new image, returned from `applyEffect()`, will replace the older one in the `ImageViewer`.

```
lastTimerCall = System.nanoTime();
timer = new AnimationTimer() {
    @Override public void handle(long now) {
        if (now > lastTimerCall + 30_000_000) {
            imageView.setImageResource(applyEffect());
            lastTimerCall = now;
        }
    }
};
```

Now we add the Listener subclass to the controller, so we can start listening to the Leap Motion device. We'll track two kinds of events by means of changes in BooleanProperty objects. First we add a listener to track the Swipe event, which will clear the water effect (resetting waterMap) from the screen.

```
leapController.addListener(listener);
listener.isSwipe().addListener((ov, b, b1) -> {
    if(b1){ Platform.runLater(() -> waterMap = new short[size] ); }
});
```

Second, we add a listener to the finger positions on the screen in order to generate the water effect over the desktop image. For that we listen only to the BooleanProperty object changes, and when the value is true (meaning the list of fingers is completed), we perform the waterDrop() call. As in the previous example, we need to convert the screen coordinates to scene coordinates. We'll use the z-coordinate as a magnifying factor for the water drop effect.

```
listener.isDrop().addListener((ov, b, b1) -> {
    if(b1){
        Platform.runLater(() -> {
            for(int i=0; i<listener.points.size(); i++){
                Point3D t1=listener.points.get(i);
                Point2D d=root.sceneToLocal(
                    t1.getX()-scene.getX()-scene.getWindow().getX(),
                    t1.getY()-scene.getY()-scene.getWindow().getY());
                double dx=d.getX(), dy=d.getY(), dz = Math.abs(t1.getZ());
                int rad=(dz<50)?5:((dz<150)?4:3);
                if(dx>=0d && dx<=root.getWidth() && dy>=0d && dy<=root.getHeight()){
                    waterDrop((int)dx,(int)dy,rad);
                }
            }
        });
    }
});
```

Note Observe that we use a for loop to iterate through the list. The reason for that is to avoid concurrent modification exceptions that could occur using functional operations. The list may be changed by the Leap while it is still being processed on the scene graph.

Finally, we set the style of the stage, show it and start the animation:

```
primaryStage.initStyle(StageStyle.UNDECORATED);
primaryStage.setScene(scene);
primaryStage.show();

timer.start();
}
```

The water-drop effect consists of a distortion added to the pixels in the surroundings of every point on the list of fingers. The distorted area will be wider if the finger is closer to the Leap device:

```
public void waterDrop(int dx, int dy, int rad) {
    for (int j=dy-rad; j<dy+rad; j++) {
        for (int k=dx-rad; k<dx+rad; k++) {
            if (j>=0 && j<height && k>=0 && k<width) {
                waterMap[oldInd+(j*width)+k] += 128;
            }
        }
    }
}
```

The water effect is achieved by creating a second image (`water`) with `PixelWriter` starting from the original image and including the water drops (`waterMap`) in terms of distorted pixels. A damping effect is included, so the drops seem to vanish after a certain time. Credit goes to Neil Wallis who implemented this effect here: <http://www.neilwallis.com/projects/java/water/index.php>.

```
private Image applyEffect() {
...
    for (int y=0;y<height;y++) {
        for (int x=0;x<width;x++) {
            short data = (short)((waterMap[mapInd-width]+waterMap[mapInd+width]+
                waterMap[mapInd-1]+waterMap[mapInd+1])>>1);
            data -= waterMap[newInd+i];
            data -= data>>4;
            waterMap[newInd+i]=data;
            data = (short)(1024-data);
            a=((x-halfWidth)*data/1024)+halfWidth;
            b=((y-halfHeight)*data/1024)+halfHeight;
...
            water[i++]=original[a+(b*width)];
            mapInd++;
        }
    }
}
```

When the array `water`, containing the pixels of the distorted image, is completely generated, we use the `PixelWriter` class to transform it into an image (see the `applyEffect()` method in Listing 12-4). This will replace the former image in the `ImageView`, as just described:

```
WritableImage raster = new WritableImage(width, height);
PixelWriter pixelWriter = raster.getPixelWriter();
pixelWriter.setPixels(0,0,width,height,
    PixelFormat.getIntArgbInstance(),water, 0,width);
return raster;
}
```

Running the Example

Copy the listing to a new JavaFX project, add the Leap dependencies, build it and run it with the Leap connected. Play with your hands, and enjoy the cool effect over your screen like the one shown in Figure 12-8.

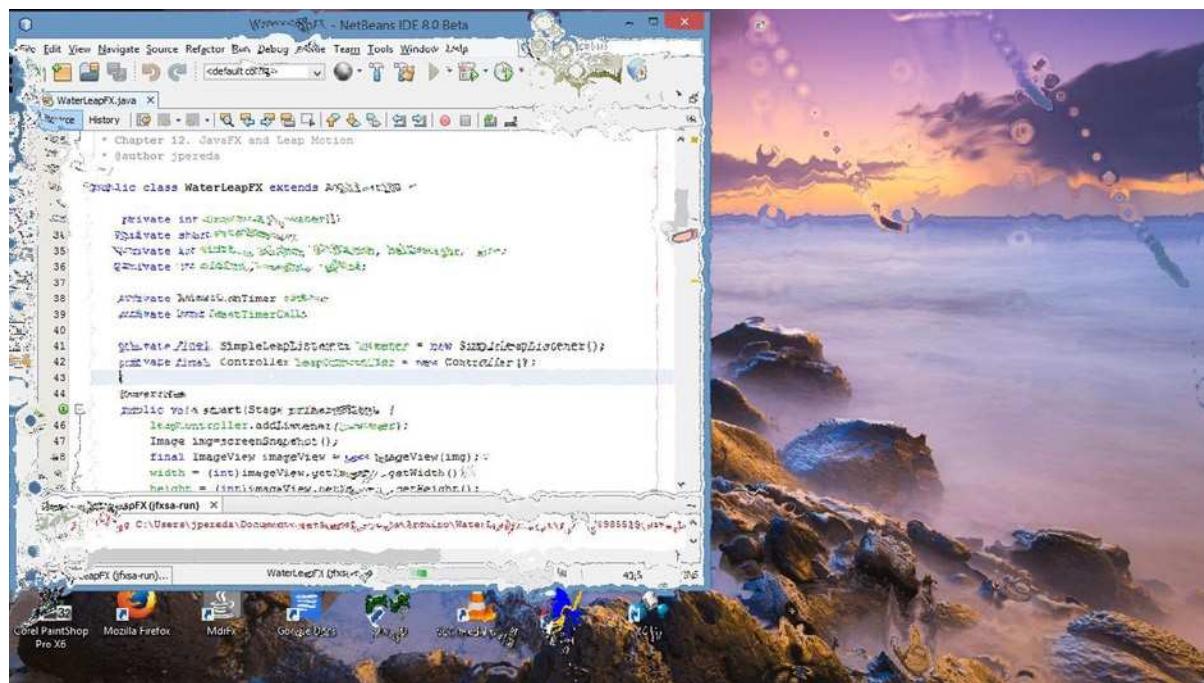


Figure 12-8. Running the water ripple effect example

Try swiping your hand from time to time to clear the screen and start all over again. Finally, press Esc to quit the application.

More Examples

For more examples of using JavaFX with the Leap Motion device, please refer to online resources such as <http://www.parleys.com/share.html#play/525467d6e4b0a43ac12124ad> or <http://jperedadnr.blogspot.com.es/2013/06/leap-motion-controller-and-javafx-new.html>. For interaction with other programming languages, please go to <https://developer.leapmotion.com/libraries>.

Summary

In the first part of this chapter you were introduced to various Gesture events and the subtle differences between them. The events you learned included TouchPressed, TouchReleased, TouchStationary, Rotate and Zoom. You learned how to add Gesture event handling to your JavaFX application whether it was a traditional 2D or even a 3D scene.

In the second part of this chapter you've discovered the impressive Leap Motion device, and the really nice combination effects that result from using it to enhance JavaFX applications. You began by learning about the device and how it works; next, you saw briefly its SDK for Java, along with two sample applications where you learned about listening and processing the data from the Leap device in one thread, while triggering events in the JavaFX thread to deal with them in the scene graph.

References

Does your JavaFX journey stop here? Of course not! In this chapter you will find many useful links and references that will help you gain further knowledge about all things JavaFX. Toward the end of this chapter, please be sure to check out the many frameworks, libraries, and projects that use JavaFX in production today.

Java 8 SDK

As a reminder, JavaFX 8 is built into Java 8 SDK. This means that you only need to download the Java 8 SDK. The Java 8 software development kit and related information can be downloaded from the following locations.

- Java 8 at Oracle Technology Network:
<http://oracle.com/java8>
- Where to download the Java development kit:
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- What is new in Java 8?
<http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>

Java 8 API Documentation

The Java 8 documentation and guides are at the following links:

- Java 8 Javadoc API documentation:
<http://docs.oracle.com/javase/8>
- JavaFX 8 Javadoc API documentation:
<http://docs.oracle.com/javase/8/javafx/api>
- Java 8 Overview Documentation:
<http://docs.oracle.com/javase/8/docs/index.html>

IDEs

Integrated development environments are used to enter source code and compile Java code. The following are popular IDEs used in the industry.

- NetBeans:
<https://netbeans.org>
- IDEA IntelliJ:
<http://www.jetbrains.com/idea>
- Eclipse:
<https://www.eclipse.org>
- BlueJ:
<http://www.bluej.org>

Deploying Applications

When deploying JavaFX applications, you'll want to use the following libraries and tools for packaging your applications.

- *Deploying JavaFX Applications:*
<http://docs.oracle.com/javafx/2/deployment/jfxpub-deployment.htm>
- *Packaging JavaFX Applications as Native Installers:*
<http://fxexperience.com/2012/03/packaging-javafx-applications-as-native>
- JavaFX and Maven:
<http://www.zenjava.com/2012/02/17/javafx-and-maven>
<http://www.zenjava.com/firstcontact/architecture/setup/install-javafx>
- JavaFX Gradle Plugin 0.3.0:
<https://bitbucket.org/shemnon/javafx-gradle>
- Launch4J:
<http://launch4j.sourceforge.net>

JavaFX 2D Shapes

The Javadoc API documentation includes abundant information about drawing shape nodes onto the JavaFX scene graph:

- <http://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Shape.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Path.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/SVGPath.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx/scene/text/Text.html>

JavaFX Color

The Javadoc API documentation also includes the following information about gradient color fills:

- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.paint/LinearGradient.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.paint/RadialGradient.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene/effect/Blend.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.paint/ImagePattern.html>

Java 8's Language Features

New in the Java 8 language are lambda expressions, defender methods, and the stream API. The following are links to forum discussions, tutorials, and videos covering many of the new language changes to Java 8.

JavaFX 2.x Builder Classes

Prior to JavaFX 8, the use of builder classes was quite widespread for very good reasons. Certain bugs, however, caused a conflict with the new Java 8, so the JavaFX teams decided to deprecate the builder classes. The following links discuss the change:

- A discussion on the Oracle JavaFX forum about the reason for deprecating builder classes:
<https://community.oracle.com/thread/2544323?tstart=0>
- An email thread from the Open JDK on the real reason why JavaFX Builder classes were deprecated in Java 8:
<http://mail.openjdk.java.net/pipermail/openjfx-dev/2013-March/006725.html>

JavaFX Printing

New to Java 8 is the JavaFX printing API. This book does not discuss how to use the printing API, but the links shown below will help you get started. See the following links for information about printing with JavaFX:

- Javadoc API documentation:
<http://docs.oracle.com/javase/8/javafx/api/javafx/print/package-summary.html>
- *All the Nodes That Are Fit to Print (Part 1)*, Phil Race (Oracle):
<https://www.youtube.com/watch?v=Ma506QBmj90>
- *All the Nodes That Are Fit to Print (Part 2)*, Phil Race (Oracle):
<https://www.youtube.com/watch?v=-fE0t7L0v-8>
- *Introduction by Example: JavaFX 8 Printing*, Carl Dea:
<http://java.dzone.com/articles/introduction-example-javafx-8>

Project Lambda

The core Java 8 added language features are lambda expressions and the stream API. The following references are road maps, blogs, and videos of topics surrounding project lambda.

- *State of the Lambda*, Brian Goetz (Oracle):
<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>
- Q & A with Brian Goetz and Stuart Marks (NightHacking tour hosted by Stephen Chin from Oracle):
<https://www.youtube.com/watch?v=vaSv3FjUIVI>
- *Java 8: Lambdas, Part 2*, Ted Neward:
<http://www.oracle.com/technetwork/articles/java/architect-lambdas-part2-2081439.html>
- *Java 8 Revealed: Lambdas, Default Methods and Bulk Data Operations*, Anton Arhipov:
<http://zeroturnaround.com/rebellabs/java-8-revealed-lambdas-default-methods-and-bulk-data-operations>
- *10 Examples of Lambda Expressions and Streams in Java 8*, Javin Paul:
<http://javarevisited.blogspot.com/2014/02/10-example-of-lambda-expressions-in-java8.html>
- *Java SE 8: Lambda Quick Start*, Oracle:
<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>
- *Java 8: Lambda Expression Basics*, Blue Sky Workshop:
<http://blueskyworkshop.com/topics/Java-Pages/lambda-expression-basics/>
- *Java 8: Closures, Lambda Expressions Demystified*, Frank Hinkel:
<http://frankhinkel.blogspot.com/2012/11/java-8-closures-lambda-expressions.html>
- Maurice Naftalin's Lambda FAQ:
<http://www.lambdafaq.org>
- Tutorials on Lambda expressions, Oracle:
<http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>
- *Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions*, Venkat Subramaniam (Pragmatic Bookshelf, 2014.
ISBN-13: 978-1937785468)
- *Java 8 Streams*, Sven Ruppert (@SvenRuppert) (German version by Entwickler Press):
http://www.amazon.de/dp/B00HALCBMC/ref=rdr_kindle_ext_tmb
<http://www.barnesandnoble.com/w/java-8-streams-sven-ruppert/1117767060?ean=9783868024982>

Nashorn

Java 8 includes a new scripting engine called Nashorn, which is a new and improved JavaScript engine for the Java runtime. The engine enables developers to use JavaScript the language to program applications. The following links and references are articles and blogs describing Nashorn.

- *Oracle's Nashorn: A next Generation JavaScript Engine for the JVM*, Julien Ponge
http://www.oraclejavamagazine-digital.com/javamagazine_twitter/20140102/?pg=60#pg60
- Open JDK's Nashorn site:
<https://wiki.openjdk.java.net/display/Nashorn/Main>
- The Nashorn blog:
<https://blogs.oracle.com/nashorn>

Properties and Bindings

Properties and bindings are essential to JavaFX when synchronizing values between JavaFX nodes. The following are great resources about read-only properties, listeners, and the role of JavaFX Beans.

- *Creating Read-Only Properties in JavaFX*, Michael Heinrichs:
<http://blog.netopyr.com/2012/02/02/creating-read-only-properties-in-javafx>
- *When to Use a ChangeListener or an InvalidationListener*, Michael Heinrichs:
<http://blog.netopyr.com/2012/02/08/when-to-use-a-changelistener-or-an-invalidationlistener>
- *The Unknown JavaBean*, Richard Bair:
https://weblogs.java.net/blog/rbair/archive/2006/05/the_unknown_jav.html
- *Using JavaFX Properties and Binding*, Scott Hommel <http://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>
- *Creating JavaFX Properties*, Michael Heinrichs:
<http://blog.netopyr.com/2011/05/19/creating-javafx-properties>
- *Pro JavaFX 8*, (Chapter 4, “Properties and Bindings”), Johan Vos, James Weaver, Weiqi Gao, Stephen Chin, Dean Iverson, (Apress, 2014):
<http://www.apress.com/9781430265740>
- Open Dolphin: A JavaFX MVC framework (founded, Dierk Koenig of Canoo Engineering):
http://open-dolphin.org/dolphin_website/Home.html
- JavaFX MVP framework based on Convention over Configuration and Dependency Injection (founded, Adam Bien):
<http://afterburner.adam-bien.com>

Layouts

Building UIs can be a challenge when it comes to layout. The following references are tutorials and libraries on how to layout JavaFX nodes.

- *Laying Out a User Interface with JavaFX 2.0*, Jim Weaver (Oracle):
<http://www.oracle.com/technetwork/articles/java/layoutfx-1536156.html>
- *Working With Layouts in JavaFX*, Joni Gordon:
<http://docs.oracle.com/javafx/2/layout/jfxpub-layout.htm>
- *Pro JavaFX 2*, Chapter 4, “Building Dynamic UI Layouts in JavaFX” (Apress, 2012):
<http://www.apress.com/9781430268727>
- *In-Depth Layout and Styling with the JavaFX Scene Builder*
http://www.youtube.com/watch?v=7Nu3_5doZK4
- Miglayout, Mikael Grev:
<http://www.miglayout.com>
- *JavaFX 2.0 EA and MigLayout*, Tom Eugelink:
<http://tbeernot.wordpress.com/2011/03/11/javafx-2-0-ea-and-miglayout>

JavaFX Tools

Building JavaFX you’ll find helpful tools from building GUIs to debugging applications in the links below:

- SceneBuilder: A tool to build JavaFX-based UIs using the FXML markup language:
<http://www.oracle.com/technetwork/java/javafx/tools/index.html>
- Scenic View tool: A tool to visually debug JavaFX UIs during runtime:
<http://fxexperience.com/scenic-view>
- Caspian Styler, Animation Spline Editor, and Derived Color Calculator:
<http://fxexperience.com/2012/03/announcing-fx-experience-tools>
- JavaFX Tooling and Runtime for Eclipse and OSGi (founder Tom Schindl):
<http://www.eclipse.org/efxclipse/index.html>
<http://tomsondev.bestsolution.at>
- Firebug lite: A browser-based debugger tool for JavaFX WebView node:
<http://stackoverflow.com/questions/17387981/javafx-webview-webengine-firebuglite-or-some-other-debugger>
- JavaFX 3D Viewer and Importer:
<https://bitbucket.org/openjfxmirrors/openjfx-8-master-rt/src/f89b7dc932af7e151387264bb13045650b42d17a/apps/experiments/3DViewer/?at=default>

- IDR Solutions showcasing a PDF viewer for JavaFX:
<http://blog.idrsolutions.com/2014/03/sneak-preview-new-javafx-pdf-viewer>
- TestFX: Easy-to-use library for testing JavaFX. TestFX provides:
<https://github.com/SmartBear/TestFX>
- Introducing MarvinFx, Hendrik Ebbers: Easily tests JavaFX controls and scenes with a special attention to properties:
<http://www.guigarage.com/2013/03/introducing-marvinfo>
- JemmyFX: Provides an API for testing JavaFX user interface in your application:
<https://jemmy.java.net/JemmyFXGuide/jemmy-guide.html>

Enterprise GUI Frameworks

GUI Frameworks enable developers to develop enterprise applications rapidly. From binding to MVC framework patterns, the following will help you become productive when developing JavaFX applications.

- Griffon (project lead Andres Almiray, @aalmiray): Griffon is an application framework for developing desktop applications in the JVM.
<http://griffon.codehaus.org>
- Open Dolphin: A JavaFX MVC framework (founded by Dierk Koenig (@mittie) of Canoo Engineering):
http://open-dolphin.org/dolphin_website/Home.html
- JavaFX MVP framework based on convention over configuration and dependency injection. Founded by Adam Bien of Oracle:
<http://afterburner.adam-bien.com>
- FXForm2 is a library providing automatic JavaFX 2.0 form generation.
<http://dooapp.github.io/FXForm2>

Domain-Specific Languages

A domain-specific language for graphics can help eliminate a lot of boilerplate code, while keeping code easier to read and manage. The following are DSLs for JavaFX in various computer languages that run on top of the Java runtime environment.

- GroovyFX: A Groovy based domain-specific language to create JavaFX applications.
<http://groovyfx.org>
- ScalaFX: A Scala based domain-specific language to create JavaFX applications.
<https://code.google.com/p-scalafx>
- RubyFX: A Ruby based domain-specific language to create JavaFX applications.
<https://github.com/jruby/jrubify>
- Visage: A JavaFX- based domain-specific language to create JavaFX applications.
<https://code.google.com/p/visage>

Custom UIs

To customize a user interface such as styling using CSS or Custom controls the following links are reference guides and blogs.

- JavaFX 8 CSS reference:
<http://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html>
- JavaFX FXML introduction:
http://docs.oracle.com/javase/8/javafx/api/javafx/fxml/doc-files/introduction_to_fxml.html
- *Styling UI Controls with CSS*:
<http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/apply-css.htm>
- *Modena Theme for JavaFX 8*, Jasper Potts:
<http://fxexperience.com/2013/03/modena-theme-update>
- *Mastering JavaFX 8 Controls*, Hendrik Ebbers (Oracle Press, 2014. ISBN 9780071833776):
<http://mhprofessional.com/product.php?isbn=0071833773>
<http://www.guigarage.com/javafx-book>
- A Mac OSX look and feel called AquaFX with Elements, by Claudine Zillmann (@etteClaudette):
<http://aquafx-project.com/project.html>
- A Windows 8 look and feel called JMetro by Pedro Duque Vieira (@P_Duke):
<http://pixelduke.wordpress.com/category/metro>
<https://github.com/JFXtras/jfxtras-styles>
- Roku-like or Apple TV-like interface called the Flatter look and feel by Hendrik Ebbers (@hendrikEbbers):
<http://www.guigarage.com/2013/09/flatter>
<http://www.guigarage.com/2013/08/boxfx-javaone-preview-1>
- GuiGarage's collection of JavaFX talks at JavaOne (provided by Parleys):
<http://www.guigarage.com/2013/12/javafx-at-javaone>
- *Why Use FXML* by Irina Fedortsova of Oracle:
http://docs.oracle.com/javafx/2/fxml_get_started/why_use_fxml.htm
- Gerrit Grunwald's JavaFX 8 examples using many different strategies for making custom controls:
<https://github.com/HanSolo/JFX8CustomControls>
- Enzo: JavaFX gauges, clocks, and many custom controls by Gerrit Grunwald:
<https://bitbucket.org/hansolo/enzo/wiki/Home>
- JFXtras: A JavaFX custom controls community:
<http://jfxtras.org>

- ControlsFX: Another custom controls community started and lead by Jonathan Giles of Oracle:
<http://fxexperience.com/controlsfx>
- *JideFX Highlights of the JideFX Beta Release (1 of 3)* by David Qiao:
<http://www.jidesoft.com/blog/2013/06/10/highlights-of-the-jidefx-beta-release-1-of-3>
<http://www.jidesoft.com>
- *Styling FX Buttons with CSS* by Jasper Potts:
<http://fxexperience.com/2011/12/styling-fx-buttons-with-css>
- *JavaFX Custom Control: Nest Thermostat Part 1-3* by Laurent Nicolas (@MrLoNee):
<http://www.javacodegeeks.com/2014/01/javafx-custom-control-nest-thermostat-part-1.html>
- *Do It Yourself: Custom JavaFX Controls* by Gerrit Grunwald:
https://www.youtube.com/watch?v=ts_b2mBix3U

Operating System Style Guidelines

In JavaFX there are many ways to develop esthetically pleasing applications; however, there are times you will need style guidance on the host operating system. The following links are many of the mainstream operating system style guidelines.

- iOS7 style guidelines:
<https://developer.apple.com/library/ios/documentation/userexperience/conceptual/MobileHIG/index.html>
- Android style guidelines:
<http://developer.android.com/design/style/index.html>
- Windows 8 style guidelines:
<http://msdn.microsoft.com/en-us/library/windows/apps/hh465424.aspx>
- Ubuntu design guidelines:
<http://design.ubuntu.com/apps>
- BlackBerry design guidelines:
http://developer.blackberry.com/native/documentation/cascades/best_practices/uiguidelines
- *HTML Styles: CSS* by w3schools.com:
http://www.w3schools.com/html/html_css.asp

JavaFX Media

JavaFX contains a powerful media API capable to process audio and video. Here you will find many links that will assist you in taking advantage of the MediaView and MediaPlayer API. Also, you will be seeing many links that help you create content.

- Oracle's tutorial to incorporating media using JavaFX:
<http://docs.oracle.com/javase/8/javafx/media-tutorial/overview.htm>
- *Audio and Video Processing in JavaFX* by Brian Burkhalter and David DeHaven (principal members of the technical staff at Oracle):
<https://www.youtube.com/watch?v=jaPUbfJx2A>
- *Pro JavaFX 2*, Chapter 8, “Media Classes” (Apress, 2014) <http://www.apress.com/9781430268727> Media College:
<http://www.mediacollege.com/adobe/flash/video/tutorial/example-flv.html>
- VP6 encoder for FLV container formats:
<http://www.wildform.com>
- Sony Vegas:
<http://www.sonycreativesoftware.com/vegassoftware>
- Adobe Premier:
<http://www.adobe.com/products/premiere.html>
- Apple's Final Cut Pro: Movie making software:
<http://www.apple.com/final-cut-pro>
- Apple's iMovie: Movie making software:
<https://www.apple.com/mac/imovie>
- Apple's Quick Time Pro: A media player:
<http://www.apple.com/quicktime>
- FFmpeg: a complete, cross-platform solution to record, convert and stream audio and video
<http://www.ffmpeg.org>
- On2 Technologies is acquired by Google:
http://en.wikipedia.org/wiki/On2_Technologies

JavaFX on the Web

Many of the following links relate to the powerful WebView node in JavaFX and how to interact with web services. Also, you will see some new and interesting concepts such as JavaFX in the browsers using JavaScript.

- Overview of the JavaFX WebView component:
<http://docs.oracle.com/javase/8/javafx/embedded-browser-tutorial/overview.htm>
- A talk on HTML5 and JavaFX by Ryan Cuprak (At the Silicon Valley JavaFX User Group):
<https://www.youtube.com/watch?v=xdgD7EBX4fw>
- Other w3c document object model APIs:
JDOM: <http://www.jdom.org>
DOM4J: <http://dom4j.sourceforge.net>

- Bug RT-35757 relating to loading files containing HTML5 with a file extension other than .html:
<https://javafx-jira.kenai.com/browse/RT-35757>
- Converting from w3c Document into a string:
<http://stackoverflow.com/questions/2567416/document-to-string>
- Loading a single text file as a string, by Pat Niemeyer:
https://weblogs.java.net/blog/pat/archive/2004/10/stupid_scanner.html:
- Single line read text file with Java by Fuad Saud:
<https://coderwall.com/p/lyalwg>
- How to read data from InputStream into String in Java by Lokesh Gupta:
<http://howtodoinjava.com/2013/10/06/how-to-read-data-from-inputstream-into-string-in-java>
- Java 8 Streams API to read a file by Cay Horstmann:
<https://weblogs.java.net/blog/cayhorstmann/archive/2013/12/06/java-8-really-impatient>
- Firebug Lite:
<http://stackoverflow.com/questions/17387981/javafx-webview-webengine-firebuglite-or-some-other-debugger/18396900>
- Reading JSON using URL Connection:
<http://docs.oracle.com/javase/tutorial/networking/urls/readingWriting.html>
- Making a POST request instead of a GET using URL Connection:
<http://stackoverflow.com/questions/4205980/java-sending-http-parameters-via-post-method-easily>
- JavaScript 12 hour am and pm format dates:
<http://stackoverflow.com/questions/8888491/how-do-you-display-javascript-datetime-in-12-hour-am-pm-format>
- OpenWeatherMap: free weather data and forecast API:
<http://openweathermap.org>
- Free Geo-location:
<http://freegeoip.net/json>
- JSObject Live connect:
https://developer.mozilla.org/en-US/docs/LiveConnect/LiveConnect_Reference/JSObject
- DataFX by Johan Vos, Jonathan Giles and Hendrik Ebbers:
<http://www.javafxdata.org>
- *Building Rich Client Applications with JSR 356: Java API for WebSocket*,
Johan Vos:
http://www.oraclejavamagazine-digital.com/javamagazine_twitter/20140102#pg67

- Bck2Brwsr: JavaFX in the browser:
<http://wiki.apidesign.org/wiki/Bck2Brwsr>
- Canvas for Bck2Brwsr:
<http://jayskills.com/blog/2013/01/22/canvas-for-bck2brwsr>

JavaFX 3D

The highly anticipated JavaFX 3D API has finally arrived, and is part of the Java 8 release. The following links are to videos and tutorials that will help enrich your JavaFX 3D skills:

- JavaFX 3D Oracle tutorial:
http://docs.oracle.com/javafx/8/3d_graphics/jfxpub-3d_graphics.htm
<http://docs.oracle.com/javase/8/javafx/graphics-tutorial/javafx-3d-graphics.htm>
- *JavaFX for Business: Monitoring a Container Terminal* at JavaOne 2012, Dierk Konig and Andres Almiray:
<https://www.youtube.com/watch?v=m5NcuWGKnDo>
- *Modeling, Texturing, and Lighting Mesh Geometry in JavaFX 3D*, John Yoon of Oracle:
<http://www.parleys.com/share.html#play/5252e81de4b0c4f11ec576b0>
- JavaFX 3D: The Third Dimension [CON7810] at JavaOne 2013, Kevin Rushforth and Chien Yang:
https://oracleus.activeevents.com/2013/connect/sessionDetail.ww?SESSION_ID=7810
- *Polaris, JavaFX, ControlsFX and F(X)yz supporting NASA Missions*, Sean Phillips:
<https://www.youtube.com/watch?v=aA5GfWc8ho4>

JavaFX Gaming

And you thought JavaFX was only for business. Well, take a look at the following links to see some fun and interesting games.

- *Writing a Tile Engine in JavaFX*, Toni Epple:
<http://www.javacodegeeks.com/2013/01/writing-a-tile-engine-in-javafx.html>
- *Tower Defense type game Parts 1-5*, Toni Epple:
<http://jayskills.com/blog/2013/08/07/tower-defense-in-javafx-2>
- Angry Duke: Physics-Based Games in JavaFX, Toni Epple:
<https://www.youtube.com/watch?v=jcSp2EoMTyE>
- JavaOne 2013 Chess Robot, Jasper Potts: In the JavaOne 2013 technical keynote the Java team demonstrated a robot controlled by Java playing chess.
<http://vimeo.com/75368008>

- A JavaFX based Game Authoring System, Mirko Sertic:
<http://www.mirkosertic.de/doku.php/javastuff/javafxgameauthoring>
- A JavaFX game tutorial Parts 1-6, Carl Dea:
<http://carlfx.wordpress.com/2012/03/29/javafx-2-gametutorial-part-1>

Java & JavaFX Embedded

The following links and references are very cool DIY projects using the combination of Java embedded and JavaFX. Many of the links below are videos that will help you see the glimpse in the future of the Internet of Things.

- Java 8 Embedded documentation:
<http://docs.oracle.com/javase/8/javase-embedded.htm>
- JavaFX based tablet computer:
<http://j.mp/DukePad>
<https://wiki.openjdk.java.net/display/OpenJFX/DukePad>
<http://vimeo.com/76486818>
- Nao Robot NightHacking:
<http://nighthacking.com/nao-robot-nighthacking-part-1>
<http://nighthacking.com/nao-robot-nighthacking-part-2>
- Pre-interview with Mark Heckler (NightHacking tour hosted, Stephen Chin of Oracle):
<https://www.youtube.com/watch?v=LXRp57B58e8>
- NightHacking with Gerrit Grunwald on Pi, XBee, and Sensor Networks. (NightHacking tour hosted, Stephen Chin of Oracle):
<https://www.youtube.com/watch?v=GTroCtrlNGY>
- NightHacking with Gerrit Grunwald and Michael Hoffer. (NightHacking tour hosted by Stephen Chin of Oracle):
<https://www.youtube.com/watch?v=cWFW4VT8auw>
- *Beyond Beauty: JavaFX, Parallax, Touch, Gyroscopes and Much More*, Angela Caicedo (Java Evangelist at Oracle) (NightHacking tour hosted by Stephen Chin of Oracle):
<https://www.youtube.com/watch?v=rYt45rs2Bjk>
- Raspberry Pi Java forum:
<http://www.raspberrypi.org/phpBB3/viewforum.php?f=81>
- *How to safely overclock your Raspberry Pi*, Eben Upton (Raspberry Pi Founder, CEO of Raspberry Pi's engineering team.):
<http://www.raspberrypi.org/archives/2008>
- *End-to-End Raspberry Pi with NetBeans IDE 8*, José Pereda (@JPeredaDnr):
<http://netbeans.dzone.com/articles/nb-8-raspberry-pi-end2end>

Software and Device Manufacturers

In the last few chapters we discussed Java embedded devices and gesture-based input devices. The following links reference various manufacturers.

- Raspberry Pi:
<http://www.raspberrypi.org>
- Arduino:
<http://www.arduino.cc>
- BeagleBoard:
<http://beagleboard.org/Products/BeagleBone>
- Leap Motion:
<https://www.leapmotion.com>
- Microsoft Windows 8 Surface:
<http://www.microsoft.com/surface/en-us>
- Oculus Rift:
<http://www.oculusvr.com>
- Thalmic Labs:
<https://www.thalmic.com/en/myo>

JavaFX Communities

So you want to get involved with the JavaFX community? Please check out the following links.

- Java.net JavaFX community site:
<https://www.java.net/community/javafx>
- FXExperience: JavaFX News, Demos and Insight (@fxexperience):
<http://fxexperience.com>
- Nighthacking (@_nighthacking): Hosted by Stephen Chin. A tour around the world to see everything about Java, JavaFX, and IoT. Amazing live talks.
<http://nighthacking.com>
- Oracle's JavaFX Community portal to Real World Use Cases, Community Support, Third Party tools and Open JFX:
<http://www.oracle.com/technetwork/java/javase/community/index.html>
- JFXtras: A JavaFX custom controls community:
<http://jfxtras.org>
- ControlsFX: Another custom controls community, started by Jonathan Giles of Oracle:
<http://fxexperience.com/controlsfx>
- Feed Cluster: Any and all JavaFX news:
<http://javafx-blogs.feedcluster.com>

- Silicon valley JavaFX user's group:
<http://www.meetup.com/svjugfx>
- Silicon valley JavaFX user's group Live stream:
<http://www.ustream.tv/channel/silicon-valley-javafx-user-group>
- Oracle Forums on JavaFX:
https://community.oracle.com/community/developer/english/java/javafx/javafx_2.0_and_later

Applications

Many of the following links are rich GUI applications written using the JavaFX platform.

- Picmodo (founded by Jens Deter, @Jerady): A JavaFX-based picture database application:
http://www.jensd.de/wordpress/?page_id=22
- VWorkflows by Michael Hoffer (@mihosoft): A Scientific Visualization Tool
<https://github.com/miho/VWorkflows>:
<http://mihosoft.eu>
- eteoBoard: The interactive digital task board. A touch and gesture-based application tool for team collaboration.
<http://www.eteoboard.de>
<http://www.saxsys.de>
- PiDome client (by John Sirach): The PiDome client is the user interface for the PiDome server. The web interface is primarily used for configuring the server:
<https://www.youtube.com/watch?v=KOvmwH2EQqQ>
<http://pidome.wordpress.com/clients>
- Object Wheel (Founder and CEO Mike Tallent @abigdreamer): A user-defined social media platform and RAD framework.
<https://en.wikipedia.org/wiki/Objectwheel>
- FlexGanttFX: Aircraft / Flight Scheduling Application (by Dirk Lemmermann @dlemmermann):
<http://www.dlsc.com>
https://www.youtube.com/channel/UCFyRQ_euxxPDwlqyhff-xoQ
- BaufiFX (by Thomas Bolz): A home finance calculator:
<http://www.baufifx.de>
- SnapCode is an IDE and RAD tool for education, entertainment and the enterprise. Report Mill Software.
<http://www.reportmill.com/snap>
- FXMark: JavaFX benchmarking tool by Felix Bembrick (@Lucan1d):
<http://justmy2bits.com/2014/02/07/fxmark-javafx-benchmarking-software-to-be-released-soon>

- Modellus is a freely available application that enables students and teachers (high school and university) to use mathematics to create or explore models interactively. Modellus was developed by Vitor Duarte Teodoro, João Paulo Duque Vieira and Pedro Duque Vieira:
<http://modellus.co/index.php/en>
- MIME Browser by Christoph Nahr is a compact JavaFX desktop application for browsing MIME messages that are locally stored in standard EML files.
<http://kynosarges.org/MimeBrowser.html>

Java/JavaFX Books and Magazines

The following links are newer book titles that relate to the new Java 8 and JavaFX 8 platform.

- *Pro JavaFX 8*, (Johan Vos, James Weaver, Weiqi Gao, Stephen Chin, Dean Iverson (Apress, 2014. ISBN: 978-1-4302-6574-0)
<http://www.apress.com/9781430265740>
- *Java 8 Recipes*, Josh Juneau (Apress, 2014. ISBN: 978-1-4302-6827-7)
<http://www.apress.com/9781430268277>
- *JavaFX Rich Client Programming on the NetBeans Platform* , Paul Anderson and Gail Anderson (Addison-Wesley Professional, 2014. ISBN: 978-0321927712):
https://blogs.oracle.com/geertjan/entry/new_book_javafx_rich_client
<http://www.amazon.com/JavaFX-Client-Programming-NetBeans-Platform/dp/0321927710>
- *Mastering JavaFX 8 Controls*, Hendrik Ebbers (Oracle Press, 2014. ISBN: 9780071833776):
<http://mhprofessional.com/product.php?isbn=0071833773>
<http://www.guigarage.com/javafx-book>
- *Quick Start Guide to JavaFX*, J.F. DiMarzio (Oracle Press, 2014. ISBN: 978-0071808965 by):
<http://www.mhprofessional.com/product.php?isbn=0071808965>
- *Java SE 8 for the Really Impatient*, Cay S. Horstmann (Addison-Wesley, 2014. ISBN 978-0321927767)
<http://www.addison-wesley.de/9780321927767.html>
- *Mastering Lambdas*, Maurice Naftalin (Oracle Press, 2014. ISBN: 007-1829628):
<http://www.mhprofessional.com/product.php?isbn=0071829628>
- *Java Magazine* from Oracle:
<http://www.oracle.com/technetwork/java/javamagazine/index.html>

Author Blogs

If you are having trouble with this book's content or code examples, please feel free to comment or mention issues at the book's catalog page at apress.com, in the errata tab. If you are totally stumped, please feel free to contact the book authors via Twitter or their blogs.

- Following are the book's authors with their Twitter accounts and blog URLs:Carl Dea (@carldea):
<http://carlfx.wordpress.com>
- Gerrit Grunwald (@hansolo_):
<http://harmoniccode.blogspot.com>
- Mark Heckler (@MkHeck):
<https://blogs.oracle.com/javajungle>
- José Pereda (@JPeredaDnr):
<http://jperedadnr.blogspot.com>
- Sean Phillips (@SeanMiPhillips):
<http://birdasaur.tumblr.com>

Index

A

Animation
 API, 139
 key frames and values, 140–141
 timeline, 141–142
AquaFX, 153
Arduino board
 Due (front), 270
 Uno (front and back), 269–270
 USB cable, 270–271
Arduino IDE
 bill of materials, 277
 blink sketch, 276
 code files, sketches, 275
 Fritzing, 277
 installation, 274
 jSSC, 283–284
 Linux Setup, 274
 Mac OS X Setup, 274
 photocell sketch, testing, 280–281
 security warning, 274
 Serial Monitor, 282
 serial port, 273
 serial reading, 282–283
 Sketch Photocell.ino, 278–279
 tinkering, 278
 Windows installation, 271
 working process, 279–280

B

Background process
 classes, 118
 createWorker() convenience method, 120
 demonstration, 118–119
 GUI developers, 117
 messages and progress, 120

output, 120
rebinding task, 121
Blink, 215
Bootable SD Card
 creation steps, 307
 NOOBS download, 308
SD Association
 Formatter tool, 307–308
BorderPane layout. See Layouts
BoxFX project, 153

C

Custom UIs, 372–373
 custom controls
 “extend Region” approach, 182
 LED (see Light-emitting diode (LED))
 Skin.class, 182
 JavaFX CSS (see JavaFX CSS)
Scene Builder
 FXML, 171
 javafx.fxml.FXMLLoader class, 172
 tool, 170–171
theming
 AquaFX, 153
 BoxFX project, 153
 getStylesheets().add(String URL)
 method, 155
 JavaFX CSS themes, 153–154
 JMetro, 153
 look-and-feel
 themes, 151–152, 157, 160–161
 Modena theme, 153
 pseudo-class selectors, 153
 setUserAgentStyleSheet(String URL)
 method, 154–155
 Sky look and feel, 153
 switching themes, 156–157

D

Deploying JavaFX applications, 366
 3D, gestures
 active node, 345
 description, 344
 OnTouchStationary, 345
 OnZoom event, 346
 rotation, 345–346
 TriangleMesh, 345
 zooming, 347
 Domain-specific languages (DSLs), 371
 Drawing lines
 dash offset property value, 37
 DrawingLines.java
 source code, 33–34, 36
 getStrokeDashArray().add() method, 36
 lambda expressions, 37
 stroke color, width and line cap, 36
 Drawing shapes
 complex shape, 39–42
 cubic curve, 42–43
 donut shape, 45–46
 drawing rectangles, 37–38
 ice cream cone, 44
 JavaFX API, 38
 JavaFX’s Shape API, 37
 smile shape, 45
 Drawing text
 capabilities, 53
 changing text fonts, 56–57
 DrawingText.java, 54–55
 the font() method, 58
 drop shadow and
 reflection effects, 58
 Java 8’s lambda syntax, 58
 JavaFX node, 53
 Java inheritance hierarchy, 53
 text node’s baseline (x, y)
 coordinate position, 55

E

EDA. See Event-driven architecture (EDA)
 Event-driven architecture (EDA), 183

F

Fixed network (IP) address configuration
 wired networking, 310–311
 wireless networking, 311–312

G

Gestures
 3D, 344–347
 events, 339–341
 MouseEvent, 340
 touch events, 341–344
 TouchPoint, 340
 Graphics, JavaFX
 animation, 139–141
 loading images
 constructor’s parameters, 124
 description, 123
 protocol file and HTTP, 125
 strategies, 124
 NewsTickerPhotoViewer
 (see NewsTickerPhotoViewer)
 photo viewer application
 code listing, 128, 130, 132–133
 custom button panel, 138–139
 features, 126–127
 ImageView, 136
 instance variables, 135
 OnDragDropped event, 137–138
 OnDragOver event, 136
 start() method, 135
 user instructions, 127
 viewing images
 creation, 125
 description, 123
 GUI frameworks, 371

H

Hero Picker application, 108–110
 HTTP live streaming format (HLS), 185

I

Integrated development environments (IDEs), 271, 366

J, K

- Java 8 API documentation, 365
 Java 8 Java Development Kit (JDK)
 environment variables, 6–8
 installation
 complete, 5–6
 Destination Folder, 5
 Next button, 3
 optional features, 3–4
 progress bar, 4
 security warning, 2
 Java 8 language
 JavaFX 2.x Builder Classes, 367
 JavaFX printing, 367
 Nashorn, 369
 project lambda, 368
 Java 8 SDK, 365
 JavaBean class, getter and
 setter methods, 74
 JavaFX
 3D, 376
 and Java embedded, 377
 books and magazines, 380
 Charts API
 axes, creation, 289, 293
 luxSeries, 296
 ObservableList of Series, 294
 photocell example, 296–297
 pie charts, 289
 properties, 294
 scene, creation, 295
 Serial class, 293
 serial.disconnect() method, 296
 source code, 289–292
 tamMax value, 294–295
 communities, 378
 do-it-yourself (DIY) skills, 303
 functionality
 Arduino Test 2.
 Photocell&LED, 302–303
 CSS File photocell.css, 299–301
 LED, 297–298
 Photocell.ino, modification, 298
 Serial.java, modification, 299
 gaming, 376
 GUI applications, 379–380
 HTML5 content (WebView)
 analog clock source code, 225–227
 HTML5 analog clock, 224–225
 Inkscape and SVG, 228–229
 working, 227–228
 layout, 370
 media, 373–374
 operating system style guidelines, 373
 properties and bindings, 369
 serial communications
 ardPort variable, 287–288
 disconnect() method, 289
 jSSC, 287
 parameters, 288–289
 Serial class, 285–287
 tools, 370–371
 Weather Widget
 condition, 230–231
 source code, 231–235
 working, 238–239
 WebEngine
 HTML DOM content, 217
 HTTP POST request, 223–224
 JavaScript bridge, 218
 JavaScript to Java
 communication, 219–220
 Java to JavaScript
 communication, 218
 limitations, 221
 one-liner, file reading, 223
 org.w3c.dom.Document
 (DOM) object, 217
 raw XML content, 218
 RESTful call requests, 221–222
 URL and HttpURLConnection APIs, 220
 WebEngine API, 215
 web engine's load() method, 216
 WebEvents, 229–230
 web view node, 374–375
 JavaFX 2D shapes, 366
 JavaFX 3D
 first-person perspective
 Keyboard Event Handler, 252
 MouseEvent Handler, 253–254
 interaction
 3D objects, 251
 primitives, Mouse Event, 250

- JavaFX 3D (*cont.*)
objects, creation, 242
PerspectiveCamera, 243
primitives
 cylinder, 243–245
 grouped, 248–249
 multiple transformation, 247–248
 primitives, 249
 rotations, 246
 transformations, 245
TriangleMesh class
 buildPyramid() method, 256–257, 259
 custom 3D objects, 255–256
 default light, 259–260
 DrawMode.FILL, 262–263
 fill parameter, 261
 MeshView, 261
 multiple pyramids, 264
 Point Light, 266
 pyramids of different colours, 263
 Roll Camera!, 264–265
 winding and wuthering, 256–259
- JavaFX color, 367
- JavaFX CSS
 CSS class type selectors, 163
 CSS File mobile_buttons.css, 166–167
 CSS id type selectors, 162
 -fx-based styling
 properties (rules), 167
 node's styling property, 168
 number pad application, 166
 rules, 169–170
 selector patterns
 common selector patterns, 163
 pseudo-class selectors, 164
 selectors, 162
 selector style definition block, 168
 smart phone's number pad, 165
 styling properties (rules) limitations, 169
- JavaFX fundamentals
 drawing lines (see Drawing lines)
 drawing shapes, 37
 drawing text (see Drawing text)
 JavaFX lines (see JavaFX lines)
 JavaFX scene graph, 31
 painting colors (see Painting colors)
- JavaFX Hello World Application
 command-line prompt, 23, 25
Netbeans IDE
 HelloWorld project, 19–20
 Java Platform Manager, 18–19
 launch, 17
 Main Project, 22
 project and categories, 17
 Project Location and Project
 Folder fields, 18
 Project Properties, 20–21
source code
 JavaFX Node, 27
 JavaFX Scene Graph, 26
 main() method, 25–26
 overridden start() method, 26
 packaging, 27–28
- JavaFX lines
 geometry, 31
 javafx.scene.shape.Shape
 Class Properties, 33
 monitors and printers, 31
 objects, 32
 setter methods, 32
 standard screen coordinate system, 31–32
 the javafx.scene.shape.Line class, 32
- JavaFX's
 and gestures (see Gestures)
 animation, 341
 Shape3D object, 344
 thread, 351, 355
 touch-enabled device, 339
- Java Simple Serial Connector (jSSC), 283–284
- JDK8 installation
 development machine, 316
 Raspberry Pi, 316–318
- jSSC. See Java Simple Serial Connector (jSSC)
-  **L**
- Lambda expressions
 anonymous inner classes, 62
 binding
 bidirectional, 80
 high-level, 81
 JavaFX API packages, 80
 low-level, 81

- default methods
 - aggregate operations, 68
 - Cat domain model, 68–69
 - coding, cat interfaces, 73
 - Java interfaces, 68
 - tiger and lion classes, 69
 - virtual extension methods, 68
- description, 61
- functional interfaces
 - aggregate operations, 66, 68
 - vs. imperative languages, 66
 - Java 8’s functional interfaces, 65
- Greek symbol, 61
- JavaFX JavaBean, 78
- Java™ Programming Language, 62
- logon dialog window
 - description, 82
 - instructions, 83
 - source code, JavaFX, 89
- properties
 - JavaFX, 75
 - read-only, 77
 - read/writable, 76
 - unidirectional/bidirectional binding, 75
- property change support
 - ChangeListener instance, 78–79
 - InvalidationListener instance, 79
 - JavaFX property objects, 78
- syntax, 64
- UI patterns, 74
- variable capture, 65
- Layouts
 - BorderPane, 96–97
 - FlowPane
 - creation, 96
 - flow alignment, 96
 - nodes, 96
 - GridPane
 - column constraints, 98, 100
 - file demonstration, 98
 - FlowPane layout, styling, 101
 - horizontal space, 101
 - in business applications, 97
 - output, 99
 - root node, 100
 - top banner area, 101
 - UI controls and, 100
- GUI applications, 91
- HBox
 - argument constructor, 93
 - code, 92
 - java file demonstration, 92
 - output, 93
 - rectangles, 93
 - Shape nodes, 92
 - working, 92
- VBox
 - description, 94
 - java file demonstration, 94
 - output, 94
 - rectangle child nodes, 95
 - size, 95
- Leap Motion controller
 - Cartesian coordinate system, 350
 - description, 348
 - diagnostic visualizer application, 349
 - hardware, 348
 - infrared LEDs, 349
 - SDK, 350–351
 - shooting star, 351, 353–355
 - software, 348
 - water ripple effect, 355–358, 360–363
- LED. See Light-emitting diode (LED)
- Light-emitting diode (LED), 297
 - code, structure, 174
 - CSS file, 179
 - initialization code, 177–178
 - properties, 175–177
 - region-based custom LED control, 182
 - resizing, 180
 - results, 181
 - structure, 172–173
 - vector drawing, 173
 - vector, parts, 174
 - visualization code, 178

M

- Media, JavaFX
 - APIs, 183
 - audio player, 185 (see also MP3 Player)
 - event-driven architecture (EDA), 183
 - EventHandler, 184
 - media and mediaplayer events, 184–185
 - video player

- Media, JavaFX (*cont.*)
 closed-captioning video, 212
 media marker events, 213
 MediaView node, 211
 MPEG-4, 205–206
 playing video, 212
 source code, 207–210
 stage window, 211
 VP6 .flv, 206
 working process, 211, 213
- Modena theme, 153
- MP3 player
 audio player's UI, 186
- AudioSpectrumListener interface
 close button, 205
 inbound parameters, 205
 visualization, 203–204
- cleaning up, 202
- close button, 187
- custom button panel, creation, 200–201
- drag-and-drop support, 202
- instance variables, 198–199
- media metadata, 203
- OnEndOfMedia event, 203
- OnReady event, 203
- play progress, rewind,
 and fast forward, 201–202
- progress and seek position slider control, 187
- setup container node, visualizations, 200
- source code, 187–192
- stage window, 199
- stop, play, and pause buttons, 186
- working process, 198
- MPEG-4, 205–206
- N**
- Nashorn, 369
- NetBeans 8
 configuration, 319–320
 installation, 319
- Raspberry Pi
 Create New Configuration, 334
 Light Cycle game building steps, 333
 Light Cycle running, 336
 program running on
 MacBook Pro, 337–338
- Project Properties box, 333–334
 remote configuration, 335
 screen in action, 337
 TRace ON, 336
- Raspberry Pideployment, 336
- NetBeans IDE
 installation
 Check for Updates option, 13
 destination directory, 11
 drop-down, 11
 JDK 8, 12–13
 JUnit license agreement, 10
 license agreement, 9–10
 Next button, 8
 progress bar, 14
 Run button, 8
 setup complete, 15–16
 start page, 16–17
- JavaFX Hello World Application. JavaFX
 Hello World Application
- NewsTickerPhotoViewer
 button panel, fade in/out transition, 145
 createTickerControl()
 custom news
 ticker control, 146–147
 news feed container, 149
 repositioning control, 148
 rounded rectangular
 clip region, 148
 scene's nodes, 148
 scroll animation, 150
 scroll ticker content, 149–150
 definition, 142
 enhancements and descriptions, 142–143
 photos, fade in/out transition
 createWorker() method, 144
 ParallelTransition, 143
 SequentialTransition, 143
- O**
- ObservableList collection class
 action code, 111
 definition, 108
 FXCollections, factory class, 110
 ListViews, 108–110
- Operating system style guidelines, 373

P, Q

Painting colors
 applying text effects, 59
 common color fills, 47
 description, 47
 gradient color, 51
 JavaFX, 48
PaintingColors.java, 49–51
 radial gradient, 51
 reflective cycle gradients, 53
 semitransparent gradients, 52–53

R

Raspberry Pi
 code explanation
`calculateDestination()`
 Method, 331–332
 capturing user input, 329
`checkCollision()` Method, 332
 Draw and Play, 330
`drawGameGrid()` Method, 327
`drawWalls()` method, 328
`forEach()` method, 333
 Game Grid creation, 327
`handleKeyPress()` Method, 329
 RaspiCycle member variables, 326
`runLightCycle()` Method, 330–331
 TRON Light Cycles game, 331
 two-dimensional (2D) points, 327
 JDK8 installion, 316–318
 models, 306
 NetBeans 8 (see NetBeans 8)
 preparation, JavaFX
 bootable SD card, 307–309
 configuration, 309–310
 fixed IP address, 310–312
 remote connection, 313
 RaspiCycle, 320–325
 specifications, 306
 SSH (see Setting up
 Secure Shell (SSH))
 RESTful web services, 221
 RIA. See Rich Internet applications (RIA)
 Rich Internet applications (RIA), 123, 215

S

Scalable Vector Graphics (SVG), 215
 SDK, 350–351
 Setting up Secure Shell (SSH)
 connected window, 315
 host key warning, 315
 PuTTY Configuration
 window, 313–314
 Shooting star
 Listener subclass, 351–352
`screen.intersect()`, 354
 source code,
 JavaFX class, 352–353
 SVGPath node, 352
 trajectory, 354–355
 Software and device
 manufacturers, 378
 Software installation
 Java 8 JDK (see Java 8 Java
 Development Kit (JDK))
 NetBeans IDE (see NetBeans IDE)
 operating systems, 1
 SSH. See Setting up
 Secure Shell (SSH)
 SVG. See Scalable Vector Graphics (SVG)

T

TableViews
 Bosses and Employees
 code block, 116
 column headers, 117
`getPeople()` method, 114–115
 inner class, 116
`ListView` control, 111, 115
 main GUI code, 113–114
`ObservableList` clearing, 117
 output, 115
`Person` class, 112
 uses, 111
 Touch events
 circle animation, 341–344
 GUI component, 344
`start()`, 344
 with paths, 343

U

UI controls

- code, visual separator, [106](#)
 - context menus, [107](#)
 - handler code, [103](#)
 - key combinations, [107](#)
 - menu bar setup, [105](#)
 - menus
 - and menu items, creation, [102](#)
 - `CheckMenuItem`, [102](#)
 - `CustomMenuItem`, [102](#)
 - definition, [101](#)
 - `MenuItem` class, [102](#)
 - `MenuItem` subclasses, [102](#)
 - `RadioMenuItem`, [102](#)
 - save menu items, [102](#)
 - `SeparatorMenuItem`, [102](#)
 - mnemonics, [106–107](#)
 - radio and separator menu items, [106](#)
 - working with menus, [103–105](#)
- User interfaces (UIs). See Custom UIs

VVP6 .flv, [206](#)**W, X, Y, Z**

WaterLeapFX

- `AnimationTimer`, [361](#)
- `BooleanProperty`, [360](#)
- for loop, [362–363](#)
- `ImageViewer`, [361](#)
- `onConnect` method, [360](#)
- `onFrame` method, [360](#)
- `PixelReader`, [361](#)
- `PixelWriter`, [363](#)
- running, [364](#)
- source code, [355–359](#)
- z-coordinate, [362](#)

WebKit, [215](#)

Windows 8

- 3D, [344–347](#)
- touch events, [341–344](#)

JavaFX 8

Introduction by Example
Second Edition



Carl Dea
Mark Heckler
Gerrit Grunwald
José Pereda
Sean M. Phillips

Apress®

JavaFX 8: Introduction by Example

Copyright © 2014 by Carl Dea, Mark Heckler, Gerrit Grunwald, José Pereda, and Sean M. Phillips

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4302-6460-6

ISBN-13 (electronic): 978-1-4302-6461-3

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

President and Publisher: Paul Manning

Lead Editor: Jonathan Gennick

Development Editor: James Markham

Technical Reviewer: David Coffin

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jim DeWolf, 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, Steve Weiss

Coordinating Editor: Jill Balzano

Copy Editor: James Compton

Compositor: SPi Global

Indexer: SPi Global

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. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

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 material 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/.



To the Java and JavaFX community.

Contents

About the Authors.....	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Introduction	xxiii
■ Chapter 1: Getting Started.....	1
Installing Required Software	1
Installing the Java 8 Development Kit	2
Setting Environment Variables.....	6
Installing the NetBeans IDE	8
Creating a JavaFX Hello World Application.....	17
Using the Netbeans IDE	17
Using the Command-Line Prompt.....	23
Walking Through the Code	25
JavaFX Scene Graph.....	26
JavaFX Node.....	27
Packaging a JavaFX Application.....	27
Summary.....	29

■ Chapter 2: JavaFX Fundamentals	31
JavaFX Lines	31
Drawing Lines	33
Drawing Shapes	37
Drawing Complex Shapes	38
A Complex Shape Example	39
The Cubic Curve	42
The Ice Cream Cone.....	44
The Smile.....	45
The Donut	45
Painting Colors	47
An Example of Color	47
Gradient Color.....	51
Radial Gradient.....	51
Semitransparent Gradients.....	52
Reflective Cycle Gradients.....	53
Drawing Text.....	53
Changing Text Fonts	56
Applying Text Effects	59
Summary.....	60
■ Chapter 3: Lambdas and Properties	61
Lambda	62
Lambda Expressions	63
Syntax.....	63
Variable Capture	64
Functional Interfaces.....	65
Functional vs. Imperative Languages	65
Aggregate Operations.....	66

Default Methods	68
An Example Case: Cats Large and Small	68
Code for the Example	69
Explanation of the Code.....	73
Properties and Binding	74
UI Patterns	74
Properties	74
Types of JavaFX Properties.....	75
JavaFX JavaBean.....	77
Property Change Support	78
Binding	79
Bidirectional Binding	80
High-level Binding	80
Low-Level Binding.....	81
A Logon Dialog Example.....	82
Login Dialog Source Code.....	83
Explanation of the Code.....	86
Summary.....	89
■ Chapter 4: Layouts and UI Controls	91
Layouts.....	91
HBox	92
VBox.....	94
FlowPane	96
BorderPane.....	96
GridPane	97
UI Controls.....	101
Menus.....	101
Creating Menus and Menu Items.....	102
Invoking a Selected MenuItem	103
Example: Working with Menus	103
Additional Ways to Select Menus and Menu Items.....	106

The ObservableList Collection Class	108
Working with ListViews	108
Working with TableViews	111
Generating a Background Process	117
Summary.....	121
■ Chapter 5: Graphics with JavaFX	123
Working with Images.....	123
Loading Images	124
Viewing Images	125
A Photo Viewer Example.....	126
Photo Viewer Features.....	127
Photo Viewer User Instructions.....	127
Photo Viewer Code Listing	128
Walking Through the Code.....	135
Animation	139
What are Key Values?	140
What are Key Frames?	140
What is a Timeline?	141
A News Ticker Example	142
Adding a Fade in/out Transition to Photos	143
Adding a Fade in/out Transition to the Button Panel.....	145
Creating a News Ticker Control	146
Summary.....	150
■ Chapter 6: Custom UIs	151
Theming	151
Applying JavaFX CSS Themes	153
An Example of Switching Themes	156
JavaFX CSS	162
What Are Selectors?	162
How to Define -fx- Based Styling Properties (Rules).....	167
Obeying the JavaFX CSS Rules.....	169

Scene Builder	170
Launching Scene Builder.....	170
FXML.....	171
Loading FXML into a Scene	172
Custom Controls	172
The LED Custom Control.....	172
Structure of the LED Custom Control Example Code	174
The Properties of the LED Control.....	175
The Initialization Code of the LED Control.....	177
Other Ways to Create a Custom Control.....	182
Summary	182
Chapter 7: Media with JavaFX.....	183
Media Events.....	183
Playing Audio.....	185
An MP3 Player Example	186
The Stop, Play, and Pause Buttons	186
The Progress and Seek Position Slider Control.....	187
The Close Button	187
MP3 Audio Player Source Code	187
How It Works.....	198
Playing Video	205
MPEG-4.....	205
VP6 .flv.....	206
A Video Player Example.....	206
Video Player Source Code.....	207
How It Works.....	211
Simulating Closed Captioning: Marking a Position in a Video	212
Closed-Captioning Video Example	212
How It Works.....	213
Summary	213

■ Chapter 8: JavaFX on the Web.....	215
WebEngine	215
 WebEngine's load() Method.....	216
 WebEngine's loadContent() Method	217
 HTML DOM Content	217
 The URL and HttpURLConnection APIs.....	220
Viewing HTML5 Content (WebView)	224
 Example: An HTML5 Analog Clock	224
 Analog Clock Source Code.....	225
 How It Works.....	227
WebEvents.....	229
Weather Widget Example	230
 Source Code	231
 How It Works.....	238
Summary.....	240
■ Chapter 9: JavaFX 3D	241
Basic 3D Scenes in JavaFX	241
Primitives	243
Interacting with Your Scene	250
Beyond the Basics.....	254
Summary.....	267
■ Chapter 10: JavaFX and Arduino	269
The Arduino Board.....	269
The Arduino IDE	271
 Windows Setup.....	271
 Mac OS X Setup.....	274
 Linux Setup.....	274
 Running the IDE	274

Serial Reading	282
Java Simple Serial Connector.....	283
JavaFX, the Charting API, and Sensor Readings.....	284
Serial Communications.....	285
The JavaFX Charts API.....	289
Adding More Functionality.....	297
Now, What?.....	303
Summary.....	304
■ Chapter 11: JavaFX on the Raspberry Pi.....	305
Preparing the Raspberry Pi for JavaFX	307
Create a Bootable SD Card	307
Configuring the Raspberry Pi.....	309
Connecting to the Pi Remotely	313
Installing JDK8	316
Installing JDK8 on Your Development Machine	316
Installing JDK8 on the Raspberry Pi	316
Installing and Configuring NetBeans 8 for Raspberry Pi Development	318
Installing NetBeans 8.....	319
Configuring NetBeans 8 for the Pi	319
RaspiCycle: The Code	320
Explanation of the Code	326
Using NetBeans with the Raspberry Pi.....	333
Summary.....	338
■ Chapter 12: JavaFX and Gestures.....	339
Recognizing Gestures in Your Application	339
Example: Animating Shapes along a Path using Touch Events.....	341
The Leap Motion Controller	348
How It Works.....	349
Getting Started with the Leap SDK.....	350

Creating a Shooting Star	351
Running the Example.....	354
Creating a Water Ripple Effect	355
How It Works.....	360
More Examples.....	364
Summary.....	364
■ Chapter 13: References	365
Java 8 SDK	365
Java 8 API Documentation.....	365
IDEs	366
Deploying Applications	366
JavaFX 2D Shapes.....	366
JavaFX Color.....	367
Java 8's Language Features.....	367
JavaFX 2.x Builder Classes.....	367
JavaFX Printing.....	367
Project Lambda.....	368
Nashorn	369
Properties and Bindings	369
Layouts.....	370
JavaFX Tools.....	370
Enterprise GUI Frameworks.....	371
Domain-Specific Languages	371
Custom UIs	372
Operating System Style Guidelines	373
JavaFX Media	373
JavaFX on the Web	374
JavaFX 3D	376
JavaFX Gaming.....	376

Java & JavaFX Embedded	377
Software and Device Manufacturers	378
JavaFX Communities	378
Applications	379
Java/JavaFX Books and Magazines	380
Author Blogs	381
Index.....	383

About the Authors



Carl P. Dea is currently a senior software engineer. He is a co-author of *Java 7 Recipes*, the author of *JavaFX 2 Introduction by Example*, and a technical reviewer of *Pro JavaFX 2*, all from Apress. Carl has been developing software for over 20 years, for many clients from Fortune 500 companies to nonprofit organizations. He has written software ranging from mission-critical applications to ecommerce-based Web applications. His passion for software development started when his middle school science teacher showed him a TRS-80 computer. Carl has been using Java since the very beginning, and he has been a huge JavaFX enthusiast since the early days when it was called F3. His current software development interests are UI/UX, game programming, data visualizations, embedded systems, smart phones, AI, and robotics.

When Carl is not working, he and his wife enjoy watching their younger daughter perform gymnastics. Carl and his wife are also proud of their older daughter, who is currently pursuing her English degree at the University of Maryland Baltimore County. Carl lives in Pasadena, Maryland, United States. He blogs at <http://carlfx.wordpress.com>. Carl is also connected to LinkedIn at <http://www.linkedin.com/in/carldea>, and his Twitter handle is @carldea.



Mark Heckler is a Java software architect/engineer with development experience in numerous environments. He has worked for and with key players in the manufacturing, emerging markets, retail, medical, telecom, and financial industries to develop and deliver critical capabilities on time and on budget. Currently, he works primarily with large government and commercial customers using Java throughout the stack and across the enterprise. He also participates in open source development at every opportunity, being a JFXtras project committer and developer of DialogFX, MonologFX, co-developer

of Autonomous4j, and more. When Mark isn't working with Java, he enjoys writing about his experiences at the Java Jungle website (<https://blogs.oracle.com/javajungle>), his personal website (<http://www.thehecklers.org>), and on Twitter (@MkHeck).

Mark lives with his very understanding wife, three kids, and dog in St. Louis, Missouri, United States.



Gerrit Grunwald is a software engineer with more than ten years of experience in software development. He has been involved in development of Java desktop applications and controls. His current interests include JavaFX, HTML5, and Swing, especially development of custom controls in these technologies. Gerrit is also interested in Java-driven embedded technologies like JavaSE Embedded on Raspberry Pi, i.MX6, BeagleBone Black, CubieBoard2, and similar devices. He is a true believer in open source software and has participated in popular projects like JFXtras.org as well as his own projects (Enzo, SteelSeries Swing, SteelSeries Canvas). Gerrit is an active member of the Java community, where he founded and leads the Java User Group Münster (Germany), co-leads the JavaFX and IoT community at Oracle, and is a JavaOne RockStar and Java Champion. He is a speaker at conferences and user groups internationally.



Dr. José Pereda, PhD in Structural Engineering, works as an Assistant Professor in the School of Engineers at the University of Valladolid in Spain. His passion is related to applying programming to solve real problems. Working with Java since 1999, he now divides his time between JavaFX and the Embedded world, developing commercial applications and open source projects (JFXtras, <https://github.com/jperedadnr>), blogging (<http://jperedadnr.blogspot.com.es/>), tweeting (@JPeredaDnr), and speaking at conferences (JavaOne, W-JAX, Jfokus). José lives with his wife and four kids in Valladolid, Spain.



Sean M Phillips is a Java software architect currently working on ground systems for Space Science missions at the NASA Goddard Space Flight Center. He has developed embedded systems, along with modeling and simulation and visualization software for projects including commercial and military RADAR, heavy manufacturing, Battlespace Awareness, and Orbital Flight Dynamics. Sean is the lead developer of F(X)yz (<http://birdasaur.github.io/FXyz/>), a free third-party library for JavaFX 3D components and data visualization tools. Sean enjoys sharing his experiments through blog entries on various sites including Java.net (<https://weblogs.java.net/blogs/sean.mi.phillips>), The NetBeans DZone (<http://netbeans.dzone.com/users/seanmiphillips>), and personal media (Twitter @SeanMiPhillips).

Sean lives in Bowie, Maryland, United States, with his very supportive wife Zulma and sons Sebastian and Sean Alexander.

About the Technical Reviewer



David Coffin is the author of *Expert Oracle and Java Security* (Apress, 2011). Find his recent blogs at <http://oraclejavasecure.blogspot.com>. He is an IT analyst working at the Savannah River Site, a large U.S. Department of Energy facility. For more than 30 years, his 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 completed one master's degree program and has begun several others. As a family man, he has raised eight children. Coffin is an athlete 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 for the first edition. A big thanks to Jim Weaver for being a great mentor and friend. 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. Hats off to my co-authors Gerrit Grunwald, Mark Heckler, José Pereda, and Sean Phillips for their amazing talents. My co-authors are truly superheroes who came to my rescue to make this book possible. A huge shout-out to the wonderful people at Apress for their professionalism and support. A special thanks to Jonathan Gennick for believing in me and whipping me into shape again. Thanks to Jill Balzano for keeping us on track and getting us across the finish line. Thanks to James Compton for stepping in for Jonathan G. while things began to heat up. Thanks to all who follow me on Twitter, especially the topics (hash tags) related to JavaFX, UI/UX, and IoT. Also, thanks to the authors (Jim Weaver, Weiqi Gao, Stephen Chin, Dean Iverson, and Johan Vos) of the *Pro JavaFX book* for allowing me to tech review chapters in the first edition. Thanks also to Stephen Chin and Keith Combs for heading up the awesome JavaFX User Group. I want to also thank Rajmahendra Hegde for helping me with the eWidgetFX open source framework effort. Thanks to Hendrik Ebbers for always wanting to team up with me on talks and fun projects. Most importantly, thanks to all the past JavaFX 1.x book authors that helped inspire me. Thanks go out to my employer, TurretTech 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.x and 8 were being released: Nandini Ramani, Jonathan Giles, Jasper Potts, Richard Bair, Angela Caicedo, Stuart Marks, John Yoon, David Grieve, Michael Heinrichs, David DeHaven, Nicolas Lorain, Kevin Rushforth, Sheila Cepero, Gail Chappell, Cindy Castillo, Scott Hommel, Joni Gordon, Alexander Kouznetsov, Irina Fedortsova, Dmitry Kostovarov, Alla Redko, 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)*

—Carl Dea

I would like to thank my wife Kathy and three daughters, Jennifer, Andrea, and Rebekah Heckler, for their love and support. You inspire me to learn, grow, and reach ever higher; thank you for being your amazing selves.

I would also like to thank Apress and its team of editors, tech reviewers, and all who had a hand in making this book possible. You are all simply outstanding, and your dedication has made this book far more than the “sum of its parts.” Thank you.

—Mark Heckler

First of all, I would like to thank the JavaFX community and especially the guys with whom I have the pleasure to share the writing of this book, starting with Carl for his kind invitation.

I’d like to thank also the Arduino community, starting with its founders, and the Leap Motion company for letting me in the developer program.

Finally, this couldn’t have been done with the great support of my (very understanding) family.

—José Pereda

I am very pleased to thank my family, especially my wife Zulma, for their patience and support throughout this effort, along with so many others.

I’d like to thank Sean Henely for helping shake off my 3D mathematics rust and Jason Arcido for helping flesh out my 3D example designs.

I also thank Geertjan Wielenga for plugging me into the greater Java community.

Finally, thank you, Carl for this awesome opportunity!

—Sean M. Phillips