3.6 (Optional) GUI and Graphics Case Study: A Simple GUI

This case study is designed for those who want to begin learning Java's powerful capabilities for creating graphical user interfaces (GUIs) and graphics early in the book, before our deeper discussions of these topics in Chapters 12, 13 and 22. We feature **JavaFX**—Java's GUI, graphics and multimedia technology of the future. The Swing version of this case study is still available on the book's Companion Website. The GUI and Graphics Case Study sections are summarized in Fig. 3.11. Each section introduces new concepts, provides examples with screen captures that show sample interactions and is followed immediately by one or more exercises in which you'll use the techniques you learned in that section.

Section or Exercise	What you'll do
Section 3.6: A Simple GUI	Display text and an image.
Section 4.15: Event Handling; Drawing Lines	In response to a `Button` click, draw lines using JavaFX graphics capabilities.
Section 5.11: Drawing	Draw filled shapes in multiple colors.

12 JavaFX Graphical User Interfaces: Part 1

Objectives

In this chapter you'll:

- Build JavaFX GUIs and handle events generated by user interactions with them.
- Understand the structure of a JavaFX app window.
- Use JavaFX Scene Builder to create FXML files that describe JavaFX scenes containing `Label`s, `ImageView`s, `TextField`s, `Slider`s and `Button`s without writing any code.
- Arrange GUI components using the `VBox` and `GridPane` layout containers.
- Use a controller class to define event handlers for JavaFX FXML GUI.
- Build two JavaFX apps.

Outline

12.1 Introduction

A **graphical user interface** (**GUI**) presents a user-friendly mechanism for interacting with an app. A GUI (pronounced “GOO-ee”) gives an app a distinctive “look-and-feel.” GUIs are built from **GUI components**—also called *controls* or *widgets* (short for window gadgets). A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition.

Look-and-Feel Observation 12.1

Providing different apps with consistent, intuitive user-interface components gives users a sense of familiarity with a new app, so that they can learn it more quickly and use it more productively.

History of GUI in Java

Java’s original GUI library was the Abstract Window Toolkit (AWT). Swing was added to the platform in Java SE 1.2. Until recently, Swing was the primary Java GUI technology. Swing will remain part of Java and is still widely used. We discuss Swing in online Chapters 26 and 35.

JavaFX is Java's GUI, graphics and multimedia API of the future. Sun Microsystems (acquired by Oracle in 2010) announced JavaFX in 2007 as a competitor to Adobe Flash and Microsoft Silverlight. JavaFX 1.0 was released in 2008. Prior to version 2.0, developers wrote JavaFX apps in JavaFX Script, which compiled to Java bytecode, allowing JavaFX apps to run on the Java Virtual Machine. Starting with version 2.0 in 2011, JavaFX was reimplemented as Java libraries that could be used directly in Java apps. Some of the benefits of JavaFX over Swing include:

- JavaFX is easier to use—it provides one API for client functionality, including GUI, graphics and multimedia (images, animation, audio and video). Swing is only for GUIs, so you need to use other APIs for graphics and multimedia apps.
- With Swing, many IDEs provided GUI design tools for dragging and dropping components onto a layout; however, each IDE produced different code (such as different variable and method names). JavaFX Scene Builder (Section 12.2) can be used standalone or integrated with many IDEs and it produces the same code regardless of the IDE.
- Though Swing components could be customized, JavaFX gives you complete control over a JavaFX GUI's look-and-feel (Chapter 13) via Cascading Style Sheets (CSS)—the same technology used to style web pages.
- JavaFX has better threading support, which is important for getting the best application performance on today's multi-core systems.
- JavaFX uses the GPU (graphics processing unit) for hardware-accelerated rendering.
- JavaFX supports transformations for repositioning and reorienting JavaFX components, and animations for changing the properties of JavaFX components over time. These can be used to make apps more intuitive and

easier to use.

- JavaFX provides multiple upgrade paths for enhancing existing GUIs—Swing GUI capabilities may be embedded into JavaFX apps via class `SwingNode` and JavaFX capabilities may be embedded into Swing apps via class `JFXPanel`.

This chapter introduces JavaFX GUI basics—we present a more detailed treatment of Java FX GUI in the next chapter. Chapter 22 discusses graphics and multimedia. We placed the Java How to Program, 10/e Swing and Java 2D chapters on the book's Companion Website—see the inside front cover for Companion Website access instructions.

12.2 JavaFX Scene Builder

Most Java textbooks that introduce GUI programming provide hand-coded GUIs—that is, the authors build the GUIs from scratch in Java code, rather than using a visual GUI design tool. This is due to the fractured Java IDE market—there are many Java IDEs, so authors can’t depend on any one IDE being used, and each generates different code.

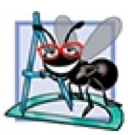
JavaFX is organized differently. The **Scene Builder** tool is a standalone JavaFX GUI visual layout tool that can also be used with various IDEs, including the most popular ones—Eclipse, IntelliJ IDEA and NetBeans. You can download Scene Builder at:

```
http://gluonhq.com/labs/scene-builder/
```

JavaFX Scene Builder enables you to create GUIs by dragging and dropping GUI components from Scene Builder’s library onto a design area, then modifying and styling the GUI—all without writing any code. JavaFX Scene Builder’s live editing and preview features allow you to view your GUI as you create and modify it, without compiling and running the app. You can use **Cascading Style Sheets (CSS)** to change the entire look-and-feel of your GUI—a concept sometimes called **skinning**. In Chapter 22, we’ll introduce styling with CSS.

FXML (FX Markup Language)

As you create and modify a GUI, JavaFX Scene Builder generates **FXML (FX Markup Language)**—an XML vocabulary for defining and arranging JavaFX GUI controls without writing any Java code. XML (eXtensible Markup Language) is a widely used language for describing things—it's readable both by computers and by humans. In JavaFX, FXML concisely describes GUI, graphics and multimedia elements. *You do not need to know FXML or XML to study this chapter.* As you'll see in Section 12.4, JavaFX Scene Builder hides the FXML details from you, so you can focus on defining *what* the GUI should contain without specifying *how* to generate it—this is an example of *declarative programming*.

 Software Engineering Observation 12.1

The FXML code is separate from the program logic that's defined in Java source code—this separation of the interface (the GUI) from the implementation (the Java code) makes it easier to debug, modify and maintain JavaFX GUI apps.

12.3 JavaFX App Window Structure

A JavaFX app window consists of several parts (Fig. 12.1).

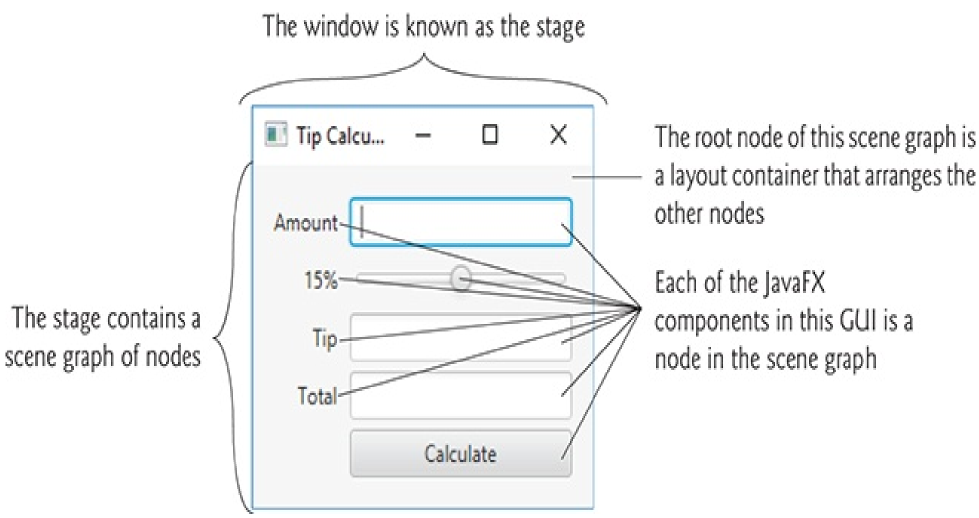


Fig. 12.1

JavaFX app window parts.

Description

Controls

Controls are GUI components, such as `Label`s that display text, `TextField`s that enable a program to receive user input, `Button`s that users click to initiate actions, and more.

Stage

The window in which a JavaFX app's GUI is displayed is known as the **stage** and is an instance of class `Stage` (package `javafx.stage`).

Scene

The stage contains one active **scene** that defines the GUI as a **scene graph**—a tree data structure of an app's visual elements, such as GUI controls, shapes, images, video, text and more (trees are discussed in Section 21.7). The scene is an instance of class `Scene` (package `javafx.scene`).

Nodes

Each visual element in the scene graph is a **node**—an instance of a subclass of `Node` (package `javafx.scene`), which defines common attributes and behaviors for all nodes. With the exception of the first node in the scene graph—the **root node**—each node in the scene graph has one parent. Nodes can have transforms (e.g., moving, rotating and scaling), opacity (whether a node is transparent, partially transparent or opaque), effects (e.g., drop shadows, blurs, reflection and lighting) and more that we'll introduce in Chapter 22.

Layout Containers

Nodes that have children are typically **layout containers** that arrange their child nodes in the scene. You'll use two layout containers (`VBox` and `GridPane`) in this chapter and learn several more in Chapters 13–22. The nodes arranged in a layout container are a combination of controls and, in more complex GUIs, possibly other layout containers.

Event Handler and Controller Class

When the user interacts with a control, such as clicking a `Button` or typing text into a `TextField`, the control generates an event. Programs can respond to these events—known as event handling—to specify what should happen when each user interaction occurs. An **event handler** is a method that responds to a user interaction. An FXML GUI's event handlers are defined in a so-called **controller class** (as you'll see in Section 12.5.5).

12.4 Welcome App—Displaying Text and an Image

In this section, *without writing any code*, you'll build a GUI that displays text in a `Label` and an image in an `ImageView` (Fig. 12.2). You'll use visual-programming techniques to *drag-and-drop* JavaFX components onto Scene Builder's content panel—the design area. Next, you'll use Scene Builder's **Inspector** to configure options, such as the `Labels`'s text and font size, and the `ImageView`'s image. Finally, you'll view the completed GUI using Scene Builder's **Show Preview in Window** option. In Section 12.5's **Tip Calculator** app, we'll discuss the Java code necessary to load and display an FXML GUI. Then, in Exercise 12.3, you'll create the Java application that displays the **Welcome** GUI you build in this section.

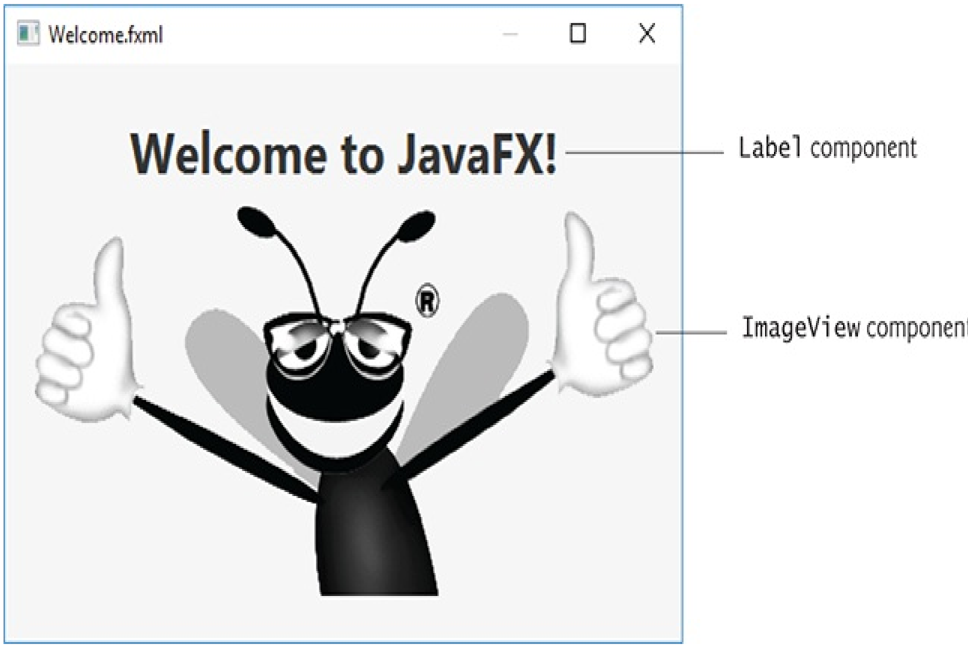


Fig. 12.2

Final **Welcome** GUI in a preview window on Microsoft Windows 10.

Description

12.4.1 Opening Scene Builder and Creating the File `Welcome.fxml`

Open Scene Builder so that you can create the FXML file that defines the GUI. The window initially appears as shown in Fig. 12.3. **Untitled** at the top of the window indicates that

Scene Builder has created a new FXML file that you have not yet saved.[1] Select **File** > **Save** to display the **Save As** dialog, then select a location in which to store the file, name the file `Welcome.fxml` and click the **Save** button.

1. We show the Scene Builder screen captures on Microsoft Windows 10, but Scene Builder is nearly identical on Windows, macOS and Linux. The key difference is that the menu bar on macOS is at the top of the screen, whereas the menu bar is part of the window on Windows and Linux.

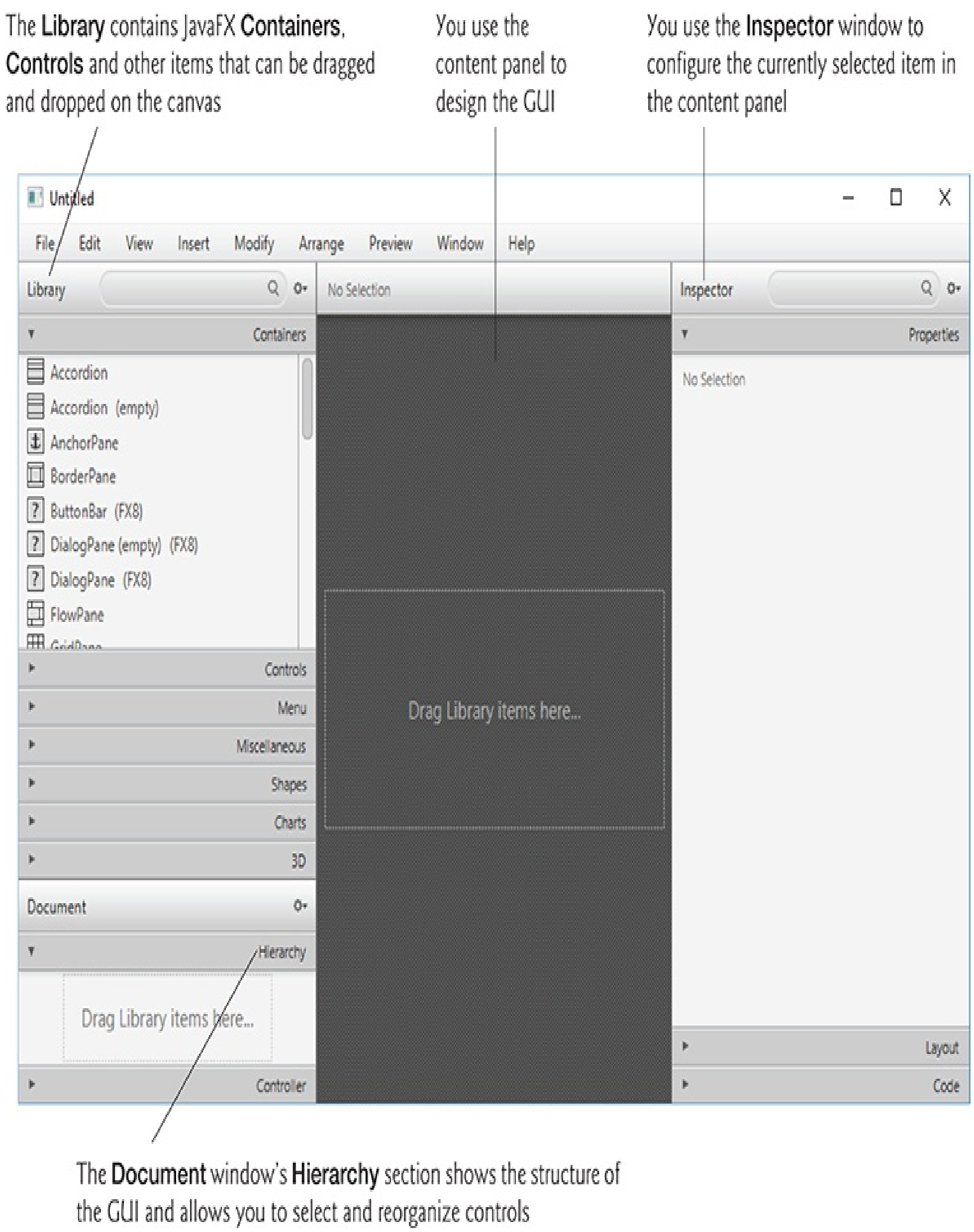


Fig. 12.3

JavaFX Scene Builder when you first open it.

Description

12.4.2 Adding an Image to the Folder Containing `Welcome.fxml`

The image you'll use for this app (`bug.png`) is located in the `images` subfolder of this chapter's examples folder. To make it easy to find the image when you're ready to add it to the app, locate the `images` folder on your file system, then copy `bug.png` into the folder where you saved `Welcome.fxml`.

12.4.3 Creating a `VBox` Layout Container

For this app, you'll place a `Label` and an `ImageView` in a `VBox` **layout container** (package `javafx.scene.layout`), which will be the scene graph's root node. Layout containers help you arrange and size GUI components. A `VBox` arranges its nodes *vertically* from top to bottom. We discuss the `GridPane` layout container in Section 12.5 and several others in Chapter 13. To add a `VBox` to Scene Builder's content panel so you can begin designing the GUI, double-click **VBox** in the **Library** window's **Containers** section. (You also can drag-and-drop a `VBox` from the **Containers** section onto Scene Builder's content panel.)

12.4.4 Configuring the `VBox` Layout Container

You'll now specify the `VBox`'s alignment, initial size and padding.

Specifying the `VBox`'s Alignment

A `VBox`'s **alignment** determines the layout positioning of the `VBox`'s children. In this app, we'd like each child node (the `Label` and the `ImageView`) to be centered horizontally in the scene, and we'd like both children to be centered vertically, so that there is an equal amount of space above the `Label` and below the `ImageView`. To accomplish this:

1. Select the `VBox` in Scene Builder's content panel by clicking it. Scene Builder displays many `VBox` properties in the Scene Builder **Inspector**'s **Properties** section.
2. Click the **Alignment** property's drop-down list and notice the variety of potential alignment values you can use. Click `CENTER` to set the

Alignment.

Each property value you specify for a JavaFX object is used to set one of that object’s instance variables when JavaFX creates the object at runtime.

Specifying the `VBox`’s Preferred Size

The **preferred size** (width and height) of the scene graph’s root node is used by the scene to determine its window size when the app begins executing. To set the preferred size:

1. Select the `VBox`.
2. Expand the **Inspector**’s **Layout** section by clicking the right arrow (▶) next to **Layout**. The section expands and the right arrow changes to a down arrow. Clicking the arrow again would collapse the section.
3. Click the **Pref Width** property’s text field, type `450` and press *Enter* to change the preferred width.
4. Click the **Pref Height** property’s text field, type `300` and press *Enter* to change the preferred height.

12.4.5 Adding and Configuring a `Label`

Next, you’ll create the `Label` that displays `"Welcome to JavaFX!"`.

Adding a `Label` to the `VBox`

Expand the Scene Builder **Library** window's **Controls** section by clicking the right arrow () next to **Controls**, then drag-and-drop a **Label** from the **Controls** section onto the `VBox` in Scene Builder's content panel. (You also can double-click **Label** in the **Containers** section to add the `Label`.) Scene Builder automatically centers the `Label` object horizontally and vertically in the `VBox`, based on the `VBox`'s **Alignment** property.

Changing the `Label`'s Text

You can set a `Label`'s text either by double clicking it and typing the new text, or by selecting the `Label` and setting its **Text** property in the **Inspector**'s **Properties** section. Set the `Label`'s text to `"Welcome to JavaFX!"`.

Changing the `Label`'s Font

For this app, we set the `Label` to display in a large bold font. To do so, select the `Label`, then in the **Inspector**'s **Properties** section, click the value to the right of the **Font** property. In the window that appears, set the **Style** property to `Bold` and the **Size** property to `30`. The design should now appear as shown in Fig. 12.4.

Fig. 12.4

Welcome GUI's design after adding and configuring a `Label`.

Description

12.4.6 Adding and Configuring an `ImageView`

Finally, you'll add the `ImageView` that displays `bug.png`.

Adding an `ImageView` to the `VBox`

Drag and drop an **ImageView** from the **Library** window's **Controls** section to just below the `Label`, as shown in Fig. 12.5. You can also double-click **ImageView** in the **Library** window, in which case Scene Builder automatically places the new `ImageView` object below the `Label`. You can reorder a `VBox`'s controls by dragging them in the `VBox` or in the **Document** window's **Hierarchy** section (Fig. 12.3). Scene Builder automatically centers the `ImageView` horizontally in the `VBox`. Also notice that the `Label` and `ImageView` are centered vertically such that the same amount of space appears above the `Label` and below the `ImageView`.

Setting the `ImageView`'s Image

Next you'll set the image to display:

1. Select the `ImageView`, then in the **Inspector**'s **Properties** section click the ellipsis (**…**) button to the right of the **Image** property. By default, Scene Builder opens a dialog showing the folder in which the FXML file is saved. This is where you placed the image file `bug.png` in Section 12.4.2.
2. Select the image file, then click **Open**. Scene Builder displays the image and resizes the `ImageView` to match the image's aspect ratio—that is, the ratio of the image's width to its height.

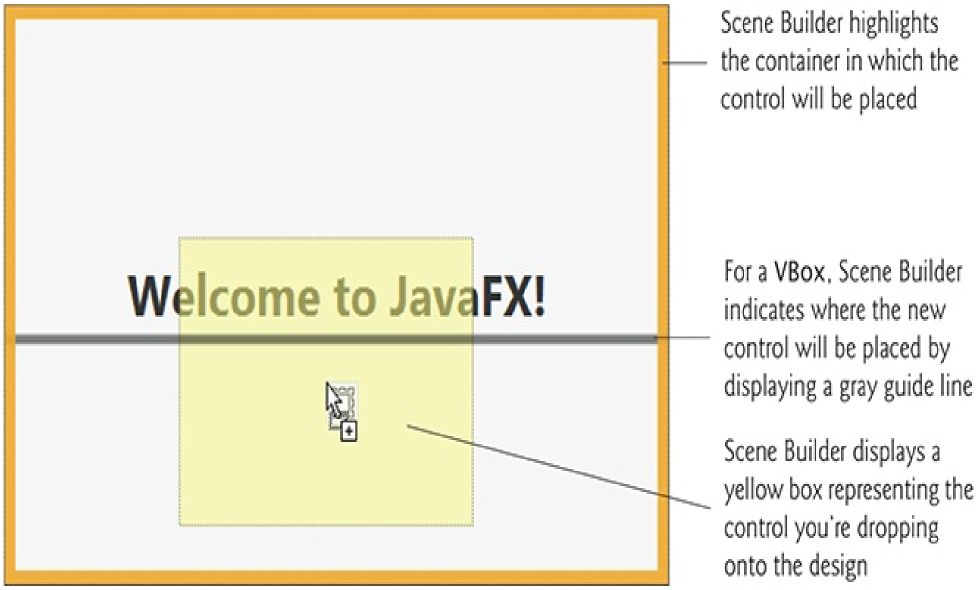


Fig. 12.5

Dragging and dropping the `ImageView` below the `Label`.

Description

Changing the `ImageView`'s Size

We'd like to display the image at its original size. If you reset the `ImageView`'s default **Fit Width** and **Fit Height** property values—which Scene Builder set when you added the `ImageView` to the design—Scene Builder will resize the `ImageView` to the image's exact dimensions. To reset these properties:

1. Expand the **Inspector**’s **Layout** section.
2. Hover the mouse over the **Fit Width** property’s value. This displays the button  to the right property’s value. Click the button and select **Reset to Default** to reset the value. This technique can be used with any property value to reset its default.
3. Repeat *Step 2* to reset the **Fit Height** property’s value.

You’ve now completed the GUI. Scene Builder’s content panel should now appear as shown in Fig. 12.6. Save the FXML file by selecting **File** > **Save**.

Fig. 12.6

Completed **Welcome** GUI in Scene Builder’s content panel.

Description

12.4.7 Previewing the `Welcome` GUI

You can preview what the design will look like in a running application's window. To do so, select **Preview > Show Preview in Window**, which displays the window in Fig. 12.7.

Fig. 12.7

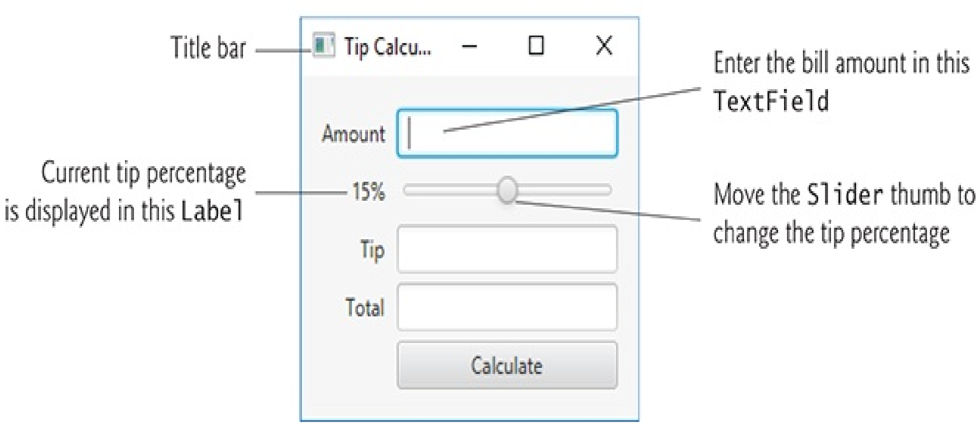
Previewing the **Welcome** GUI on Microsoft Windows 10—only the window borders will differ on Linux, macOS and earlier Windows versions.

Description

12.5 Tip Calculator App—Introduction to Event Handling

The **Tip Calculator** app (Fig. 12.8(a)) calculates and displays a restaurant bill tip and total. By default, the app calculates the total with a 15% tip. You can specify a tip percentage from 0% to 30% by moving the `Slider` *thumb*—this updates the tip percentage (Fig. 12.8(b)) and (c)). In this section, you'll build a **Tip Calculator** app using several JavaFX components and learn how to respond to user interactions with the GUI.

a) Initial **Tip Calculator** GUI



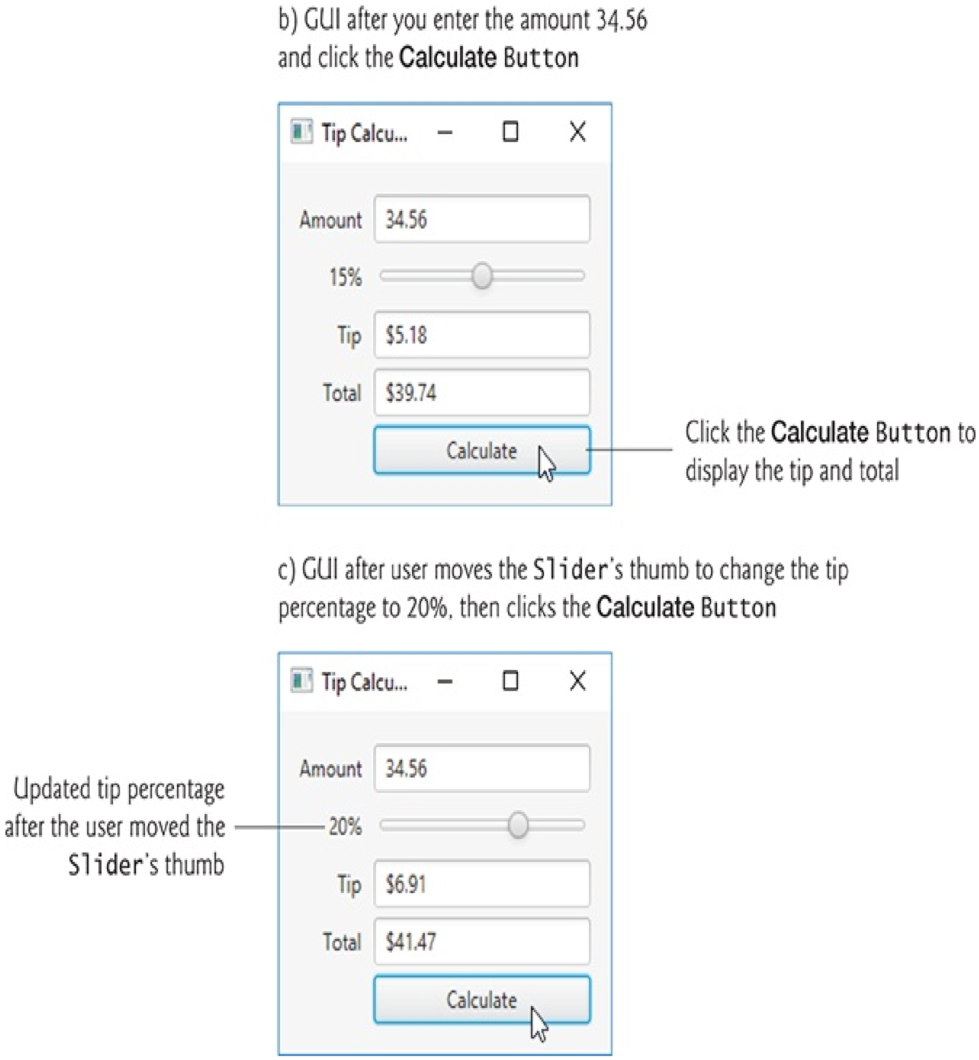


Fig. 12.8

Entering the bill amount and calculating the tip.

Description

You'll begin by test-driving the app, using it to calculate 15% and 20% tips. Then we'll overview the technologies you'll use to create the app. You'll build the app's GUI using the Scene Builder. Finally, we'll present the complete Java code for the app and do a detailed code walkthrough.

12.5.1 Test-Driving the Tip Calculator App

Compile and run the app located in the `TipCalculator` folder with this chapter's examples. The class containing the `main` method is named `TipCalculator`.

Entering a Bill Total

Using your keyboard, enter `34.56`, then press the **Calculate** `Button`. The **Tip** and **Total** `TextField`s show the tip amount and the total bill for a 15% tip (Fig. 12.8(b)).

Selecting a Custom Tip Percentage

Use the `Slider` to specify a *custom* tip percentage. Drag the `Slider`'s *thumb* until the percentage reads **20%** (Fig. 12.8(c)), then press the **Calculate** `Button` to display the updated tip and total. As you drag the thumb, the tip percentage in the `Label` to the `Slider`'s left updates continuously. By default, the `Slider` allows you to select values from 0.0 to 100.0, but in this app we'll restrict the

`Slider` to selecting whole numbers from 0 to 30.

12.5.2 Technologies Overview

This section introduces the technologies you'll use to build the **Tip Calculator** app.

Class `Application`

The class responsible for launching a JavaFX app is a subclass of `Application` (package `javafx.application`). When the subclass's `main` method is called:

1. Method `main` calls class `Application`'s `static launch` method to begin executing the app.
2. The `launch` method, in turn, causes the JavaFX runtime to create an object of the `Application` subclass and call its `start` method.
3. The `Application` subclass's `start` method creates the GUI, attaches it to a `Scene` and places it on the `Stage` that `start` receives as an argument.

Arranging JavaFX Components with a `GridPane`

Recall that layout containers arrange JavaFX components in a `Scene`. A `GridPane` (package `javafx.scene.layout`) arranges JavaFX components into *columns* and *rows* in a rectangular grid.

This app uses a `GridPane` (Fig. 12.9) to arrange views into two columns and five rows. Each cell in a `GridPane` can be empty or can hold one or more JavaFX components, including layout containers that arrange other controls. Each component in a `GridPane` can span *multiple* columns or rows, though we did not use that capability in this GUI. When you drag a `GridPane` onto Scene Builder's content panel, Scene Builder creates the `GridPane` with two columns and three rows by default. You can add and remove columns and rows as necessary. We'll discuss other `GridPane` features as we present the GUI-building steps. To learn more about class `GridPane`, visit:

```
https://docs.oracle.com/javase/8/javafx/api/javafx/sc
```

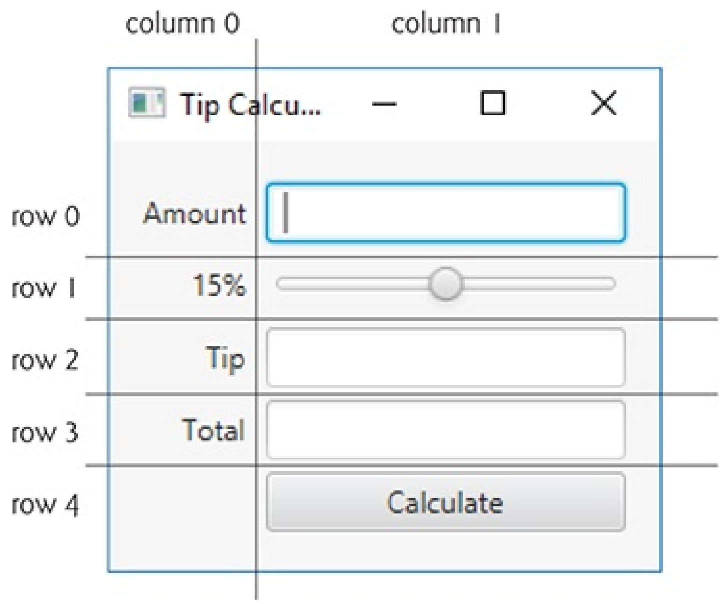


Fig. 12.9

Tip Calculator GUI's `GridPane` labeled by its rows and columns.

Description

Creating and Customizing the GUI with Scene Builder

You'll create `Label`s, `TextField`s, a `Slider` and a `Button` by dragging them onto Scene Builder's content panel, then customize them using the **Inspector** window.

- A `TextField` (package `javafx.scene.control`) can accept text input from the user or display text. You'll use one editable `TextField` to input the bill amount from the user and two *uneditable* `TextField`s to display the tip and total amounts.

- A `Slider` (package `javafx.scene.control`) represents a value in the range 0.0–100.0 by default and allows the user to select a number in that range by moving the `Slider`'s thumb. You'll customize the `Slider` so the user can choose a custom tip percentage *only* from the more limited integer range 0 to 30.
- A `Button` (package `javafx.scene.control`) allows the user to initiate an action—in this app, pressing the **Calculate** `Button` calculates and displays the tip and total amounts.

Formatting Numbers as Locale-Specific Currency and Percentage Strings

You'll use class `NumberFormat` (package `java.text`) to create *locale-specific* currency and percentage strings—an important part of *internationalization*.[2]

2. Recall that the new JavaMoney API (`http://javamoney.github.io`) was developed to meet the challenges of handling currencies, monetary amounts, conversions, rounding and formatting. At the time of this writing, it was not yet incorporated into the JDK.

Event Handling

Normally, a user interacts with an app's GUI to indicate the tasks that the app should perform. For example, when you write an e-mail, clicking the e-mail app's **Send** button tells the app to send the e-mail to the specified e-mail addresses.

GUIs are **event driven**. When the user interacts with a GUI component, the interaction—known as an **event**—drives the program to perform a task. Some common user interactions that cause an app to perform a task include *clicking* a button, *typing* in a text field, *selecting* an item from a menu, *closing* a window and *moving* the mouse. The code that performs a task in response to an event is called an **event handler**, and the process of responding to events is known as **event handling**.

Before an app can respond to an event for a particular control, you must:

1. Define an event handler that implements an appropriate interface—known as an **event-listener interface**.
2. Indicate that an object of that class should be notified when the event occurs—known as **registering the event handler**.

In this app, you'll respond to two events—when the user moves the `Slider`'s thumb, the app will update the `Label` that displays the current tip percentage, and when the user clicks the **Calculate** `Button`, the app will calculate and display the tip and total bill amount.

You'll see that for certain events—such as when the user clicks a `Button`—you can link a control to its event-handling method by using the **Code** section of Scene Builder's **Inspector** window. In this case, the event-listener interface is implemented for you to call the method that you specify. For events that occur when the value of a control's property changes—such as when the user moves a `Slider`'s thumb to change the `Slider`'s value—you'll see that you must create the event handler entirely in code.

Implementing Interface `ChangeListener` for Handling `Slider` Thumb Position Changes

You'll implement interface `ChangeListener` (package `javafx.beans.value`) to respond when the user moves the `Slider`'s thumb. In particular, you'll use the interface's `changed` method to display the updated tip percentage as the user moves the `Slider`'s thumb.

Model-View-Controller (MVC) Architecture

JavaFX applications in which the GUI is implemented as FXML adhere to the **Model-View-Controller (MVC) design pattern**, which separates an app's data (contained in the **model**) from the app's GUI (the **view**) and the app's processing logic (the **controller**).

The controller implements logic for processing user inputs. The model contains application data, and the view presents the data stored in the model. When a user provides some input, the controller modifies the model with the given input. In the **Tip Calculator**, the model is the bill amount, the tip and the total. When the model changes, the controller updates the view to present the changed data.

In a JavaFX FXML app, a **controller class** defines instance variables for interacting with controls programmatically, as well as event-handling methods that respond to the user's interactions. The controller class may also declare additional instance variables, `static` variables and methods that support the app's operation. In a simple app like the **Tip Calculator**, the model and controller are often combined into a single class, as we'll do in this example.

`FXMLLoader` Class

When a JavaFX FXML app begins executing, class `FXMLLoader`'s `static` method `load` is used to load the FXML file that represents the app's GUI. This method:

- Creates the GUI's scene graph—containing the GUI's layouts and controls—and returns a `Parent` (package `javafx.scene`) reference to the scene graph's root node.
- Initializes the controller's instance variables for the components that are manipulated programmatically.
- Creates and registers the event handlers for any events specified in the FXML.

We'll discuss these steps in more detail in Sections 12.5.4–12.5.5.

12.5.3 Building the App’s GUI

In this section, we’ll show the precise steps for creating the **Tip Calculator**’s GUI. The GUI will not look like the one shown in Fig. 12.8 until you’ve completed the steps.

fx:id Property Values for This App’s Controls

If the controller class will manipulate a control or layout programmatically (as we’ll do with one `Label`, all the `TextField`s and the `Slider`), you must provide a name for that control or layout. In Section 12.5.4, you’ll learn how to declare Java variables for each such component in the FXML, and we’ll discuss how those variables are initialized for you. Each object’s name is specified via its **fx:id property**. You can set this property’s value by selecting a component in your scene, then expanding the **Inspector** window’s **Code** section —the **fx:id** property appears at the top of the **Code** section. Figure 12.10 shows the **fx:id** properties of the **Tip Calculator**’s programmatically manipulated controls. For clarity, our naming convention is to use the control’s class name in the **fx:id** property.

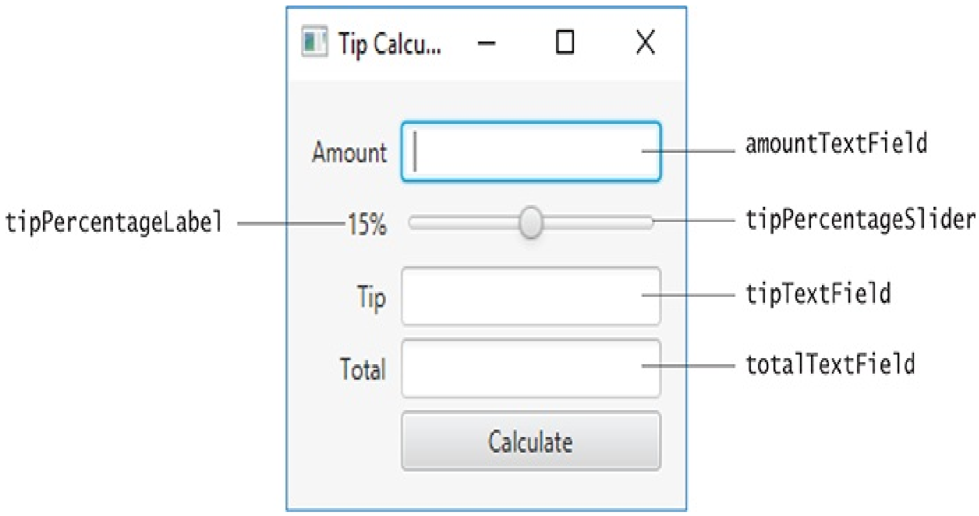


Fig. 12.10

Tip Calculator’s programmatically manipulated controls labeled with their **fx:id**s.

Description

Creating the `TipCalculator.fxml` File

As you did in Section 12.4.1, open Scene Builder to create a new FXML file. Then, select **File** > **Save** to display the **Save As** dialog, specify the location in which to store the file, name the file `TipCalculator.fxml` and click the **Save** button.

Step 1: Adding a `GridPane`

Drag a `GridPane` from the **Library** window's **Containers** section onto Scene Builder's content panel. By default, the `GridPane` contains two columns and three rows as shown in Fig. 12.11.

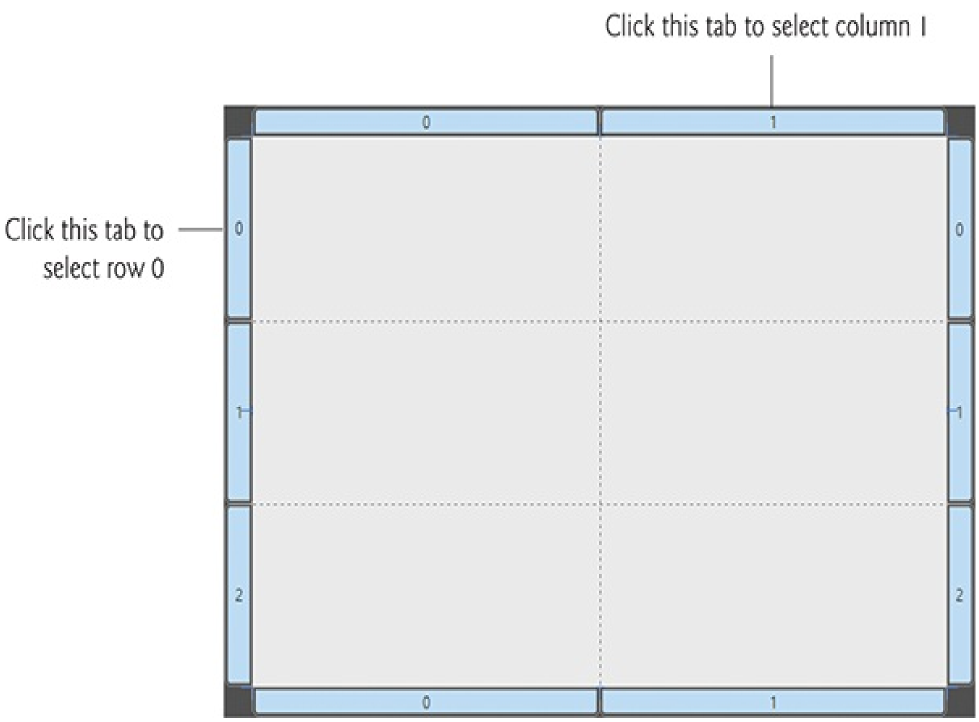


Fig. 12.11

`GridPane` with two columns (0 and 1) and three rows (0, 1 and 2).

Step 2: Adding Rows to the

GridPane

Recall that the GUI in Fig. 12.9 has two columns and five rows. Here you'll add two more rows. To add a row above or below an existing row:

1. Right click any row's row number tab and select either **Grid Pane > Add Row Above** or **Grid Pane > Add Row Below**.
2. Repeat this process to add another row.

After adding two rows, the `GridPane` should appear as shown in Fig. 12.12. You can use similar steps to add columns. You can delete a row or column by right clicking the tab containing its row or column number and selecting **Delete**.

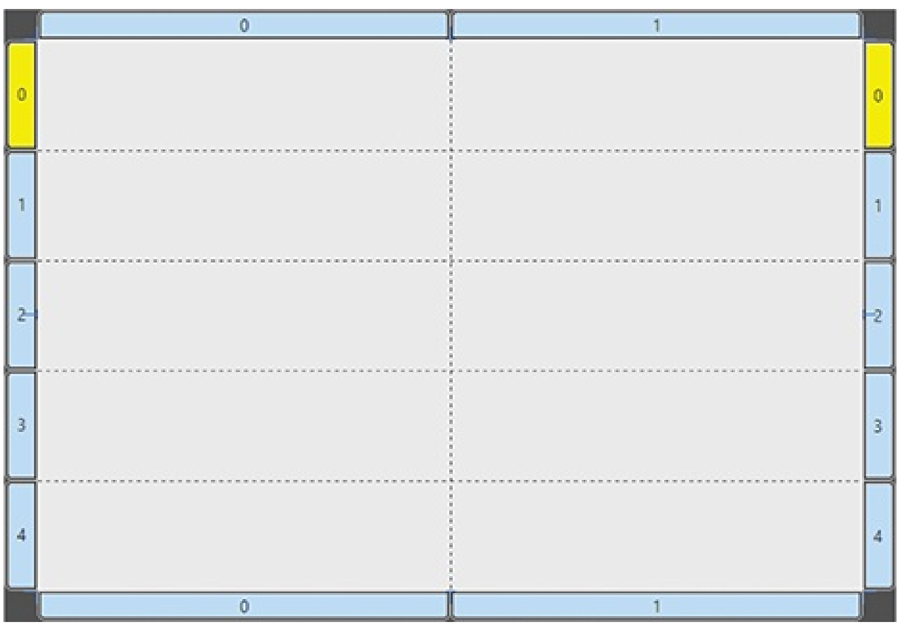


Fig. 12.12

`GridPane` after adding two more rows.

Step 3: Adding the Controls to the `GridPane`

You’ll now add the controls in Fig. 12.9 to the `GridPane`. For each control that has an **fx:id** in Fig. 12.10, when you drag the control onto the `GridPane`, set the control’s **fx:id** property in the **Inspector** window’s **Code** section. Perform the following steps:

1. ***Adding the*** `Label`***s.*** Drag `Label`s from the **Library** window’s **Controls** section into the first four rows of column 0 (the `GridPane`’s left column). As you add each `Label`, set its text as shown Fig. 12.9.
2. ***Adding the*** `TextField`***s.*** Drag `TextField`s from the **Library** window’s **Controls** section into rows 0, 2 and 3 of column 1 (the `GridPane`’s right column).
3. ***Adding a*** `Slider`**.** Drag a horizontal `Slider` from the **Library** window’s **Controls** section into row 1 of column 1.
4. ***Adding a*** `Button`**.** Drag a `Button` from the **Library** window’s **Controls** section into row 4 of column 1. Change the `Button`’s text to **Calculate**. You can set the `Button`’s text by double clicking it, or by selecting the `Button`, then setting its **Text** property in the **Inspector** window’s **Properties** section.

The `GridPane` should appear as shown in Fig. 12.13.

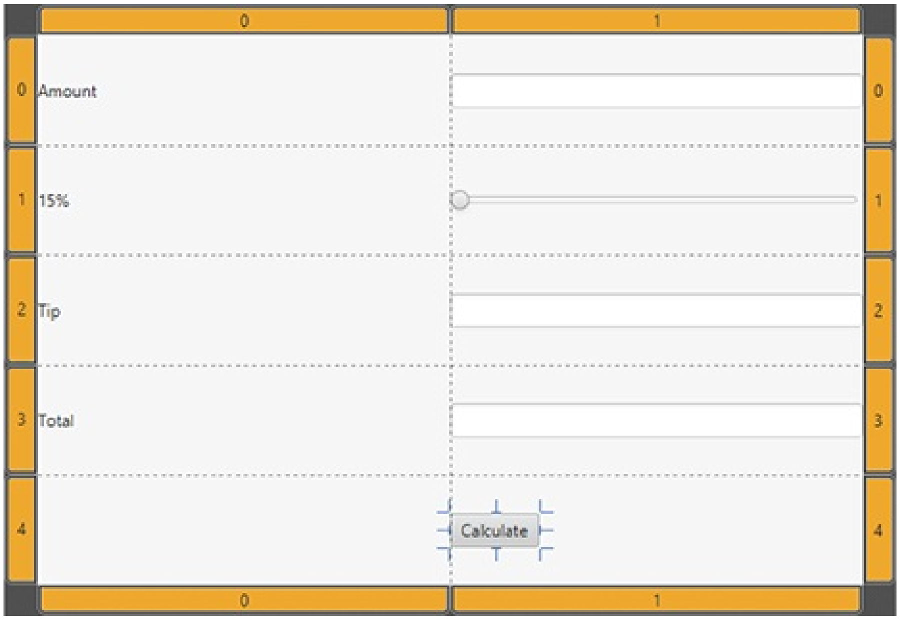


Fig. 12.13

`GridPane` filled with the **Tip Calculator**’s controls.

Description

Step 4: Sizing the `GridPane` to Fit Its Contents

When you begin designing a GUI by adding a layout, Scene Builder automatically sets the layout object’s **Pref Width** property to `600` and **Pref Height** property to `400`, which is much larger than this GUI’s final width and height. For this app, we’d like the layout’s size to be computed, based on the layout’s contents. To make this change:

1. First, select the `GridPane` by clicking inside the `GridPane`, but not on any of the controls you’ve placed into its columns and rows. Sometimes, it’s easier to select the `GridPane` node in the Scene Builder **Document** window’s **Hierarchy** section.
2. In the **Inspector**’s **Layout** section, reset the **Pref Width** and **Pref Height** property values to their defaults (as you did in Section 12.4.4). This sets both properties’ values to `USE_COMPUTED_SIZE`, so the layout calculates its own size.

The layout now appears as shown in Fig. 12.14.

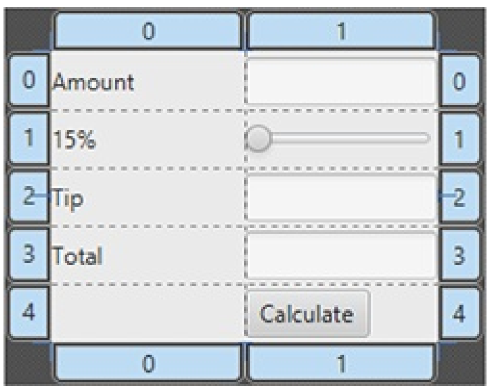


Fig. 12.14

`GridPane` sized to fit its contents.

Description

Step 5: Right-Aligning `GridPane` Column 0’s

Contents

A GridPane column's contents are left-aligned by default. To right-align the contents of column 0, select it by clicking the tab at the top or bottom of the column, then in the **Inspector**'s **Layout** section, set the **Halignment** (horizontal alignment) property to RIGHT.

Step 6: Sizing the GridPane Columns to Fit Their Contents

By default, Scene Builder sets each GridPane column's width to 100 pixels and each row's height to 30 pixels to ensure that you can easily drag controls into the GridPane's cells. In this app, we sized each column to fit its contents. To do so, select the column 0 by clicking the tab at the top or bottom of the column, then in the **Inspector**'s **Layout** section, reset the **Pref Width** property to its default size (that is, USE_COMPUTED_SIZE) to indicate that the column's width should be based on its widest child—the **Amount** Label in this case. Repeat this process for column 1. The GridPane should appear as shown in Fig. 12.15.

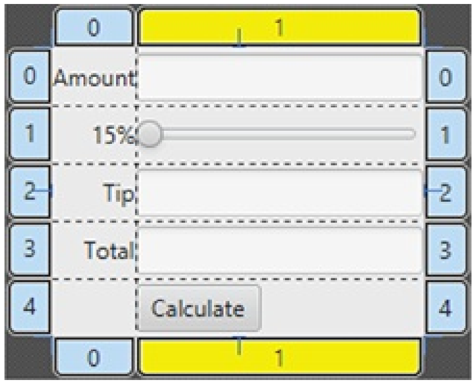


Fig. 12.15

`GridPane` with columns sized to fit their contents.

Description

Step 7: Sizing the `Button`

By default, Scene Builder sets a `Button`'s width based on its text. For this app, we chose to make the `Button` the same width as the other controls in the `GridPane`'s right column. To do so, select the `Button`, then in the **Inspector**'s **Layout** section, set the **Max Width** property to `MAX_VALUE`. This causes the `Button`'s width to grow to fill the column's width.

Previewing the GUI

Preview the GUI by selecting **Preview > Show Preview in Window**. As you can see in Fig. 12.16, there's no space between the `Label`s in the left column and the controls in the right column. In addition, there's no space around the `GridPane`, because by default the `Stage` is sized to fit the `Scene`'s contents. Thus, many of the controls touch the window's borders. You'll fix these issues in the next step.

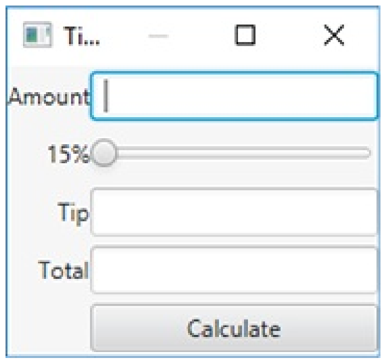


Fig. 12.16

`GridPane` with the `TextField`s and `Button` resized.

Description

Step 8: Configuring the `GridPane`'s Padding and Horizontal Gap Between Its Columns

The space between a node's contents and its top, right, bottom and left edges is known as the **padding**, which separates the contents from the node's edges. Since the `GridPane`'s size determines the `Stage`'s window size, the `GridPane`'s padding separates its children from the window's edges. To set the padding, select the `GridPane`, then in the **Inspector**'s **Layout** section, set the **Padding** property's four values (which represent the **TOP**, **RIGHT**, **BOTTOM** and **LEFT**) to `14`—the JavaFX recommended distance between a control's edge and the `Scene`'s edge.

You can specify the default amount of space between a `GridPane`'s columns and rows with its **Hgap** (horizontal gap) and **Vgap** (vertical gap) properties, respectively. Because Scene Builder sets each `GridPane` row's height to 30 pixels—which is greater than the heights of this app's controls—there's already some vertical space between the components. To specify the horizontal gap between the columns, select the `GridPane` in the **Document** window's **Hierarchy** section, then in the **Inspector**'s **Layout** section, set the **Hgap** property to `8`—the recommended distance between controls. If you'd like to precisely control the vertical space between components, you can reset each row's **Pref Height** to its default value, then set the `GridPane`'s **Vgap** property.

Step 9: Making the `tipTextField` and `totalTextField` Uneditable and Not

Focusable

The `tipTextField` and `totalTextField` are used in this app only to display results, not receive text input. For this reason, they should not be interactive. You can type in a `TextField` only if it's "in **focus**"—that is, it's the control that the user is interacting with. When you click an interactive control, it receives the focus. Similarly, when you press the *Tab* key, the focus transfers from the current focusable control to the next one—this occurs in the order the controls were added to the GUI. Interactive controls—such as `TextField`s, `Slider`s and `Button`s—are focusable by default. Non-interactive controls—like `Label`s—are not focusable.

In this app, the `tipTextField` and `totalTextField` are neither editable nor focusable. To make these changes, select both `TextField`s, then in the **Inspector**'s **Properties** section uncheck the **Editable** and **Focus Traversable** properties. To select multiple controls at once, you can click the first (in the **Document** window's **Hierarchy** section or in the content panel), then hold the *Shift* key and click each of the others.

Step 10: Setting the `Slider`'s Properties

To complete the GUI, you'll now configure the **Tip**

Calculator's `Slider`. By default, a `Slider`'s range is `0.0` to `100.0` and its initial value is 0.0. This app allows only integer tip percentages in the range 0 to 30 with a default of 15. To make these changes, select the `Slider`, then in the **Inspector**'s **Properties** section, set the `Slider`'s **Max** property to `30` and the **Value** property to `15`. We also set the **Block Increment** property to `5`—this is the amount by which the **Value** property increases or decreases when the user clicks between an end of the `Slider` and the `Slider`'s thumb. Save the FXML file by selecting **File** > **Save**.

Though we set the **Max**, **Value** and **Block Increment** properties to integer values, the `Slider` still produces floating-point values as the user moves its thumb. In the app's Java code, we'll restrict the `Slider`'s values to integers when we respond to its events.

Previewing the Final Layout

You've now completed the **Tip Calculator**'s design. Select **Preview** > **Show Preview in Window** to view the final GUI (Fig. 12.17). When we discuss the `TipCalculatorController` class in Section 12.5.5, we'll show how to specify the **Calculate** `Button`'s event handler in the FXML file.

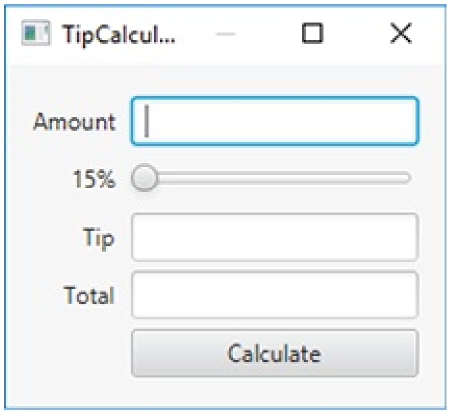


Fig. 12.17

Final GUI design previewed in Scene Builder.

Description

Specifying the Controller Class’s Name

As we mentioned in Section 12.5.2, in a JavaFX FXML app, the app’s controller class typically defines instance variables for interacting with controls programmatically, as well as event-handling methods. To ensure that an object of the controller class is created when the app loads the FXML file at runtime, you must specify the controller class’s name in the FXML file:

1. Expand Scene Builder **Document** window’s **Controller** section (located below the **Hierarchy** section in Fig. 12.3).
2. In the **Controller Class** field, type `TipCalculatorController`—by

convention, the controller class's name starts with the same name as the FXML file (`TipCalculator`) and ends with `Controller`.

Specifying the Calculate `Button`'s Event-Handler Method Name

You can specify in the FXML file the names of the methods that will be called to handle specific control's events. When you select a control, the **Inspector** window's **Code** section shows all the events for which you can specify event handlers in the FXML file. When the user clicks a `Button`, the method specified in the **On Action** field is called—this method is defined in the controller class you specify in Scene Builder's **Controller** window. Enter `calculateButtonPressed` in the **On Action** field.

Generating a Sample Controller Class

You can have Scene Builder generate the initial controller class containing the variables you'll use to interact with controls programmatically and the empty **Calculate** `Button` event handler. Scene Builder calls this the "controller skeleton." Select **View > Show Sample Controller Skeleton** to generate the skeleton (Fig. 12.18). As you can see, the sample class has the class name you specified, a variable for each control that has an **fx:id** and an empty **Calculate** `Button` event handler. We’ll discuss the `@FXML` annotation in Section 12.5.5 To use this skeleton to create your controller class, you can click the **Copy** button, then paste the contents into a file named `TipCalculatorController.java` in the same folder as the `TipCalculator.fxml` file you created in this section.

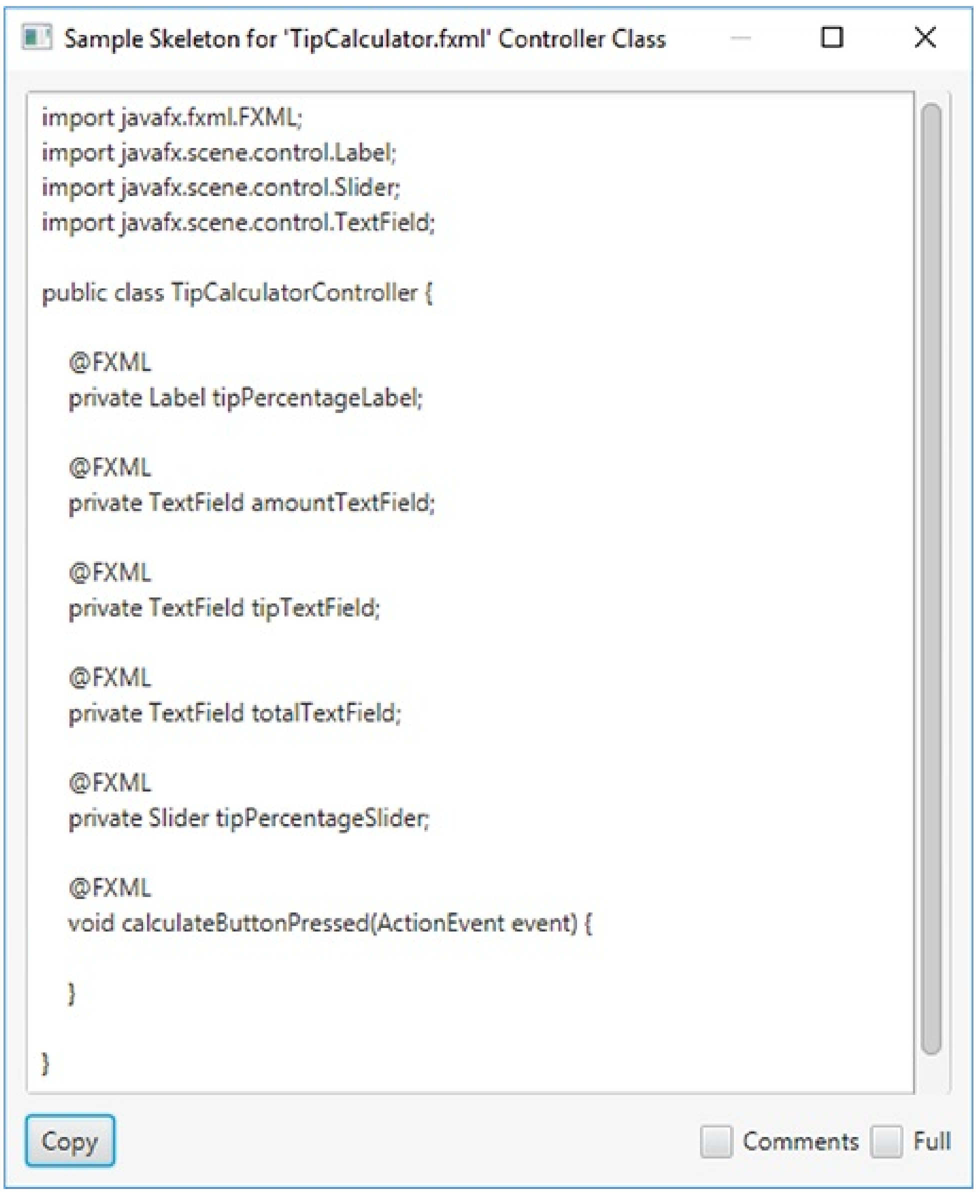


Fig. 12.18

Skeleton code generated by Scene Builder.

Description

12.5.4 `TipCalculator` Class

A simple JavaFX FXML-based app has two Java source-code files. For the **Tip Calculator** app these are:

- `TipCalculator.java`—This file contains the `TipCalculator` class (discussed in this section), which declares the `main` method that loads the FXML file to create the GUI and attaches the GUI to a `Scene` displayed on the app's `Stage`.
- `TipCalculatorController.java`—This file contains the `TipCalculatorController` class (discussed in Section 12.5.5), where you'll specify the `Slider` and `Button` controls' event handlers.

Figure 12.19 presents class `TipCalculator`. As we discussed in Section 12.5.2, the starting point for a JavaFX app is an `Application` subclass, so class `TipCalculator` extends `Application` (line 9). The `main` method calls class `Application`'s `static launch` method (line 23) to initialize the JavaFX runtime and to begin executing the app. This method causes the JavaFX runtime to create an object of the `TipCalculator` class and calls its `start` method (lines 10–19), passing the `Stage` object representing the window in which the app will be displayed. The JavaFX runtime creates the window.

```
// Fig. 12.19: TipCalculator.java
// Main app class that loads and displays the Ti
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class TipCalculator extends Application {
    @Override
    public void start(Stage stage) throws Excepti
        Parent root =
            FXMLLoader.load(getClass().getResource(

        Scene scene = new Scene(root); // attach s
        stage.setTitle("Tip Calculator"); // displ
        stage.setScene(scene); // attach scene to
        stage.show(); // display the stage
    }

    public static void main(String[] args) {
        // create a TipCalculator object and call
        launch(args);
    }
}
```

Fig. 12.19

Main app class that loads and displays the **Tip Calculator**’s GUI.

Overridden `Application` Method `start`

Method `start` (lines 11–19) creates the GUI, attaches it to a `Scene` and places it on the `Stage` that method `start` receives as an argument. Lines 12–13 use class `FXMLLoader`'s`static` method `load` to create the GUI's scene graph. This method:

- Returns a `Parent` (package `javafx.scene`) reference to the scene graph's root node—this is a reference to the GUI's `GridPane` in this app.
- Creates an object of the `TipCalculatorController` class that we specified in the FXML file.
- Initializes the controller's instance variables for the components that are manipulated programmatically.
- Attaches the event handlers specified in the FXML to the appropriate controls. This is known as registering the event handlers and enables the controls to call the corresponding methods when the user interacts with the app.

We discuss the initialization of the controller's instance variables and the registration of the event handlers in Section 12.5.5.

Creating the `Scene`

To display the GUI, you must attach it to a `Scene`, then attach the `Scene` to the `Stage` that method `start` receives as an argument. To attach the GUI to a `Scene`, line 15 creates a

`Scene`, passing `root` (the scene graph's root node) as an argument to the constructor. By default, the `Scene`'s size is determined by the size of the scene graph's root node. Overloaded versions of the `Scene` constructor allow you to specify the `Scene`'s size and fill (a color, gradient or image), which appears in the `Scene`'s background. Line 16 uses `Stage` method `setTitle` to specify the text that appears in the `Stage` window's title bar. Line 17 calls `Stage` method `setScene` to place the `Scene` onto the `Stage`. Finally, line 18 calls `Stage` method `show` to display the `Stage` window.

12.5.5 `TipCalculatorController` Class

Figure 12.20–12.23 present the `TipCalculatorController` class that responds to user interactions with the app's `Button` and `Slider`.

Class `TipCalculatorController`'s `import` Statements

Figure 12.20 shows class `TipCalculatorController`'s `import` statements.

```
1   // TipCalculatorController.java
2   // Controller that handles calculateButton and t
3   import java.math.BigDecimal;
4   import java.math.RoundingMode;
5   import java.text.NumberFormat;
6   import javafx.beans.value.ChangeListener;
7   import javafx.beans.value.ObservableValue;
8   import javafx.event.ActionEvent;
9   import javafx.fxml.FXML;
10  import javafx.scene.control.Label;
11  import javafx.scene.control.Slider;
12  import javafx.scene.control.TextField;
13
```

Fig. 12.20

`TipCalculatorController`'s `import` declarations.

The classes and interfaces used by class `TipCalculatorController` include:

- Class `BigDecimal` of package `java.math` (line 3) is used to perform precise monetary calculations. The `RoundingMode enum` of package `java.math` (line 4) is used to specify how `BigDecimal` values are rounded during calculations or when formatting floating-point numbers as `String`s.
- Class `NumberFormat` of package `java.text` (line 5) provides numeric formatting capabilities, such as locale-specific currency and percentage formats. For example, in the U.S. locale, the monetary value 34.95 is formatted as $34.95 and the percentage 15 is formatted as 15%. Class `NumberFormat` determines the locale of the system on which your app runs, then formats currency amounts and percentages accordingly.
- You implement interface `ChangeListener` of package

javafx.beans.value (line 6) to respond when the user moves the Slider's thumb. This interface's changed method receives an object that implements interface ObservableValue (line 7)—that is, a value that generates an event when it changes.

- A Button's event handler receives an ActionEvent object (line 8; package javafx.event) indicating which Button the user clicked. As you'll see in Chapter 13, many JavaFX controls support ActionEvents.
- The annotation FXML (line 9; package javafx.fxml) is used in a JavaFX controller class's code to mark instance variables that should refer to JavaFX components in the GUI's FXML file and methods that can respond to the events of JavaFX components in the GUI's FXML file.
- Package javafx.scene.control (lines 10–12) contains many JavaFX control classes, including Label, Slider and TextField.

TipCalculatorController's static Variables and Instance Variables

Lines 16—37 of Fig. 12.12 present class TipCalculatorController's static and instance variables. The NumberFormat objects (lines 16–19) are used to format currency values and percentages, respectively. NumberFormat method getCurrencyInstance returns a NumberFormat object that formats values as currency using the default locale for the system on which the app is running. Similarly, NumberFormat method getPercentInstance returns a NumberFormat object that formats values as percentages using the system's default locale. The BigDecimal object tipPercentage (line 21) stores the current tip percentage and is used in the tip calculation (Fig. 12.22) when the user clicks the **Calculate** `Button`.

```
public class TipCalculatorController {
   // formatters for currency and percentages
   private static final NumberFormat currency =
      NumberFormat.getCurrencyInstance();
   private static final NumberFormat percent =
      NumberFormat.getPercentInstance();

   private BigDecimal tipPercentage = new BigDec

   // GUI controls defined in FXML and used by t
   @FXML
   private TextField amountTextField;

   @FXML
   private Label tipPercentageLabel;

   @FXML
   private Slider tipPercentageSlider;

   @FXML
   private TextField tipTextField;

   @FXML
   private TextField totalTextField;

```

Fig. 12.21

`TipCalculatorController`’s `static` and instance variables.

`@FXML` Annotation

Recall from Section 12.5.3 that each control that this app manipulates in its Java source code needs an **fx:id**. Lines 24–37 (Fig. 12.21) declare the controller class's corresponding instance variables. The `@FXML` **annotation** that precedes each declaration (lines 24, 27, 30, 33 and 36) indicates that the variable name can be used in the FXML file that describes the app's GUI. The variable names that you specify in the controller class must precisely match the **fx:id** values you specified when building the GUI. When the `FXMLLoader` loads `TipCalculator.fxml` to create the GUI, it also initializes each of the controller's instance variables that are declared with `@FXML` to ensure that they refer to the corresponding GUI components in the FXML file.

`TipCalculatorController`'s `calculateButtonPressed` Event Handler

Figure 12.22 presents class `TipCalculatorController`'s calculateButtonPressed method, which is called when the user clicks the **Calculate** Button. The @FXML annotation (line 40) preceding the method indicates that this method can be used to specify a control's event handler in the FXML file that describes the app's GUI. For a control that generates an ActionEvent (as is the case for many JavaFX controls), the event-handling method must return void and receive one ActionEvent parameter (line 41).

```
39    // calculates and displays the tip and total
40    @FXML
41    private void calculateButtonPressed(ActionEve
42       try {
43          BigDecimal amount = new BigDecimal(amou
44          BigDecimal tip = amount.multiply(tipPer
45          BigDecimal total = amount.add(tip);
46
47          tipTextField.setText(currency.format(ti
48          totalTextField.setText(currency.format(
49       }
50       catch (NumberFormatException ex) {
51          amountTextField.setText("Enter amount")
52          amountTextField.selectAll();
53          amountTextField.requestFocus();
54       }
55    }
56
```

Fig. 12.22

TipCalculatorController's

`calculateButtonPressed` event handler.

Registering the Calculate Button’s Event Handler

When the `FXMLLoader` loads `TipCalculator.fxml` to create the GUI, it creates and registers an event handler for the **Calculate** `Button`’s `ActionEvent`. The event handler for this event must implement interface `EventHandler<ActionEvent>`—`EventHandler` is a generic type, like `ArrayList` (introduced in Chapter 7). This interface contains a `handle` method that returns `void` and receives an `ActionEvent` parameter. This method’s body, in turn, calls method `calculateButtonPressed` when the user clicks the **Calculate** `Button`. `FXMLLoader` performs similar tasks for every event listener you specify via the Scene Builder **Inspector** window’s **Code** section.

Calculating and Displaying the Tip and Total Amounts

Lines 43–48 calculate and display the tip and total. Line 43 calls the `amountTextField`’s `getText` method to get the bill amount typed by the user. This `String` is passed to `Big-Decimal`’s constructor, which throws a `NumberFormatException` if its argument is not a number. In that case, line 51 calls `amountTextField`'s `setText` method to display the message `"Enter amount"` in the `TextField`. Line 52 then calls method `selectAll` to select the `TextField`'s text and line 53 calls `requestFocus` to give the `TextField` the focus. Now the user can immediately type a value in the `amountTextField` without having to first select its text. Methods `getText`, `setText` and `selectAll` are inherited into class `TextField` from class `TextInputControl` (package `javafx.scene.control`), and method `requestFocus` is inherited into class `TextField` from class `Node` (package `javafx.scene`).

If line 43 does not throw an exception, line 44 calculates the `tip` by calling method `multiply` to multiply the `amount` by the `tipPercentage`, and line 45 calculates the `total` by `add`ing the `tip` to the bill `amount`. Next lines 47 and 48 use the `currency` object's `format` method to create currency-formatted `String`s representing the tip and total amounts, which we display in `tipTextField` and `totalTextField`, respectively.

`TipCalculatorController`'s `initalize` Method

Figure 12.23 presents class `TipCalculatorController`'s `initialize` method.

This method can be used to configure the controller before the GUI is displayed. Line 60 calls the `currency` object’s `setRoundingMode` method to specify how currency values should be rounded. The value `RoundingMode.HALF_UP` indicates that values greater than or equal to .5 should round up—for example, 34.567 would be formatted as 34.57 and 34.564 would be formatted as 34.56.

```
57    // called by FXMLLoader to initialize the con
58    public void initialize() {
59       // 0-4 rounds down, 5-9 rounds up
60       currency.setRoundingMode(RoundingMode.HALF
61
62       // listener for changes to tipPercentageSl
63       tipPercentageSlider.valueProperty().addLis
64          new ChangeListener<Number>() {
65             @Override
66             public void changed(ObservableValue<
67                Number oldValue, Number newValue)
68                tipPercentage =
69                   BigDecimal.valueOf(newValue.in
70                tipPercentageLabel.setText(percen
71             }
72          }
73       );
74    }
75 }
```

Fig. 12.23

`TipCalculatorController`’s `initalize` method.

Using an Anonymous Inner Class for Event Handling

Each JavaFX control has properties. Some—such as a `Slider`'s value—can generate events when they change. For such events, you must manually register as the event handler an object that implements the `ChangeListener` interface (package `javafx.beans.value`).

`ChangeListener` is a generic type that's specialized with the property's type. The call to `valueProperty` (line 63) returns a `DoubleProperty` (package `javax.beans.property`) that represents the `Slider`'s value. A `DoubleProperty` is an `ObservableValue<Number>` that can notify listeners when a value changes. Each class that implements interface `ObservableValue` provides method `addListener` (called on line 63) to register an event-handler that implements interface `ChangeListener`. For a `Slider`'s value, `addListener`'s argument is an object that implements `ChangeListener<Number>`, because the `Slider`'s value is a numeric value.

If an event handler is not reused, you often define it as an instance of an **anonymous inner class**—a class that's declared without a name and typically appears inside a method. The `addListener` method's argument is specified in lines 64–72 as one statement that

- declares the event listener's class,

- creates an object of that class and
- registers it as the listener for changes to the `tipPercentageSlider`'s value.

Since an anonymous inner class has no name, you must create an object of the class at the point where it's declared (thus the keyword `new` in line 64). A reference to that object is then passed to `addListener`. After the `new` keyword, the syntax

```
ChangeListener<Number>()
```

in line 64 begins the declaration of an anonymous inner class that implements interface `ChangeListener<Number>`. This is similar to beginning a class declaration with

```
public class MyHandler implements ChangeListener<Numb
```

The opening left brace at 64 and the closing right brace at line 72 delimit the anonymous inner class's body. Lines 65–71 declare the interface's `changed` method, which receives a reference to the `ObservableValue` that changed, a `Number` containing the `Slider`'s old value before the event occurred and a `Number` containing the `Slider`'s new value. When the user moves the `Slider`'s thumb, lines 68–69 store the new tip percentage and line 70 updates the `tipPercentageLabel`. (The notation `? extends Number` in line 66 indicates that the `ObservableValue`'s type argument is a `Number` or a subclass of `Number`. We explain this notation in more detail in Section 20.7.)

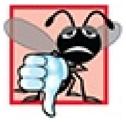
Anonymous Inner Class Notes

8

An anonymous inner class can access its top-level class's instance variables, `static` variables and methods—in this case, the anonymous inner class uses the instance variables `tipPercentage` and `tipPercentageLabel`, and the `static` variable `percent`. However, an anonymous inner class has limited access to the local variables of the method in which it's declared—it can access only the `final` or effectively `final` (Java SE 8) local variables declared in the enclosing method's body.

Software Engineering Observation 12.2

The event listener for an event must implement the appropriate event-listener interface.



Common Programming Error 12.1

If you forget to register an event-handler object for a particular GUI component's event type, events of that type will be ignored.

Java SE 8: Using a Lambda to Implement the `ChangeListener`

8

Recall from Section 10.10 that in Java SE 8 an interface containing one method—such as `ChangeListener` in Fig. 12.23—is a functional interface. We'll show how to implement such interfaces with lambdas in Chapter 17.

12.6 Features Covered in the Other JavaFX Chapters

JavaFX is a robust GUI, graphics and multimedia technology. In Chapters 13 and 22, you'll:

- Learn additional JavaFX layouts and controls.
- Handle other event types (such as `MouseEvent`s).
- Apply transformations (such as moving, rotating, scaling and skewing) and effects (such as drop shadows, blurs, reflection and lighting) to a scene graph's nodes.
- Use CSS to specify the look-and-feel of controls.
- Use JavaFX properties and data binding to enable automatic updating of controls as corresponding data changes.
- Use JavaFX graphics capabilities.
- Perform JavaFX animations.
- Use JavaFX multimedia capabilities to play audio and video.

In addition, our JavaFX Resource Center

```
http://www.deitel.com/JavaFX
```

contains links to online resources where you can learn more about JavaFX's capabilities.

12.7 Wrap-Up

In this chapter, we introduced JavaFX. We presented the structure of a JavaFX stage (the application window). You learned that the stage displays a scene graph, that the scene graph is composed of nodes and that nodes consist of layouts and controls.

You designed GUIs using visual programming techniques in JavaFX Scene Builder, which enabled you to create GUIs without writing any Java code. You arranged `Label`, `ImageView`, `TextField`, `Slider` and `Button` controls using the `VBox` and `GridPane` layout containers. You learned how class `FXMLLoader` uses the FXML created in Scene Builder to create the GUI.

You implemented a controller class to respond to user interactions with `Button` and `Slider` controls. We showed that certain event handlers can be specified directly in FXML from Scene Builder, but event handlers for changes to a control's property values must be implemented directly in the controllers code. You also learned that the `FXMLLoader` creates and initializes an instance of an application's controller class, initializes the controller's instance variables that are declared with the `@FXML` annotation, and creates and registers event handlers for any events specified in the FXML.

In the next chapter, you'll use additional JavaFX controls and layouts and use CSS to style your GUI. You’ll also learn more about JavaFX properties and how to use a technique called data binding to automatically update elements in a GUI with new data.

Summary

Section 12.1 Introduction

- A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an app. A GUI (pronounced “GOO-ee”) gives an app a distinctive “look-and-feel”.
- GUIs are built from GUI components—sometimes called controls or widgets.
- Providing different apps with consistent, intuitive user-interface components gives users a sense of familiarity with a new app, so that they can learn it more quickly and use it more productively.
- Java’s GUI, graphics and multimedia API of the future is JavaFX.

Section 12.2 JavaFX Scene Builder

- The Scene Builder tool is a standalone JavaFX GUI visual layout tool that can also be used with various IDEs.
- JavaFX Scene Builder enables you to create GUIs by dragging and dropping GUI components from Scene Builder’s library onto a design area, then modifying and styling the GUI—all without writing any code.
- JavaFX Scene Builder’s live editing and preview features allow you to view your GUI as you create and modify it, without compiling and running the app.
- You can use Cascading Style Sheets (CSS) to change the entire look-and-feel of your GUI—a concept sometimes called skinning.
- As you create and modify a GUI, JavaFX Scene Builder generates FXML (FX Markup Language)—an XML vocabulary for defining and arranging JavaFX GUI controls without writing any Java code.
- XML (eXtensible Markup Language) is a widely used language for describing things—it’s readable both by computers and by humans.
- FXML concisely describes GUI, graphics and multimedia elements.
- The FXML code is separate from the program logic that’s defined in Java source code.
- Separation of the interface (the GUI) from the implementation (the Java code) makes it easier to debug, modify and maintain JavaFX GUI apps.

Section 12.3 JavaFX App Window Structure

- The window in which a JavaFX app's GUI is displayed is known as the stage and is an instance of class `Stage` (package `javafx.stage`).
- The stage contains one scene that defines the GUI as a scene graph—a tree structure of an app's visual elements, such as GUI controls, shapes, images, video, text and more. The scene is an instance of class `Scene` (package `javafx.scene`).
- Each visual element in the scene graph is a node—an instance of a subclass of `Node` (package `javafx.scene`), which defines common attributes and behaviors for all nodes in the scene graph.
- The first node in the scene graph is known as the root node.
- Nodes that have children are typically layout containers that arrange their child nodes in the scene.
- The nodes arranged in a layout container are a combination of controls and possibly other layout containers.
- When the user interacts with a control, it generates an event. Programs can use event handling to specify what should happen when each user interaction occurs.
- An event handler is a method that responds to a user interaction. An FXML GUI's event handlers are defined in a controller class.

Section 12.4 Welcome App—Displaying Text and an Image

- Visual-programming techniques enable you to drag-and-drop JavaFX components onto Scene Builder's design area (known as the content panel), then use Scene Builder's Inspector to configure options.
- Layout containers help you arrange and size GUI components.
- A `VBox` layout container (package `javafx.scene.layout`) arranges its nodes vertically from top to bottom.
- To add a layout to Scene Builder's content panel, double-click the layout in the **Library** window's **Containers** section or drag-and-drop the layout from the **Containers** section onto Scene Builder's content panel.
- A `VBox`'s **Alignment** property determines the layout positioning of the `VBox`'s children.
- Each property value you specify for a JavaFX object is used to set one of that object's instance variables when JavaFX creates the object at runtime.
- The preferred size (width and height) of the scene graph's root node is used by the scene to determine its window size when the app begins executing.
- To add a control to a layout, drag-and-drop the control from the **Library** onto a layout in Scene Builder's content panel. You also can double-click an item in the **Library** to add it.
- You can set a `Label`'s text either by double clicking it and typing the new text, or by selecting the `Label` and setting its **Text** property in the **Inspector**'s **Properties** section.
- To set a `Label`'s font, select the `Label`, then in the **Inspector**'s **Properties** section, click the value to the right of the **Font** property. In the

window that appears, set the font’s attributes.

- You can reorder a `VBox`’s controls by dragging them in the `VBox` or in Scene Builder **Document** window’s **Hierarchy** section.
- To specify an `ImageView`’s image, select the `ImageView`, then in the **Inspector**’s **Properties** section click the ellipsis (**…**) button to the right of the **Image** property. Select the image from the dialog.
- To reset a property to its default value, hover the mouse over the property’s value. This displays a button to the right of the property’s value. Click the button and select **Reset to Default** to reset the value.
- You can preview what a design will look like in a running application’s window by selecting **Preview > Show Preview in Window**.

Section 12.5.2 Technologies Overview

- A JavaFX app’s main class inherits from `Application` (package `javafx.application`).
- The main class’s `main` method calls class `Application`’s `static launch` method to begin executing a JavaFX app. This method, in turn, causes the JavaFX runtime to create an object of the `Application` subclass and call its `start` method, which creates the GUI, attaches it to a `Scene` and places it on the `Stage` that method start receives as an argument.
- A `GridPane` (package `javafx.scene.layout`) arranges JavaFX nodes into columns and rows in a rectangular grid.
- Each cell in a `GridPane` can be empty or can hold one or more JavaFX components, including layout containers that arrange other controls.
- Each component in a `GridPane` can span multiple columns or rows.
- A `TextField` (package `javafx.scene.control`) can accept text input or display text.
- A `Slider` (package `javafx.scene.control`) represents a value in the range 0.0–100.0 by default and allows the user to select a number in that range by moving the `Slider`’s thumb.
- A `Button` (package `javafx.scene.control`) allows the user to initiate an action.
- Class `NumberFormat` (package `java.text`) can format locale-specific currency and percentage strings.
- GUIs are event driven. When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform a task.
- The code that performs a task in response to an event is called an event

handler.

- For certain events you can link a control to its event-handling method by using the **Code** section of Scene Builder's **Inspector** window. In this case, the class that implements the event-listener interface will be created for you and will call the method you specify.
- For events that occur when the value of a control's property changes, you must create the event handler entirely in code.
- You implement the `ChangeListener` interface (package `javafx.beans.value`) to respond when the user moves the `Slider`'s thumb.
- JavaFX applications in which the GUI is implemented as FXML adhere to the Model-View-Controller (MVC) design pattern, which separates an app's data (contained in the model) from the app's GUI (the view) and the app's processing logic (the controller). The controller implements logic for processing user inputs. The view presents the data stored in the model. When a user provides input, the controller modifies the model with the given input. When the model changes, the controller updates the view to present the changed data. In a simple app, the model and controller are often combined into a single class.
- In a JavaFX FXML app, you define the app's event handlers in a controller class. The controller class defines instance variables for interacting with controls programmatically, as well as event-handling methods.
- Class `FXMLLoader`'s `static` method `load` uses the FXML file that represents the app's GUI to creates the GUI's scene graph and returns a `Parent` (package `javafx.scene`) reference to the scene graph's root node. It also initializes the controller's instance variables, and creates and registers the event handlers for any events specified in the FXML.

Section 12.5.3 Building the App’s GUI

- If a control or layout will be manipulated programmatically in the controller class, you must provide a name for that control or layout. Each object’s name is specified via its **fx:id** property. You can set this property’s value by selecting a component in your scene, then expanding the **Inspector** window’s **Code** section—the **fx:id** property appears at the top.
- By default, the `GridPane` contains two columns and three rows. To add a row above or below an existing row, right click the row’s tab and select either **Grid Pane > Add Row Above** or **Grid Pane > Add Row Below**.
- You can delete a row or column by right clicking the tab containing its row or column number and selecting **Delete**.
- You can set a `Button`’s text by double clicking it, or by selecting the `Button`, then setting its **Text** property in the **Inspector** window’s **Properties** section.
- A `GridPane` column’s contents are left-aligned by default. To change the alignment, select the column by clicking the tab at the top or bottom of the column, then in the **Inspector**’s **Layout** section, set the **Halignment** property.
- Setting a node’s **Pref Width** property of a `GridPane` column to its default `USE_COMPUTED_SIZE` value indicates that the width should be based on the widest child.
- To size a `Button` the same width as the other controls in a `GridPane`’s column, select the `Button`, then in the **Inspector**’s **Layout** section, set the **Max Width** property to `MAX_VALUE`.
- The space between a node’s contents and its top, right, bottom and left edges is known as the padding, which separates the contents from the node’s edges. To set the padding, select the node, then in the **Inspector**’s **Layout** section set the **Padding** property’s values.

- You can specify the default amount of space between a `GridPane`'s columns and rows with its **Hgap** (horizontal gap) and **Vgap** (vertical gap) properties, respectively.
- You can type in a `TextField` only if it's "in focus"—that is, it's the control that the user is interacting with. When you click an interactive control, it receives the focus. Similarly, when you press the *Tab* key, the focus transfers from the current focusable control to the next one—this occurs in the order the controls were added to the GUI.

Section 12.5.4
`TipCalculator` Class

- To display a GUI, you must attach it to a `Scene`, then attach the `Scene` to the `Stage` that's passed into `Application` method `start`.
- By default, the `Scene`'s size is determined by the size of the scene graph's root node. Overloaded versions of the `Scene` constructor allow you to specify the Scene's size and fill (a color, gradient or image), which appears in the `Scene`'s background.
- `Stage` method `setTitle` specifies the text that appears in the `Stage` window's title bar.
- `Stage` method `setScene` places a `Scene` onto a `Stage`.
- `Stage` method `show` displays the `Stage` window.

Section 12.5.5 `TipCalculatorController` Class

- The `RoundingMode enum` of package `java.math` is used to specify how `BigDecimal` values are rounded during calculations or when formatting floating-point numbers as `String`s.
- Class `NumberFormat` of package `java.text` provides numeric formatting capabilities, such as locale-specific currency and percentage formats.
- A `Button`'s event handler receives an `ActionEvent`, which indicates that the `Button` was clicked. Many JavaFX controls support `ActionEvent`s.
- Package `javafx.scene.control` contains many JavaFX control classes.
- The `@FXML` annotation preceding an instance variable indicates that the variable's name can be used in the FXML file that describes the app's GUI. The variable names that you specify in the controller class must precisely match the **fx:id** values you specified when building the GUI.
- When the `FXMLLoader` loads an FXML file to create a GUI, it also initializes each of the controller's instance variables that are declared with `@FXML` to ensure that they refer to the corresponding GUI components in the FXML file.
- The `@FXML` annotation preceding a method indicates that the method can be used to specify a control's event handler in the FXML file that describes the app's GUI.
- When the `FXMLLoader` creates an object of a controller class, it determines whether the class contains an `initialize` method with no parameters and, if so, calls that method to initialize the controller. This

method can be used to configure the controller before the GUI is displayed.

- An anonymous inner class is a class that's declared without a name and typically appears inside a method declaration.
- Since an anonymous inner class has no name, one object of the class must be created at the point where the class is declared.
- An anonymous inner class can access its top-level class's instance variables, `static` variables and methods but has limited access to the local variables of the method in which it's declared—it can access only the `final` or effectively `final` (Java SE 8) local variables declared in the enclosing method's body.

13 JavaFX GUI: Part 2

Objectives

In this chapter you'll:

- Learn more details of laying out nodes in a scene graph with JavaFX layout panels.
- Continue building JavaFX GUIs with Scene Builder.
- Create and manipulate `RadioButton`s and `ListView`s.
- Use `BorderPane`s and `TitledPane`s to layout controls.
- Handle mouse events.
- Use property binding and property listeners to perform tasks when a control's property value changes.
- Programmatically create layouts and controls.
- Customize a `ListView`'s cells with a custom cell factory.
- See an overview of other JavaFX capabilities.
- Be introduced to the JavaFX 9 updates in Java SE 9.

Outline

13.1 Introduction

This chapter continues our JavaFX presentation[1] that began in Chapter 12. In this chapter, you'll:

1. The corresponding Swing GUI chapter is now online Chapter 35 and can be covered after online Chapter 26, which requires as prerequisites the only Chapters 1 through 11.

- Use additional layouts (`TitledPane`, `BorderPane` and `Pane`) and controls (`RadioButton` and `ListView`).
- Handle mouse and `RadioButton` events.
- Set up event handlers that respond to property changes on controls (such as the value of a `Slider`).
- Display `Rectangle`s and `Circle`s as nodes in the scene graph.
- Bind a collection of objects to a `ListView` that displays the collection's contents.
- Customize the appearance of a `ListView`'s cells.

Finally, we overview other JavaFX capabilities and mention Java SE 9's JavaFX changes that are discussed in our online Java SE 9 chapters.

13.2 Laying Out Nodes in a Scene Graph

A layout determines the size and positioning of nodes in the scene graph.

Node Size

In general, a node's size should *not* be defined *explicitly*. Doing so often creates a design that looks pleasing when it first loads, but deteriorates when the app is resized or the content updates. In addition to the `width` and `height` properties, most JavaFX nodes have the properties `prefWidth`, `prefHeight`, `minWidth`, `minHeight`, `maxWidth` and `maxHeight` that specify a node's *range* of acceptable sizes as it's laid out within its parent node:

- The minimum size properties specify a node's smallest allowed size in points.
- The maximum size properties specify a node's largest allowed size in points.
- The preferred size properties specify a node's preferred width and height that should be used by the layout in most cases.

Node Position and Layout

Panes

A node’s position should be defined *relative* to its parent node and the other nodes in its parent. JavaFX **layout panes** are container nodes that arrange their child nodes in a scene graph relative to one another, based on their sizes and positions. Child nodes are controls, other layout panes, shapes and more.

Most JavaFX layout panes use *relative positioning*—if a layout-pane node is resized, it adjusts its children’s sizes and positions accordingly, based on their preferred, minimum and maximum sizes. Figure 13.1 describes each of the JavaFX layout panes, including those presented in Chapter 12. In this chapter, we’ll use `Pane`, `BorderPane`, `GridPane` and `VBox` from the `javafx.scene.layout` package.

Layout	Description
`AnchorPane`	Enables you to set the position of child nodes relative to the pane’s edges. Resizing the pane does not alter the layout of the nodes.
`BorderPane`	Includes five areas—top, bottom, left, center and right—where you can place nodes. The top and bottom regions fill the `BorderPane`’s width and are vertically sized to their children’s preferred heights. The left and right regions fill the `BorderPane`’s height and are horizontally sized to their children’s preferred widths. The center area occupies all of the `BorderPane`’s remaining space. You might use the different areas for tool bars, navigation, a main content area, etc.
`FlowPane`	Lays out nodes consecutively—either horizontally or vertically. When the boundary for the pane is reached, the nodes wrap to a new line in a horizontal `FlowPane` or a new column in a vertical `FlowPane`.
`GridPane`	Creates a flexible grid for laying out nodes in rows and

	columns.
`Pane`	The base class for layout panes. This can be used to position nodes at fixed locations—known as absolute positioning.
`StackPane`	Places nodes in a stack. Each new node is stacked atop the previous node. You might use this to place text on top of images, for example.
`TilePane`	A horizontal or vertical grid of equally sized tiles. Nodes that are tiled horizontally wrap at the `TilePane`'s width. Nodes that are tiled vertically wrap at the `TilePane`'s height.
`HBox`	Arranges nodes horizontally in one row.
`VBox`	Arranges nodes vertically in one column.

Fig. 13.1

JavaFX layout panes.

13.3 Painter App: `RadioButtons`, Mouse Events and Shapes

In this section, you'll create a simple **Painter** app (Fig. 13.2) that allows you to drag the mouse to draw. First, we'll overview the technologies you'll use, then we'll discuss creating the app's project and building its GUI. Finally, we'll present the source code for its `Painter` and `PainterController` classes.

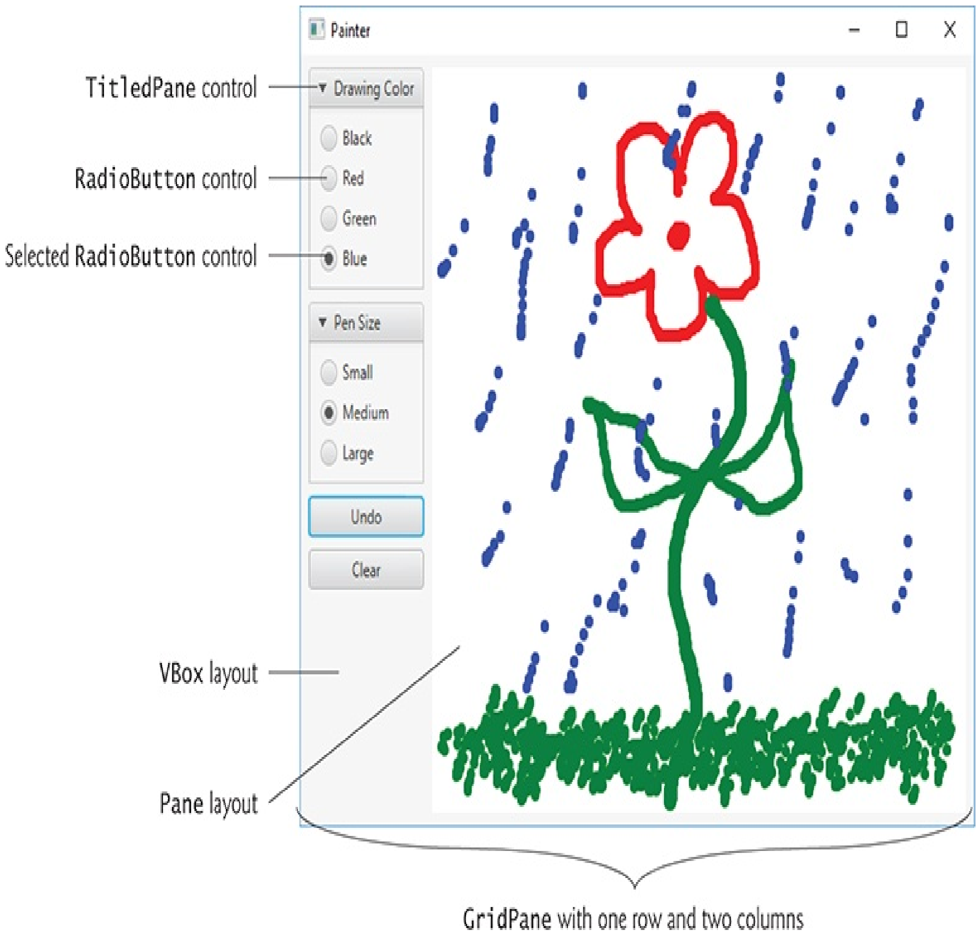


Fig. 13.2

Painter app.

Description

13.3.1 Technologies Overview

This section introduces the JavaFX features you’ll use in the

Painter app.

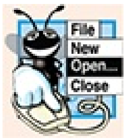
RadioButtons and ToggleGroups

RadioButtons function as *mutually exclusive* options. You add multiple RadioButtons to a ToggleGroup to ensure that only one RadioButton in a given group is selected at a time. For this app, you'll use JavaFX Scene Builder's capability for specifying each RadioButton's ToggleGroup in FXML; however, you can also create a ToggleGroup in Java, then use a RadioButton's setToggleGroup method to specify its ToggleGroup.

BorderPane Layout Container

A BorderPane **layout container** arranges controls into one or more of the five regions shown in Fig. 13.3. The top and bottom areas have the same width as the BorderPane. The left, center and right areas fill the vertical space between the top and bottom areas.

Each area may contain only one control or one layout container that, in turn, may contain other controls.



Look-and-Feel Observation 13.1

All the areas in a `BorderPane` *are optional: If the top or bottom area is empty, the left, center and right areas expand vertically to fill that area. If the left or right area is empty, the center expands horizontally to fill that area.*

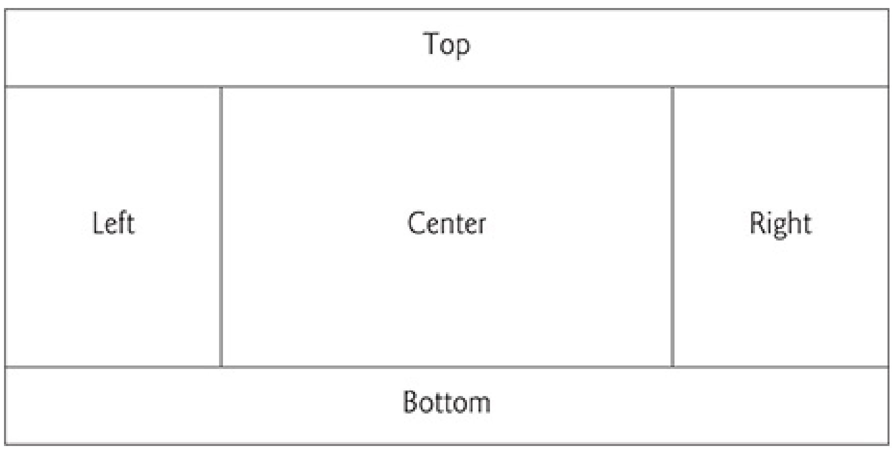


Fig. 13.3

`BorderPane`’s five areas.

`TitledPane` Layout Container

A `TitledPane` **layout container** displays a title at its top and is a collapsible panel containing a layout node, which in turn contains other nodes. You'll use `TitledPane`s to organize the app's `RadioButton`s and to help the user understand the purpose of each `RadioButton` group.

JavaFX Shapes

The `javafx.scene.shape` package contains various classes for creating 2D and 3D shape nodes that can be displayed in a scene graph. In this app, you'll programmatically create `Circle` objects as the user drags the mouse, then attach them to the app's drawing area so that they're displayed in the scene graph.

Pane Layout Container

Each `Circle` you programmatically create is attached to an `Pane` layout (the drawing area) at a specified *x*-*y* coordinate measured from the `Pane`'s upper-left corner.

Mouse Event Handling

When you drag the mouse, the app's controller responds by displaying a `Circle` (in the currently selected color and pen size) at the current mouse position in the `Pane`. JavaFX nodes support various mouse events, which are summarized in Fig.

13.4. For this app, you’ll configure an `onMouseDragged` event handler for the `Pane`. JavaFX also supports other types of input events. For example, for touchscreen devices there are various touch-oriented events and for keyboards there are various key events. For a complete list of JavaFX node events, see the `Node` class’s properties that begin with the word “on” at:

```
http://docs.oracle.com/javase/8/javafx/api/javafx/sce
```

Mouse events	When the event occurs for a given node
`onMouseClicked`	When the user clicks a mouse button—that is, presses and releases a mouse button without moving the mouse—with the mouse cursor within that node.
`onMouseDragEntered`	When the mouse cursor enters a node’s bounds during a mouse drag—that is, the user is moving the mouse with a mouse button pressed.
`onMouseDragExited`	When the mouse cursor exits the node’s bounds during a mouse drag.
`onMouseDragged`	When the user begins a mouse drag with the mouse cursor within that node and continues moving the mouse with a mouse button pressed.
`onMouseDragOver`	When a drag operation that started in a *different* node continues with the mouse cursor over the given node.
`onMouseDragReleased`	When the user completes a drag operation that began in that node.
`onMouseEntered`	When the mouse cursor enters that node’s bounds.
`onMouseExited`	When the mouse cursor exits that node’s

	bounds.
`onMouseMoved`	When the mouse cursor moves within that node’s bounds.
`onMousePressed`	When user presses a mouse button with the mouse cursor within that node’s bounds.
`onMouseReleased`	When user releases a mouse button with the mouse cursor within that node’s bounds.

Fig. 13.4

Mouse events.

Setting a Control’s User Data

Each JavaFX control has a `setUserData` **method** that receives an `Object`. You can use this to store any object you’d like to associate with that control. With each drawing-color `RadioButton`, we store the specific `Color` that the `RadioButton` represents. With each pen size `RadioButton`, we store an `enum` constant for the corresponding pen size. We then use these objects when handling the `RadioButton` events.

13.3.2 Creating the

`Painter.fxml` File

Create a folder on your system for this example's files, then open Scene Builder and save the new FXML file as `Painter.fxml`. If you already have an FXML file open, you also can choose **File > New** to create a new FXML file, then save it.

13.3.3 Building the GUI

In this section, we'll discuss the **Painter** app's GUI. Rather than providing the exact steps as we did in Chapter 12, we'll provide general instructions for building the GUI and focus on specific details for new concepts.

Software Engineering Observation 13.1

*As you build a GUI, it's often easier to manipulate layouts and controls via Scene Builder's **Hierarchy** window than directly in the stage design area.*

fx:id Property Values for This App's Controls

Figure 13.5 shows the **fx:id** properties of the **Painter** app's programmatically manipulated controls. As you build the GUI, you should set the corresponding **fx:id** properties in the FXML document, as we discussed in Chapter 12.

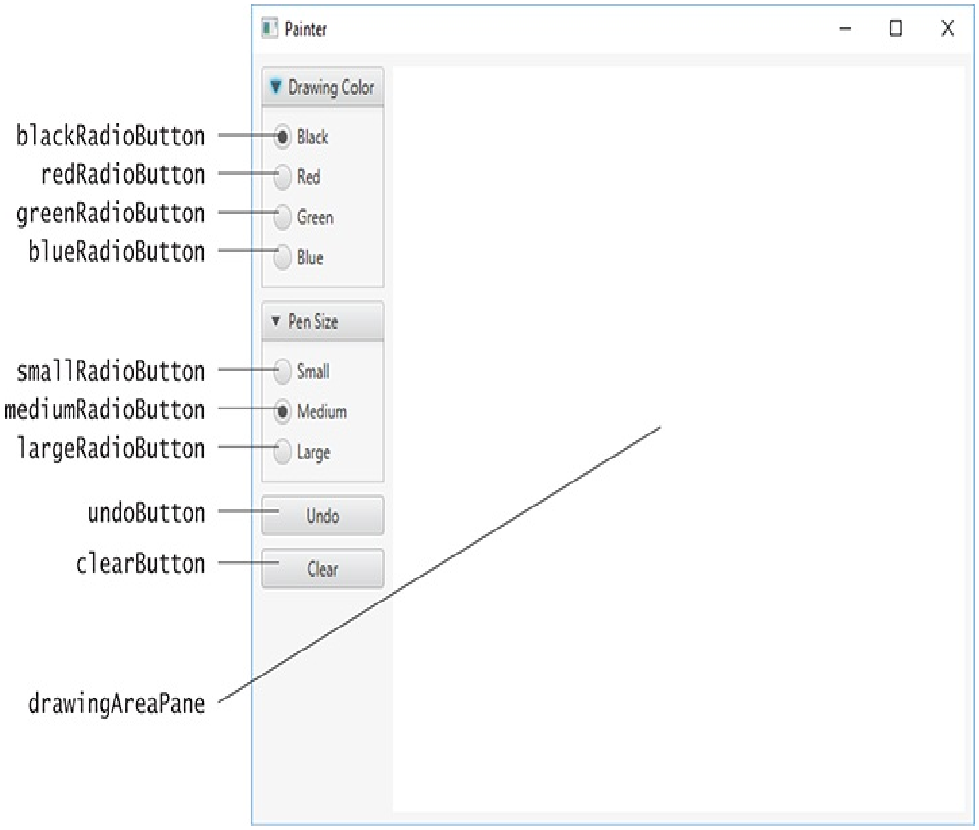


Fig. 13.5

Painter GUI labeled with **fx:id**s for the programmatically manipulated controls.

Description

Step 1: Adding a `BorderPane` as the Root Layout Node

Drag a `BorderPane` from the Scene Builder **Library** window's **Containers** section onto the content panel.

Step 2: Configuring the `BorderPane`

We set the `GridPane`'s **Pref Width** and **Pref Height** properties to `640` and `480` respectively. Recall that the stage's size is determined based on the size of the root node in the FXML document. Set the `BorderPane`'s **Padding** property to `8` to inset it from the stage's edges.

Step 3: Adding the `VBox` and `Pane`

Drag a `VBox` into the `BorderPane`'s left area and an `Pane` into the center area. As you drag over the `BorderPane`, Scene Builder shows the layout's five areas and highlights the area in which area the item you're dragging will be placed when you release the mouse. Set the `Pane`'s **fx:id** to `drawingAreaPane` as specified in Fig. 13.5.

For the `VBox`, set its **Spacing** property (in the **Inspector**'s **Layout** section) to `8` to add some vertical spacing between the controls that will be added to this container. Set its right **Margin** property to `8` to add some horizontal spacing between the `VBox` and the `Pane` be added to this container. Also reset its **Pref Width** and **Pref Height** properties to their default values (`USE_COMPUTED_SIZE`) and set its **Max Height** property to `MAX_VALUE`. This will enable the `VBox` to be as wide as it needs to be to accommodate its child nodes and occupy the full column height.

Reset the `Pane`'s **Pref Width** and **Pref Height** to their default `USE_COMPUTED_SIZE` values, and set its **Max Width** and **Max Height** to `MAX_VALUE` so that it occupies the full width and height of the `BorderPane`'s center area. In the **JavaFX CSS** category of the **Inspector** window's **Properties** section, click the field below **Style** (which is initially empty) and select `-fx-background-color` to indicate that you'd like to specify the `Pane`'s background color. In the field to the right, specify `white`.

Step 4: Adding the `TitledPanes` to the `VBox`

From the **Library** window's **Containers** section, drag two **TitledPane (empty)** objects onto the `VBox`. For the first `TitledPane`, set its **Text** property to `Drawing Color`. For the second, set its **Text** property to `Pen Size`.

Step 5: Customizing the TitledPanes

Each TitledPane in the completed GUI contains multiple RadioButtons. We'll use a VBox within each TitledPane to help arrange those controls. Drag a VBox onto each TitledPane. For each VBox, set its **Spacing** property to 8 and its **Pref Width** and **Pref Height** to USE_COMPUTED_SIZE so the VBoxes will be sized based on their contents.

Step 6: Adding the RadioButtons to the VBox

From the **Library** window's **Controls** section, drag four RadioButtons onto the VBox for the **Drawing Color** TitledPane, and three RadioButtons onto the VBox for the **Pen Size** TitledPane, then configure their **Text** properties and **fx:id**s as shown in Fig. 13.5. Select the blackRadioButton and ensure that its **Selected** property is checked, then do the same for the mediumRadioButton.

Step 7: Specifying the ToggleGroups for the

RadioButtons

Select all four RadioButtons in the first TitledPane's VBox, then set the **Toggle Group** property to colorToggleGroup. When the FXML file is loaded, a ToggleGroup object by that name will be created and these four RadioButtons will be associated with it to ensure that only one is selected at a time. Repeat this step for the three RadioButtons in the second TitledPane's VBox, but set the **Toggle Group** property to sizeToggleGroup.

Step 8: Changing the TitledPanes' Preferred Width and Height

For each TitledPane, set its **Pref Width** and **Pref Height** to USE_COMPUTED_SIZE so the TitledPanes will be sized based on their contents.

Step 9: Adding the Buttons

Add two Buttons below the TitledPanes, then configure their **Text** properties and **fx:id**s as shown in Fig. 13.5. Set each Button's **Max Width** property to MAX_VALUE so that they fill the VBox's width.

Step 10: Setting the Width the VBox

We’d like the `VBox` to be only as wide as it needs to be to display the controls in that column. To specify this, select the `VBox` in the **Document** window’s **Hierarchy** section. Set the column’s **Min Width** and **Pref Width** to `USE_COMPUTED_SIZE`, then set the **Max Width** to `USE_PREF_SIZE` (which indicates that the maximum width should be the preferred width). Also, reset the **Max Height** to its default `USE_COMPUTED_SIZE` value. The GUI is now complete and should appear as shown in Fig. 13.5.

Step 11: Specifying the Controller Class’s Name

As we mentioned in Section 12.5.2, in a JavaFX FXML app, the app’s controller class typically defines instance variables for interacting with controls programmatically, as well as event-handling methods. To ensure that an object of the controller class is created when the app loads the FXML file at runtime, you must specify the controller class’s name in the FXML file:

1. Expand Scene Builder’s **Controller** window (located below the **Hierarchy** window).
2. In the **Controller Class** field, type `PainterController`.

Step 12: Specifying the Event-Handler Method Names

Next, you'll specify in the **Inspector** window's **Code** section the names of the methods that will be called to handle specific control's events:

- For the `drawingAreaPane`, specify `drawingAreaMouseDragged` as the **On Mouse Dragged** event handler (located under the **Mouse** heading in the **Code** section). This method will draw a circle in the specified color and size for each mousedragged event.
- For the four **Drawing Color** `RadioButton`s, specify `colorRadioButtonSelected` as each `RadioButton`'s **On Action** event handler. This method will set the current drawing color, based on the user's selection.
- For the three **Pen Size** `RadioButton`s, specify `sizeRadioButtonSelected` as each `RadioButton`'s **On Action** event handler. This method will set the current pen size, based on the user's selection.
- For the **Undo** `Button`, specify `undoButtonPressed` as the **On Action** event handler. This method will remove the last circle the user drew on the screen.
- For the **Clear** `Button`, specify `clearButtonPressed` as the **On Action** event handler. This method will clear the entire drawing.

Step 13: Generating a Sample Controller Class

As you saw in Section 12.5, Scene Builder generates the initial controller-class skeleton for you when you select **View** > **Show Sample Controller Skeleton**. You can copy this code into a `PainterController.java` file and store the file in the same folder as `Painter.fxml`. We show the completed `PainterController` class in Section 13.3.5.

13.3.4 `Painter` Subclass of `Application`

Figure 13.6 shows class `Painter` subclass of `Application` that launches the app, which performs the same tasks to start the **Painter** app as described for the **Tip Calculator** app in Section 12.5.4.

```
// Fig. 13.5: Painter.java
// Main application class that loads and displa
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Painter extends Application {
    @Override
    public void start(Stage stage) throws Excepti
        Parent root =
            FXMLLoader.load(getClass().getResource(

        Scene scene = new Scene(root);
        stage.setTitle("Painter"); // displayed in
        stage.setScene(scene);
        stage.show();
```

```
    }

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

Fig. 13.6

Main application class that loads and displays the **Painter**'s GUI.

13.3.5 `PainterController` Class

Figure 13.7 shows the final version of class `PainterController` with this app’s new features highlighted. Recall from Chapter 12 that the controller class defines instance variables for interacting with controls programmatically, as well as event-handling methods. The controller class may also declare additional instance variables, `static` variables and methods that support the app’s operation.

```
// Fig. 13.6: PainterController.java
// Controller for the Painter app
```

```
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.paint.Paint;
import javafx.scene.shape.Circle;

public class PainterController {
   // enum representing pen sizes
   private enum PenSize {
      SMALL(2),
      MEDIUM(4),
      LARGE(6);

      private final int radius;

      PenSize(int radius) {this.radius = radius;

      public int getRadius() {return radius;}
   };

   // instance variables that refer to GUI compo
   @FXML private RadioButton blackRadioButton;
   @FXML private RadioButton redRadioButton;
   @FXML private RadioButton greenRadioButton;
   @FXML private RadioButton blueRadioButton;
   @FXML private RadioButton smallRadioButton;
   @FXML private RadioButton mediumRadioButton;
   @FXML private RadioButton largeRadioButton;
   @FXML private Pane drawingAreaPane;
   @FXML private ToggleGroup colorToggleGroup;
   @FXML private ToggleGroup sizeToggleGroup;

   // instance variables for managing Painter st
   private PenSize radius = PenSize.MEDIUM; // r
   private Paint brushColor = Color.BLACK; // dr

```

```
   // set user data for the RadioButtons
   public void initialize() {
      // user data on a control can be any Objec
      blackRadioButton.setUserData(Color.BLACK);
      redRadioButton.setUserData(Color.RED);
      greenRadioButton.setUserData(Color.GREEN);
      blueRadioButton.setUserData(Color.BLUE);
      smallRadioButton.setUserData(PenSize.SMALL
      mediumRadioButton.setUserData(PenSize.MEDI
      largeRadioButton.setUserData(PenSize.LARGE
   }

   // handles drawingArea's onMouseDragged Mouse
   @FXML
   private void drawingAreaMouseDragged(MouseEve
      Circle newCircle = new Circle(e.getX(), e.
         radius.getRadius(), brushColor);
      drawingAreaPane.getChildren().add(newCircl
   }

   // handles color RadioButton's ActionEvents
   @FXML
   private void colorRadioButtonSelected(ActionE
      // user data for each color RadioButton is
      brushColor =
         (Color) colorToggleGroup.getSelectedTog
   }

   // handles size RadioButton's ActionEvents
   @FXML
   private void sizeRadioButtonSelected(ActionEv
      // user data for each size RadioButton is
      radius =
         (PenSize) sizeToggleGroup.getSelectedTo
   }

   // handles Undo Button's ActionEvents
   @FXML
   private void undoButtonPressed(ActionEvent ev
      int count = drawingAreaPane.getChildren().
```

```
83
84         // if there are any shapes remove the last
85         if (count > 0) {
86            drawingAreaPane.getChildren().remove(co
87         }
88      }
89
90      // handles Clear Button's ActionEvents
91      @FXML
92      private void clearButtonPressed(ActionEvent e
93         drawingAreaPane.getChildren().clear(); //
94      }
95   }
```

Fig. 13.7

Controller for the **Painter** app.

`PenSize` `enum`

Lines 15–25 define the nested `enum` type `PenSize`, which specifies three pen sizes—`SMALL`, `MEDIUM` and `LARGE`. Each has a corresponding radius that will be used when creating a `Circle` object to display in response to a mouse-drag event.

Java allows you to declare classes, interfaces and `enum`s as **nested types** inside other classes. Except for the anonymous inner class introduced in Section 12.5.5, all the classes, interfaces and `enum`s we've discussed were **top level**—that is, they *were* not declared *inside* another type. The `enum` type

`PenSize` is declared here as a `private` nested type because it's used only by class `PainterController`. We'll say more about nested types later in the book.

Instance Variables

Lines 28–37 declare the `@FXML` instance variables that the controller uses to programmatically interact with the GUI. Recall that the names of these variables must match the corresponding **fx:id** values that you specified in `Painter.fxml`; otherwise, the `FXMLLoader` will not be able to connect the GUI components to the instance variables. Two of the `@FXML` instance variables are `ToggleGroup`s—in the `RadioButton` event handlers, we'll use these to determine which `RadioButton` was selected. Lines 40–41 define two additional instance variables that store the current drawing `Color` and the current `PenSize`, respectively.

Method `initialize`

Recall that when the `FXMLLoader` creates a controller-class object, `FXMLLoader` determines whether the class contains an `initialize` method with no parameters and, if so, calls that method to initialize the controller. Lines 44–53 define method `initialize` to specify each `RadioButton`'s corresponding user data object—either a `Color` or a `PenSize`. You'll use these objects in the `RadioButton`s' event handlers.

`drawingAreaMouseDragged` Event Handler

Lines 56–61 define `drawingAreaMouseDragged`, which responds to drag events in the `drawingAreaPane`. Each mouse event handler you define must have one `MouseEvent` parameter (package `javafx.scene.input`). When the event occurs, this parameter contains information about the event, such as its location, whether any mouse buttons were pressed, which node the user interacted with and more. You specified `drawingAreaMouseDragged` in Scene Builder as the `drawingAreaPane`'s **On Mouse Dragged** event handler.

Lines 58–59 create a new `Circle` object using the constructor that takes as arguments the center point's *x*-coordinate, the center point's *y*-coordinate, the `Circle`'s radius and the `Circle`'s `Color`.

Next, line 60 attaches the new `Circle` to the `drawingAreaPane`. Each layout pane has a `getChildren` method that returns an `ObservableList<Node>` collection containing the layout's child nodes. An `ObservableList` provides methods for adding and removing elements. You'll learn more about `ObservableList` later in this chapter. Line 60 uses the `ObservableList`'s `add` method to add a new `Node` to the `drawingAreaPane`—all JavaFX shapes inherit indirectly from class `Node` in the `javafx.scene` package.

`colorRadioButtonSelected` Event Handler

Lines 64–69 define `colorRadioButtonSelected`, which responds to the `ActionEvent`s of the **Drawing Color** `RadioButton`s—these occur each time a new color `RadioButton` is selected. You specified this event handler in Scene Builder as the **On Action** event handler for all four **Drawing Color** `RadioButton`s.

Lines 67–68 set the current drawing `Color`. `ColorToggleGroup` method `getSelectedToggle` returns the `Toggle` that's currently selected. Class `RadioButton` is one of several controls (others are `RadioButtonMenuItem` and `ToggleButton`) that implement interface `Toggle`. We then use the `Toggle`'s `getUserData` method to get the user data `Object` that was associated with the corresponding `RadioButton` in method `initialize`. For the color `RadioButton`s, this `Object` is aways a `Color`, so we cast the `Object` to a `Color` and assign it to `brushColor`.

`sizeRadioButtonSelected` Event Handler

Lines 72–77 define `sizeRadioButtonSelected`, which responds to the pen size `RadioButton`s' `ActionEvent`s. You specified this event handler as the **On Action** event handler for all three **Pen Size** `RadioButton`s. Lines 75–76 set the current `PenSize`, using the same approach as setting the current color in method `colorRadioButtonSelected`.

`undoButtonPressed` Event Handler

Lines 80–88 define `undoButtonPressed`, which responds to an `ActionEvent` from the `undoButton` by removing the last `Circle` displayed. You specified this event handler in Scene Builder as the `undoButton`'s **On Action** event handler.

To undo the last `Circle`, we remove the last child from the `drawingAreaPane`'s collection of child nodes. First, line 82 gets the number of elements in that collection. Then, if that's greater than 0, line 86 removes the node at the last index in the collection.

`clearButtonPressed` Event Handler

Lines 91–94 define `clearButtonPressed`, which responds to the `ActionEvent` from the `clearButton` by clearing `drawingAreaPane`’s collection of child nodes. You specified this event handler in Scene Builder as the `clearButton`’s **On Action** event handler. Line 93 clears the collection of child nodes to erase the entire drawing.

13.4 Color Chooser App: Property Bindings and Property Listeners

In this section, we present a **Color Chooser** app (Fig. 13.8) that demonstrates property bindings and property listeners.

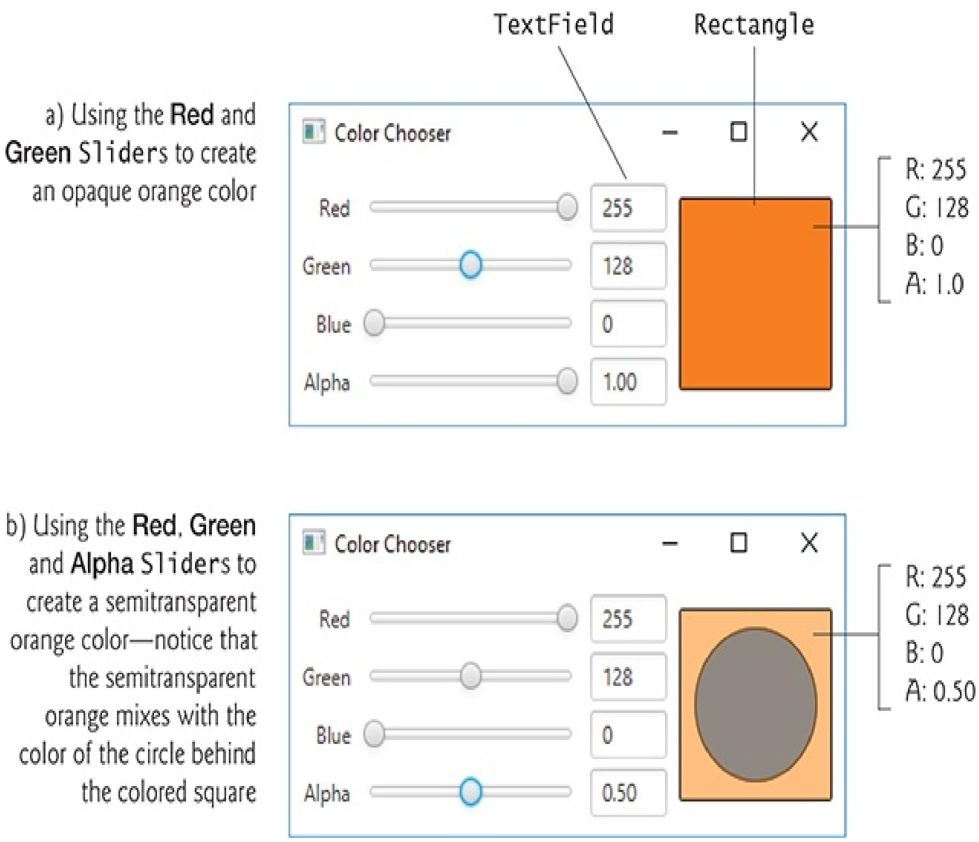


Fig. 13.8

Color Chooser app with opaque and semitransparent orange colors.

Description

13.4.1 Technologies Overview

In this section, we introduce the technologies you'll use to build the **Color Chooser**.

RGBA Colors

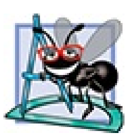
The app uses the **RGBA color system** to display a rectangle of color based on the values of four `Slider`s. In RGBA, every color is represented by its red, green and blue color values, each ranging from 0 to 255, where 0 denotes no color and 255 full color. For example, a color with a red value of 0 would contain no red component. The alpha value (A)—which ranges from 0.0 to 1.0—represents a color's *opacity*, with 0.0 being completely *transparent* and 1.0 completely *opaque*. The two colors in Fig. 13.8's sample outputs have the same RGB values, but the color displayed in Fig. 13.8(b) is *semitransparent*. You'll use a `Color` object that's created with RGBA values to fill a `Rectangle` that displays the `Color`.

Properties of a Class

JavaFX makes extensive use of properties. A **property** is defined by creating *set* and *get* methods with specific naming conventions. In general, the pair of methods that define a read/write property have the form:

```
public final void setPropertyName(Type propertyName)
public final Type getPropertyName()
```

Typically, such methods manipulate a corresponding *private* instance variable that has the same name as the property, but this is not required. For example, methods `setHour` and `getHour` together represent a property named `hour` and typically would manipulate a private `hour` instance variable. If the property represents a `boolean` value, its *get* method name typically begins with “`is`” rather than “`get`”—for example, `ArrayList` method `isEmpty`.

 Software Engineering Observation 13.2

Methods that define properties should be declared `final` *to prevent subclasses from overriding the methods, which could lead to unexpected results in client code.*

Property Bindings

JavaFX properties are implemented in a manner that makes them *observable*—when a property's value changes, other objects can respond accordingly. This is similar to event handling. One way to respond to a property change is via a **property binding**, which enables a property of one object to be updated when a property of another object changes. For example, you'll use property bindings to enable a `TextField` to display the corresponding `Slider`'s current value when the user moves that `Slider`'s thumb. Property bindings are not limited to JavaFX controls. Package `javafx.beans.property` contains many classes that you can use to define bindable properties in your own classes.

Property Listeners

Property listeners are similar to property bindings. A **property listener** is an event handler that's invoked when a property's value changes. In the event handler, you can respond to the property change in a manner appropriate for your app. In this app, when a `Slider`'s value changes, a property listener will store the value in a corresponding instance variable, create a new `Color` based on the values of all four `Slider`s and set that `Color` as the fill color of a `Rectangle` object that displays the current color. For more information on properties, property bindings and property listeners, visit:

```
http://docs.oracle.com/javase/8/javafx/properties-bin
```

13.4.2 Building the GUI

In this section, we'll discuss the **Color Chooser** app's GUI. Rather than providing the exact steps as we did in Chapter 12, we'll provide general instructions for building the GUI and focus on specific details for new concepts. As you build the GUI, recall that it's often easier to manipulate layouts and controls via the Scene Builder **Document** window's **Hierarchy** section than directly in the stage design area. Before proceeding, open Scene Builder and create an FXML file named `ColorChooser.fxml`.

fx:id Property Values for This App's Controls

Figure 13.9 shows the **fx:id** properties of the **Color Chooser** app's programmatically manipulated controls. As you build the GUI, you should set the corresponding **fx:id** properties in the FXML document, as you learned in Chapter 12.

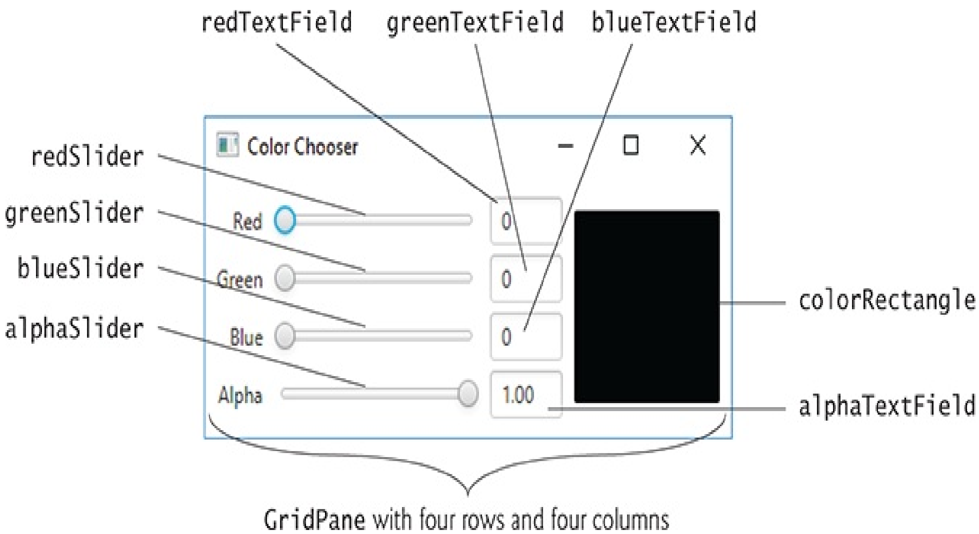


Fig. 13.9

Color Chooser app's programmatically manipulated controls labeled with their **fx:id**s.

Description

Step 1: Adding a `GridPane`

Drag a `GridPane` from the **Library** window's **Containers** section onto Scene Builder's content panel.

Step 2: Configuring the `GridPane`

This app's `GridPane` requires four rows and four columns.

Use the techniques you've learned previously to add two columns and one row to the `GridPane`. Set the `GridPane`'s **Hgap** and **Padding** properties to `8` to inset the `GridPane` from the stage's edges and to provide space between its columns.

Step 3: Adding the Controls

Using Fig. 13.9 as a guide, add the `Label`s, `Slider`s, `TextField`s, a `Circle` and a `Rectangle` to the `GridPane`—`Circle` and `Rectangle` are located in the Scene Builder **Library**'s **Shapes** section. When adding the `Circle` and `Rectangle`, place both into the rightmost column's first row. Be sure to add the `Circle` *before* the `Rectangle` so that it will be located *behind* the rectangle in the layout. Set the text of the `Label`s and `TextField`s as shown and set all the appropriate **fx:id** properties as you add each control.

Step 4: Configuring the `Sliders`

For the red, green and blue `Slider`s, set the **Max** properties to `255` (the maximum amount of a given color in the RGBA color scheme). For the alpha `Slider`, set its **Max** property to `1.0` (the maximum opacity in the RGBA color scheme).

Step 5: Configuring the `TextFields`

Set all of the `TextField`'s **Pref Width** properties to `50`.

Step 6: Configuring the `Rectangle`

Set the `Rectangle`'s **Width** and **Height** properties to `100`, then set its **Row Span** property to `Remainder` so that it spans all four rows.

Step 7: Configuring the `Circle`

Set the `Circle`'s **Radius** property to `40`, then set its **Row Span** property to `Remainder` so that it spans all four rows.

Step 8: Configuring the Rows

Set all four columns' **Pref Height** properties to `USE_COMPUTED_SIZE` so that the rows are only as tall as their content.

Step 9: Configuring the Columns

Set all four columns' **Pref Width** properties to `USE_COMPUTED_SIZE` so that the columns are only as wide as their content. For the leftmost column, set the **Halignment** property to `RIGHT`. For the rightmost column, set the **Halignment** property to `CENTER`.

Step 10: Configuring the `GridPane`

Set the `GridPane`'s **Pref Width** and **Pref Height** properties to `USE_COMPUTED_SIZE` so that it sizes itself, based on its contents. Your GUI should now appear as shown in Fig. 13.9.

Step 11: Specifying the Controller Class's Name

To ensure that an object of the controller class is created when the app loads the FXML file at runtime, specify `ColorChooserController` as the controller class's name in the FXML file as you've done previously.

Step 12: Generating a Sample Controller Class

Select **View > Show Sample Controller Skeleton**, then copy this code into a `ColorChooserController.java` file and store the file in the same folder as `ColorChooser.fxml`. We show the completed `ColorChooserController` class in Section 13.4.4.

13.4.3 `ColorChooser` Subclass of `Application`

Figure 13.6 shows the `ColorChooser` subclass of `Application` that launches the app. This class loads the FXML and displays the app as in the prior JavaFX examples.

```
// Fig. 13.8: ColorChooser.java
// Main application class that loads and display
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class ColorChooser extends Application {
   @Override
   public void start(Stage stage) throws Excepti
      Parent root =
         FXMLLoader.load(getClass().getResource(

```

```
      Scene scene = new Scene(root);
      stage.setTitle("Color Chooser");
      stage.setScene(scene);
      stage.show();
   }

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

Fig. 13.10

Application class that loads and displays the **Color Chooser**'s GUI.

13.4.4 `ColorChooserController` Class

Figure 13.11 shows the final version of class `ColorChooserController` with this app’s new features highlighted.

```
// Fig. 13.9: ColorChooserController.java
// Controller for the ColorChooser app
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
```

```
import javafx.fxml.FXML;
import javafx.scene.control.Slider;
import javafx.scene.control.TextField;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

public class ColorChooserController {
   // instance variables for interacting with GU
   @FXML private Slider redSlider;
   @FXML private Slider greenSlider;
   @FXML private Slider blueSlider;
   @FXML private Slider alphaSlider;
   @FXML private TextField redTextField;
   @FXML private TextField greenTextField;
   @FXML private TextField blueTextField;
   @FXML private TextField alphaTextField;
   @FXML private Rectangle colorRectangle;

   // instance variables for managing
   private int red = 0;
   private int green = 0;
   private int blue = 0;
   private double alpha = 1.0;

   public void initialize() {
      // bind TextField values to corresponding
      redTextField.textProperty().bind(
         redSlider.valueProperty().asString("%.0
      greenTextField.textProperty().bind(
         greenSlider.valueProperty().asString("%
      blueTextField.textProperty().bind(
         blueSlider.valueProperty().asString("%.
      alphaTextField.textProperty().bind(
         alphaSlider.valueProperty().asString("%

      // listeners that set Rectangle's fill bas
      redSlider.valueProperty().addListener(
         new ChangeListener<Number>() {
            @Override
            public void changed(ObservableValue<
```

```
            Number oldValue, Number newValue)
            red = newValue.intValue();
            colorRectangle.setFill(Color.rgb(
         }
      }
   );
   greenSlider.valueProperty().addListener(
      new ChangeListener<Number>() {
         @Override
         public void changed(ObservableValue<
            Number oldValue, Number newValue)
            green = newValue.intValue();
            colorRectangle.setFill(Color.rgb(
         }
      }
   );
   blueSlider.valueProperty().addListener(
      new ChangeListener<Number>() {
         @Override
         public void changed(ObservableValue<
            Number oldValue, Number newValue)
            blue = newValue.intValue();
            colorRectangle.setFill(Color.rgb(
         }
      }
   );
   alphaSlider.valueProperty().addListener(
      new ChangeListener<Number>() {
         @Override
         public void changed(ObservableValue<
            Number oldValue, Number newValue)
            alpha = newValue.doubleValue();
            colorRectangle.setFill(Color.rgb(
         }
      }
   );
  }
}
```

Fig. 13.11

Controller for the `ColorChooser` app.

Instance Variables

Lines 13–27 declare the controller’s instance variables. Variables `red`, `green`, `blue` and `alpha` store the current values of the `redSlider`, `greenSlider`, `blueSlider` and `alphaSlider`, respectively. These values are used to update the `colorRectangle`’s fill color each time the user moves a `Slider`’s thumb.

Method `initialize`

Lines 29–81 define method `initialize`, which initializes the controller after the GUI is created. In this app, `initialize` configures the property bindings and property listeners.

Property-to-Property Bindings

Lines 31–38 set up property bindings between a `Slider`’s value and the corresponding `TextField`’s text so that changing a `Slider` updates the corresponding `TextField`. Consider lines 31–32, which bind the `redSlider`'s `valueProperty` to the `redTextField's textProperty`:

```
redTextField.textProperty().bind(
  redSlider.valueProperty().asString("%.0f"));
```

Each `TextField` has a `text` property that's returned by its `textProperty` method as a `StringProperty` (package `javafx.beans.property`). `StringProperty` method `bind` receives an `ObservableValue` as an argument. When the `ObservableValue` changes, the bound property updates accordingly. In this case the `ObservableValue` is the result of the expression `redSlider.valueProperty().asString("%.0f")`. `Slider`'s `valueProperty` method returns the `Slider`'s `value` property as a `DoubleProperty`—an observable `double` value. Because the `TextField`'s `text` property must be bound to a `String`, we call `DoubleProperty` method `asString`, which returns a `StringBinding` object (an `ObservableValue`) that produces a `String` representation of the `DoubleProperty`. This version of `asString` receives a format-control `String` specifying the `DoubleProperty`'s format.

Property Listeners

To perform an arbitrary task when a property's value changes, register a property listener. Lines 41–80 register property listeners for the `Slider`s' `value` properties. Consider lines 41–50, which register the `ChangeListener` that executes when the user moves the `redSlider`'s thumb. As we did in Section 12.5 for the **Tip Calculator**'s `Slider`, we use an anonymous inner class to define the listener. Each `ChangeListener` stores the `int` value of the `newValue` parameter in a corresponding instance variable, then calls the `colorRectangle`'s `setFill` method to change its color, using `Color` method `rgb` to create the new `Color` object.

13.5 Cover Viewer App: Data-Driven GUIs with JavaFX Collections

Often an app needs to edit and display data. JavaFX provides a comprehensive model for allowing GUIs to interact with data. In this section, you'll build the **Cover Viewer** app (Fig. 13.12), which binds a list of `Book` objects to a `ListView`. When the user selects an item in the `ListView`, the corresponding `Book`'s cover image is displayed in an `ImageView`.

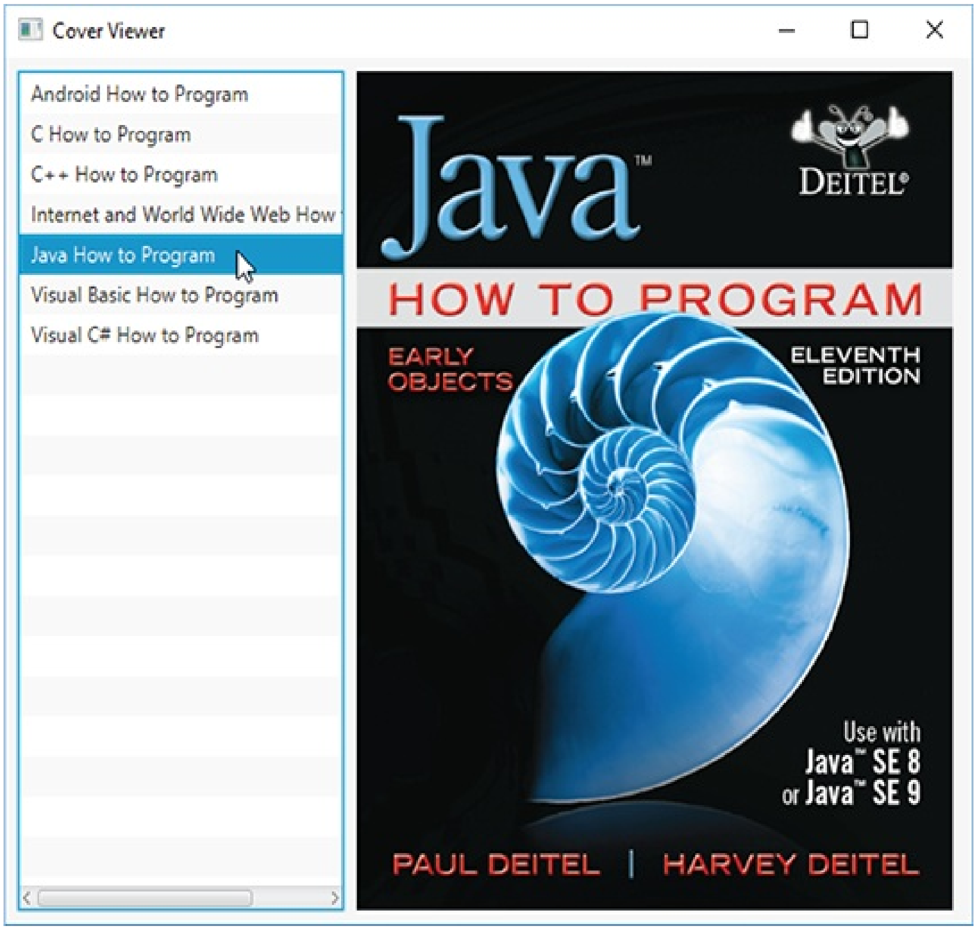


Fig. 13.12

Cover Viewer with Java How to Program selected.

Description

13.5.1 Technologies Overview

This app uses a `ListView` control to display a collection of book titles. Though you can individually add items to a `ListView`, in this app you'll bind an `ObservableList` object to the `ListView`. If you make changes to an `ObservableList`, its observer (the `ListView` in this app) will automatically be notified of those changes. Package `javafx.collections` defines `ObservableList` (similar to an `ArrayList`) and other observable collection interfaces. The package also contains class `FXCollections`, which provides `static` methods for creating and manipulating observable collections. You'll use a property listener to display the correct image when the user selects an item from the `ListView`—in this case, the property that changes is the selected item.

13.5.2 Adding Images to the App's Folder

From this chapter's examples folder, copy the `images` folder (which contains the `large` and `small` subfolders) into the folder where you'll save this app's FXML file, and the source-code files `CoverViewer.java` and `CoverViewerController.java`. Though you'll use only the `large` images in this example, you'll copy this app's folder to create the next example, which uses both sets of images.

13.5.3 Building the GUI

In this section, we’ll discuss the **Cover Viewer** app’s GUI. As you’ve done previously, create a new FXML file, then save it as `CoverViewer.fxml`.

fx:id Property Values for This App’s Controls

Figure 13.13 shows the **fx:id** properties of the **Cover Viewer** app’s programmatically manipulated controls. As you build the GUI, you should set the corresponding **fx:id** properties in the FXML document.

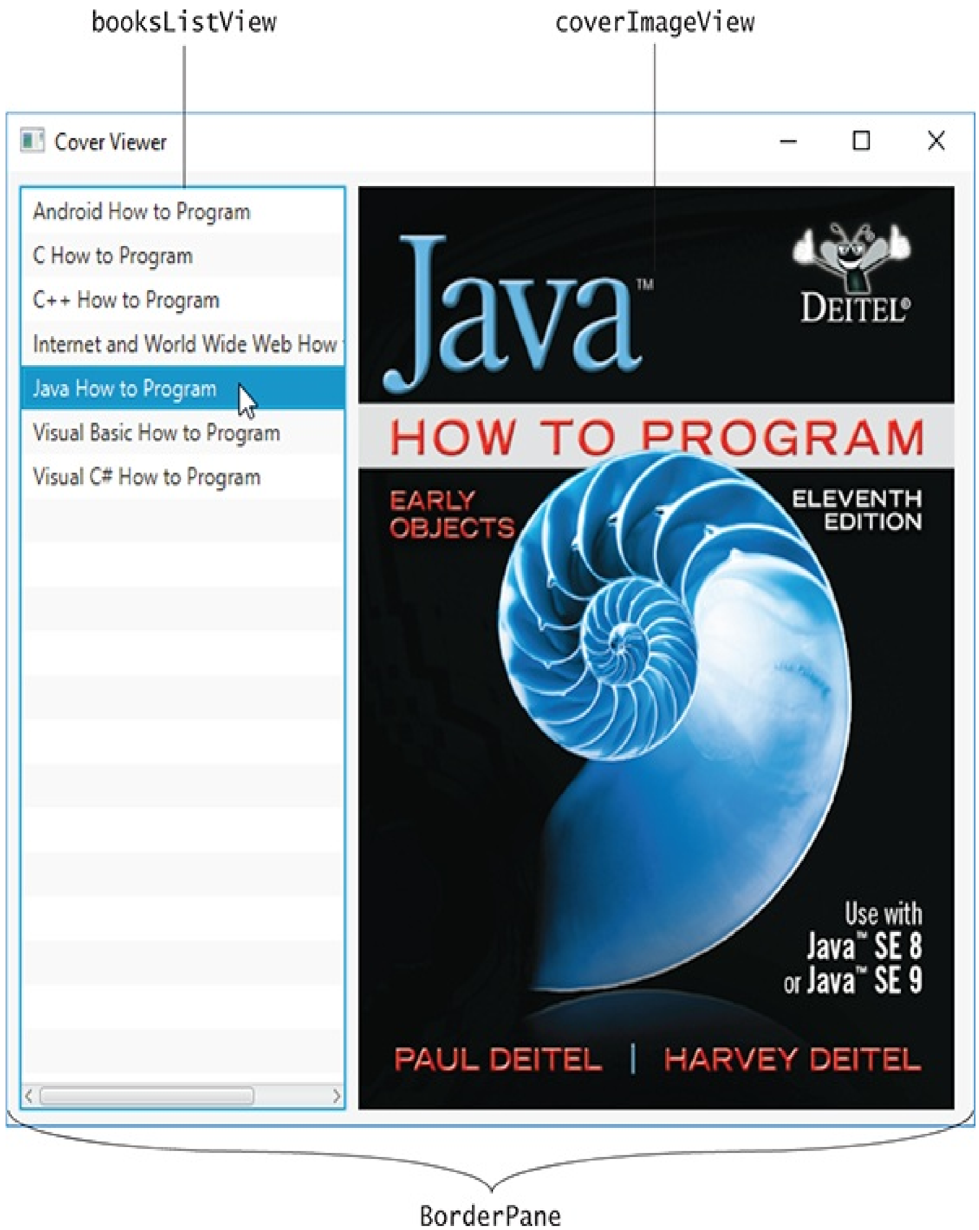


Fig. 13.13

Cover Viewer app's programmatically manipulated controls labeled with their **fx:id**s.

Adding and Configuring the Controls

Using the techniques you learned previously, create a `BorderPane`. In the left area, place a `ListView` control, and in the center area, place an `ImageView` control.

For the `ListView`, set the following properties:

- **Margin**—`8` (for the right margin) to separate the `ListView` from the `ImageView`
- **Pref Width**—`200`
- **Max Height**—`MAX_VALUE`
- **Min Width**, **Min Height**, **Pref Height** and **Max Width** —`USE_COMPUTED_SIZE`

For the `ImageView`, set the **Fit Width** and **Fit Height** properties to `370` and `480`, respectively. To size the `BorderPane` based on its contents, set its **Pref Width** and **Pref Height** to `USE_COMPUTED_SIZE`. Also, set the **Padding** property to `8` to inset the `BorderPane` from the stage.

Specifying the Controller Class's Name

To ensure that an object of the controller class is created when the app loads the FXML file at runtime, specify

`CoverViewerController` as the controller class's name in the FXML file as you've done previously.

Generating a Sample Controller Class

Select **View** > **Show Sample Controller Skeleton**, then copy this code into a `CoverViewerController.java` file and store the file in the same folder as `CoverViewer.fxml`. We show the completed `CoverViewerController` class in Section 13.5.5.

13.5.4 CoverViewer Subclass of `Application`

Figure 13.14 shows class `CoverViewer` subclass of `Application`.

```
// Fig. 13.13: CoverViewer.java
// Main application class that loads and display
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class CoverViewer extends Application {
   @Override
```

```
    public void start(Stage stage) throws Excepti
        Parent root =
            FXMLLoader.load(getClass().getResource(

        Scene scene = new Scene(root);
        stage.setTitle("Cover Viewer");
        stage.setScene(scene);
        stage.show();
    }

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

Fig. 13.14

Main application class that loads and displays the **Cover Viewer**'s GUI.

13.5.5 `CoverViewerController` Class

Figure 13.15 shows the final version of class `CoverViewerController` with the app’s new features highlighted.

```java
// Fig. 13.14: CoverViewerController.java
// Controller for Cover Viewer application
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

public class CoverViewerController {
   // instance variables for interacting with GU
   @FXML private ListView<Book> booksListView;
   @FXML private ImageView coverImageView;

   // stores the list of Book Objects
   private final ObservableList<Book>books =
      FXCollections.observableArrayList();

   // initialize controller
   public void initialize() {
      // populate the ObservableList<Book>
      books.add(new Book("Android How to Program
         "/images/small/androidhtp.jpg", "/image
      books.add(new Book("C How to Program",
         "/images/small/chtp.jpg", "/images/larg
      books.add(new Book("C++ How to Program",
         "/images/small/cpphtp.jpg", "/images/la
      books.add(new Book("Internet and World Wid
         "/images/small/iw3htp.jpg", "/images/la
      books.add(new Book("Java How to Program",
         "/images/small/jhtp.jpg", "/images/larg
      books.add(new Book("Visual Basic How to Pr
         "/images/small/vbhtp.jpg", "/images/lar
      books.add(new Book("Visual C# How to Progr
         "/images/small/vcshtp.jpg", "/images/la
      booksListView.setItems(books); // bind boo

      // when ListView selection changes, show l
```

```
41         booksListView.getSelectionModel().selected
42            addListener(
43               new ChangeListener<Book>() {
44                  @Override
45                  public void changed(ObservableVal
46                     Book oldValue, Book newValue)
47                     coverImageView.setImage(
48                        new Image(newValue.getLarge
49                  }
50               }
51            );
52      }
53   }
```

Fig. 13.15

Controller for **Cover Viewer** application.

@FXML Instance Variables

Lines 14–15 declare the controller’s `@FXML` instance variables. Notice that `ListView` is a generic class. In this case, the `ListView` displays `Book` objects. Class `Book` contains three `String` instance variables with corresponding *set* and *get* methods:

- `title`—the book’s title.
- `thumbImage`—the path to the book’s thumbnail image (used in the next example).
- `largeImage`—the path to the book’s large cover image.

The class also provides a `toString` method that returns the `Book`'s title and a constructor that initializes the three instance variables. You should copy class `Book` from this chapter's examples folder into the folder that contains `CoverViewer.fxml`, `CoverViewer.java` and `CoverViewerController.java`.

Instance Variable `books`

Lines 18–19 define the `books` instance variable as an `ObservableList<Book>` and initialize it by calling `FXCollections static` method `observableArrayList`. This method returns an empty collection object (similar to an `ArrayList`) that implements the `ObservableList` interface.

Initializing the `books` `ObservableList`

Lines 24–37 in method `initialize` create and add `Book` objects to the `books` collection. Line 38 passes this collection to `ListView` method `setItems`, which binds the `ListView` to the `ObservableList`. This *data binding* allows the `ListView` to display the `Book` objects automatically. By default, the `ListView` displays each `Book`'s `String` representation. (In the next example, you'll customize this.)

Listening for `ListView` Selection Changes

To synchronize the book cover that's being displayed with the currently selected book, we listen for changes to the `ListView`'s selected item. By default a `ListView` supports single selection—one item at a time may be selected. `ListView`s also support multiple selection. The type of selection is managed by the `ListView`'s `MultipleSelectionModel` (a subclass of `SelectionModel` from package `javafx.scene.control`), which contains observable properties and various methods for manipulating the corresponding `ListView`'s items.

To respond to selection changes, you register a listener for the `MultipleSelectionModel`'s `selectedItem` property (lines 41–51). `ListView` method `getSelectionModel` returns a `MultipleSelectionModel` object. In this example, `MultipleSelectionModel`'s`selectedItemProperty` method returns a `ReadOnlyObjectProperty<Book>`, and the corresponding `ChangeListener` receives as its `oldValue` and `newValue` parameters the previously selected and newly selected `Book` objects, respectively.

Lines 47–48 use `newValue`'s large image path to initialize a new `Image` (package `javafx.scene.image`)—this loads the image from that path. We then pass the new `Image` to the `coverImageView`’s `setImage` method to display the `Image`.

13.6 Cover Viewer App: Customizing `ListView` Cells

In the preceding example, the `ListView` displayed a `Book`’s `String` representation (i.e., its title). In this example, you’ll create a custom `ListView` cell factory to create cells that display each book as its thumbnail image and title using a `VBox`, an `ImageView` and a `Label` (Fig. 13.16).

Fig. 13.16

Cover Viewer app with Java How to Program selected.

Description

13.6.1 Technologies Overview

`ListCell` Generic Class for Custom `ListView` Cell Formats

As you saw in Section 13.5, `ListView` cells display the `String` representations of a `ListView`'s items by default. To create a custom cell format, you must first define a subclass of the `ListCell` generic class (package `javafx.scene.control`) that specifies how to create a `ListView` cell. As the `ListView` displays items, it gets `ListCell`s from its cell factory. You'll use the `ListView`'s `setCellFactory` method to replace the default cell factory with one that returns objects of the `ListCell` subclass. You'll override this class's `updateItem` method to specify the cells' custom layout and contents.

Programmatically Creating Layouts and Controls

So far, you've created GUIs visually using JavaFX Scene Builder. In this app, you'll also create a portion of the GUI programmatically—in fact, everything we've shown you in Scene Builder also can be accomplished in Java code directly. In particular, you'll create and configure a `VBox` layout containing an `ImageView` and a `Label`. The `VBox` represents the custom `ListView` cell format.

13.6.2 Copying the `CoverViewer` App

This app's FXML layout and classes `Book` and `CoverViewer` are identical to those in Section 13.5, and the `CoverViewerController` class has only one new statement. For this example, we'll show a new class that implements the custom `ListView` cell factory and the one new statement in class `CoverViewerController`. Rather than creating a new app from scratch, copy the `CoverViewer` app from the previous example into a new folder named `CoverViewerCustomListView`.

13.6.3 `ImageTextCell`

Custom Cell Factory Class

Class `ImageTextCell` (Fig. 13.17) defines the custom `ListView` cell layout for this version of the **Cover Viewer** app. The class extends `ListCell<Book>` because it defines a customized presentation of a `Book` in a `ListView` cell.

```
// Fig. 13.16: ImageTextCell.java
// Custom ListView cell factory that displays an
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.ListCell;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.scene.text.TextAlignment;

public class ImageTextCell extends ListCell<Book
   private VBox vbox = new VBox(8.0); // 8 point
   private ImageView thumbImageView = new ImageV
   private Label label = new Label();

   // constructor configures VBox, ImageView and
   public ImageTextCell() {
      vbox.setAlignment(Pos.CENTER); // center V

      thumbImageView.setPreserveRatio(true);
      thumbImageView.setFitHeight(100.0); // thu
      vbox.getChildren().add(thumbImageView); //

      label.setWrapText(true); // wrap if text t
      label.setTextAlignment(TextAlignment.CENTE
      vbox.getChildren().add(label); // attach t

      setPrefWidth(USE_PREF_SIZE); // use prefer
   }

```

```
31    // called to configure each custom ListView c
32    @Override
33    protected void updateItem(Book item, boolean
34       // required to ensure that cell displays p
35       super.updateItem(item, empty)
36
37       if (empty || item == null) {
38          setGraphic(null); // don't display anyt
39       }
40       else {
41          // set ImageView's thumbnail image
42          thumbImageView.setImage(new Image(item.
43          label.setText(item.getTitle()); // conf
44          setGraphic(vbox); // attach custom layo
45       }
46    }
47 }
```

Fig. 13.17

Custom `ListView` cell factory that displays an image and text.

Constructor

The constructor (lines 17–29) configures the instance variables we use to build the custom presentation. Line 18 indicates that the `VBox`'s children should be centered. Lines 20–22 configure the `ImageView` and attach it to the `VBox`'s collection of children. Line 20 indicates that the `ImageView` should preserve the image's aspect ratio, and line 21 indicates that the `ImageView` should be 100 points tall. Line 22 attaches the `ImageView` to the `VBox`.

Lines 24–26 configure the `Label` and attach it to the `VBox`'s collection of children. Line 24 indicates that the `Label` should wrap its text if its too wide to fit in the `Label`'s width, and line 25 indicates that the text should be centered in the `Label`. Line 26 attaches the `Label` to the `VBox`. Finally, line 28 indicates that the cell should use its preferred width, which is determined from the width of its parent `ListView`.

Method `updateItem`

Method `updateItem` (lines 32–46) configures the `Label`'s text and the `ImageView`'s `Image` then displays the custom presentation in the `ListView`. This method is called by the `ListView`'s cell factory when a `ListView` cell is required —that is, when the `ListView` is first displayed and when `ListView` cells are about to scroll onto the screen. The method receives the `Book` to display and a `boolean` indicating whether the cell that's about to be created is empty. You must call the superclass's version of `updateItem` (line 35) to ensure that the custom cells display correctly.

If the cell is empty or the item parameter is `null`, then there is no `Book` to display and line 38 calls the `ImageTextCell`'s inherited `setGraphic` method with `null`. This method receives as its argument the `Node` that should be displayed in the cell. Any JavaFX `Node` can be provided, giving you tremendous flexibility for customizing a cell’s appearance.

If there is a `Book` to display, lines 40–45 configure the `ImageTextCell`’s the `Label` and `ImageView`. Line 42 configures the `Book`’s `Image` and sets it to display in the `ImageView`. Line 43 sets the `Label`’s text to the `Book`’s title. Finally, line 38 uses method `setGraphic` to set the `ImageTextCell`’s `VBox` as the custom cell’s presentation.

Performance Tip 13.1

For the best `ListView` *performance, it’s considered best practice to define the custom presentation’s controls as instance variables in the* `ListCell` *subclass and configure them in the subclass’s constructor. This minimizes the amount of work required in each call to method* `updateItem`.

13.6.4 `CoverViewerController` Class

Once you’ve defined the custom cell layout, updating the `CoverViewerController` to use it requires that you set the `ListView`’s cell factory. Insert the following code as the last statement in the `CoverViewerController`’s `initialize` method:

```
booksListView.setCellFactory(
   new Callback<ListView<Book>, ListCell<Book>>() {
      @Override
      public ListCell<Book> call(ListView<Book> listV
         return new ImageTextCell();
      }
   }
);
```

and add an import for `javafx.util.Callback`.

The argument to `ListView` method `setCellFactory` is an implementation of the functional interface `CallBack` (package `javafx.util`). This generic interface provides a `call` method that receives one argument and returns a value. In this case, we implement interface `Callback` with an object of an anonymous inner class. In `Callback`'s angle brackets the first type (`ListView<Book>`) is the parameter type for the interface's `call` method and the second (`ListCell<Book>`) is the `call` method's return type. The parameter represents the `ListView` in which the custom cells will appear. The `call` method call simply creates and returns an object of the `ImageTextCell` class.

8

Each time the `ListView` requires a new cell, the anonymous inner class's `call` method will be invoked to get a new `ImageTextCell`. Then the `ImageTextCell`'s `update` method will be called to create the custom cell presentation. Note that by using a Java SE 8 lambda (Chapter 17) rather than an anonymous inner class, you can replace the entire statement that sets the cell factory with a single line of code.

13.7 Additional JavaFX Capabilities

This section overviews various additional JavaFX capabilities that are available in JavaFX 8 and JavaFX 9.

`TableView` Control

Section 13.5 demonstrated how to bind data to a `ListView` control. You often load such data from a database (Chapter 24, Accessing Databases with JDBC, and Chapter 29, Java Persistence API (JPA)). JavaFX's `TableView` control (package `javafx.scene.control`) displays tabular data in rows and columns, and supports user interactions with that data.

Accessibility

8

In a Java SE 8 update, JavaFX added *accessibility* features to help people with visual impairments use their devices. For example, the screen readers in various operating systems can speak screen text or text that you provide to help users with visual impairments understand the purpose of a control. Visually impaired users must enable their operating systems' screen-reading capabilities. JavaFX controls also support:

- GUI navigation via the keyboard—for example, the user can press the *Tab* key to jump from one control to the next. If a screen reader also is enabled, as the user moves the focus from control to control, the screen reader will speak appropriate information about each control (discussed below).
- A high-contrast mode to make controls more readable—as with screen readers, visually impaired users must enable this feature in their operating systems.

See your operating system's documentation for information on enabling its screen reader and high-contrast mode.

Every JavaFX `Node` subclass also has the following accessibility-related properties:

- `accessibleTextProperty`—A `String` that a screen reader speaks for a control. For example, a screen reader normally speaks the text displayed on a `Button`, but setting this property for a `Button` causes the screen reader to speak this property's text instead. You also can set this property to provide accessibility text for controls that do not have text, such as `ImageView`s.
- `accessibleHelpProperty`—A more detailed control description `String` than that provided by the `accessibleTextProperty`. This property's text should help the user understand the purpose of the control in the context of your app.
- `accessibleRoleProperty`—A value from the `enum AccessibleRole` (package `javafx.scene`). A screen reader uses this property value to determine the attributes and actions supported for a given control.
- `accessibleRoleDescriptionProperty`—A `String` text description of a control that a screen reader typically speaks followed by the control's contents (such as the text on a `Button`) or the value of the

`accessibleTextProperty`.

In addition, you can add `Label`s to a GUI that describe other controls. In such cases, you should set each `Label`'s `labelFor` property to the specific control the `Label` describes. For example, a `TextField` in which the user can enter a phone number might be preceded by a `Label` containing the text `"Phone Number"`. If the `Label`'s `labelFor` property references the `TextField`, then a screen reader will read the `Label`'s text as well when describing the `TextField` to the user.

Third-Party JavaFX Libraries

JavaFX continues to become more popular. There are various open-source, third-party libraries, which define additional JavaFX capabilities that you can incorporate into your own apps. Some popular JavaFX libraries include:

- ControlsFX (`http://www.controlsfx.org/`) provides common dialogs, additional controls, validation capabilities, `TextField` enhancements, a `SpreadSheetView`, `TableView` enhancements and more. You can find the API documentation at `http://docs.controlsfx.org/` and various code samples at `http://code.controlsfx.org`. We use one of the open-source ControlsFX dialogs in Chapter 22.
- JFXtras (`http://jfxtras.org/`) also provides many additional JavaFX controls, including date/time pickers, controls for maintaining an agenda, a calendar control, additional window features and more.

- Medusa provides many JavaFX gauges that look like clocks, speedometers and more. You can view samples at `https://github.com/HanSolo/Medusa/blob/master/README.md`.

Creating Custom JavaFX Controls

You can create custom controls by extending existing JavaFX control classes to customize them or by extending JavaFX’s `Control` class directly.

JavaFXPorts: JavaFX for Mobile and Embedded Devices

A key Java benefit is writing apps that can run on any device with a Java Virtual Machine (JVM), including notebook computers, desktop computers, servers, mobile devices and embedded devices (such as those used in the Internet of Things). Oracle officially supports JavaFX only for desktop apps. Gluon’s open-source JavaFXPorts project brings the desktip version of JavaFX to mobile devices (iOS and Android) and devices like the inexpensive Raspberry Pi (`https://www.raspberrypi.org/`), which can be used as a standalone computer or for embedded-device applications. For more information on JavaFXPorts, visit

```
http://javafxports.org/
```

In addition, Gluon Mobile provides a mobile-optimized JavaFX implementation for iOS and Android. For more information, see

```
http://gluonhq.com/products/mobile/
```

Scenic View for Debugging JavaFX Scenes and Nodes

Scenic View is a debugging tool for JavaFX scenes and nodes. You embed **Scenic View** directly into your apps or run it as a standalone app. You can inspect your JavaFX scenes and nodes, and modify them dynamically to see how changes affect their presentation on the screen—without having to edit your code, recompile it and re-run it for each change. For more information, visit

```
http://www.scenic-view.org
```

JavaFX Resources and JavaFX in the Real World

Visit

```
http://bit.ly/JavaFXResources
```

for a lengthy and growing list of JavaFX resources that includes links to:

- articles
- tutorials (free and for purchase)
- key blogs and websites
- YouTube® videos
- books (for purchase)
- many libraries, tools, projects and frameworks
- slide shows from JavaFX presentations and
- various real-world examples of JavaFX in use.

13.8 JavaFX 9: Java SE 9 JavaFX Updates

This section overviews several JavaFX 9 changes and enhancements.

Java SE 9 Modularization

9

Java SE 9's biggest new software-engineering feature is the module system. This applies to JavaFX 9 as well. The key JavaFX 9 modules are:

- `javafx.base`—Contains the packages required by all JavaFX 9 apps. All the other JavaFX 9 modules depend on this one.
- `javafx.controls`—Contains the packages for controls, layouts and charts, including the various controls we demonstrated in this chapter and Chapter 12.
- `javafx.fxml`—Contains the packages for working with FXML, including the FXML features we demonstrated in this chapter and Chapter 12.
- `javafx.graphics`—Contains the packages for working with graphics, animation, CSS (for styling nodes), text and more (Chapter 22, JavaFX Graphics and Multimedia).
- `javafx.media`—Contains the packages for incorporating audio and video (Chapter 22, JavaFX Graphics and Multimedia).

- `javafx.swing`—Contains the packages for integrating into JavaFX 9 apps Swing GUI components (Chapter 26, Swing GUI Components: Part 1, and Chapter 35, Swing GUI Components: Part 2).
- `javafx.web`—Contains the package for integrating web content.

In your apps, if you use modularization and JDK 9, only the modules required by your app will be loaded at runtime. Otherwise, your app will continue to work as it did previously, provided that you did not use so-called internal APIs—that is, undocumented Java APIs that are not meant for public use. In the modularized JDK 9, such APIs are automatically *private* and inaccessible to your apps—any code that depends on pre-Java-SE-9 internal APIs will not compile. We discuss modularization in more detail in our online Java SE 9 treatment. See the Preface for details.

New Public Skinning APIs

9

In Chapter 22, JavaFX Graphics and Multimedia, we demonstrate how to format JavaFX objects using a technology called *Cascading Style Sheets (CSS)* that was originally developed for styling the elements in web pages. As you'll see, CSS allows you to specify *presentation* (e.g., fonts, spacing, sizes, colors, positioning) separately from the GUI's *structure* and *content* (layout containers, shapes, text, GUI components, etc.). If a JavaFX GUI's presentation is determined entirely by a style sheet (which specifies the rules for styling the GUI), you can simply swap in a new style sheet

—sometimes called a **skin**—to change the GUI's appearance. This is commonly called **skinning**.

Each JavaFX control also has a skin class that determines its default appearance. In JavaFX 8, skin classes are defined as internal APIs, but many developers create custom skins by extending these skin classes. In JavaFX 9, the skin classes are now public APIs in the package `javafx.scene.control.skin`. You can extend the appropriate skin class to customize the look-and-feel for a given type of control. You then create an object of your custom skin class and set it for a control via its `setSkin` method.

GTK+ 3 Support on Linux

9

GTK+ (GIMP Toolkit—`http://gtk.org`) is a GUI toolkit that JavaFX uses behind the scenes to render GUIs and graphics on Linux. In Java SE 9, JavaFX now supports GTK+ 3—the latest version of GTK+.

High-DPI Screen Support

9

In a Java SE 8 update, JavaFX added support for High DPI (dots-per-inch) screens on Windows and macOS. Java SE 9 adds Linux High-DPI support, as well as capabilities to programmatically manipulate the scale at which JavaFX apps are rendered on Windows, macOS and Linux.

Updated GStreamer

9

JavaFX implements its audio and video multimedia capabilities using the open-source GStreamer framework (`https://gstreamer.freedesktop.org`). JavaFX 9 incorporates a more recent version of GStreamer with various bug fixes and performance enhancements.

Updated WebKit

9

JavaFX's `WebView` control enables you to embed web content in your JavaFX apps. `WebView` is based on the open source WebKit framework (`http://www.webkit.org`)—a web browser engine that supports loading and rendering web pages. JavaFX 9 incorporates an updated version of WebKit.

13.9 Wrap-Up

In this chapter, we continued our presentation of JavaFX. We discussed JavaFX layout panes in more detail and used `BorderPane`, `TitledPane` and `Pane` to arrange controls.

You learned about the many mouse events supported by JavaFX nodes, and we used the `onMouseDragged` event in a simple **Painter** app that displayed `Circle`s as the user dragged the mouse across an `Pane`. The **Painter** app allowed the user to choose the current color and pen size from groups of mutually exclusive `RadioButton`s. You used `ToggleGroup`s to manage the relationship between the `RadioButton`s in each group. You also learned how to provide a so-called user data `Object` for a control. When a `RadioButton` was selected, you obtained it from the `ToggleGroup`, then accessed the `RadioButton`'s user data `Object` to determine the drawing color or pen size.

We discussed property binding and property listeners, then used them to implement a **Color Chooser** app. You bound a `TextField`'s text to a `Slider`'s value to automatically update the `TextField` when the user moved the `Slider`'s thumb. You also used a property listener to allow the app's controller to update the color of a `Rectangle` when a `Slider`'s value changed.

In our **Cover Viewer** app, we showed how to bind an `ObservableList` collection to a `ListView` control to populate it with the collection's elements. By default, each object in the collection was displayed as a `String` in the `ListView`. You configured a property listener to display an image in an `ImageView` when the user selected an item in the `ListView`. We modified the **Cover Viewer** app to use a custom `ListView` cell factory to specify the exact layout of a `ListView` cell's contents. Finally, we introduced several other JavaFX capabilities and the Java SE 9 changes to JavaFX.

In the next chapter, we discuss class `String` and its methods. We introduce regular expressions for pattern matching in strings and demonstrate how to validate user input with regular expressions.

Summary

Section 13.2 Laying Out Nodes in a Scene Graph

- A layout determines the size and positioning of nodes in the scene graph.
- In general, a node's size should not be defined explicitly.
- In addition to the `width` and `height` properties associated with every JavaFX node, most JavaFX nodes have the properties `prefWidth`, `prefHeight`, `minWidth`, `minHeight`, `maxWidth` and `maxHeight` that specify a node's *range* of acceptable sizes as it's laid out within its parent node.
- The minimum size properties specify a node's smallest allowed size in points.
- The maximum size properties specify a node's largest allowed size in points.
- The preferred size properties specify a node's preferred width and height that should be used by a layout in most cases.
- A node's position should be defined relative to its parent node and the other nodes in its parent.
- Layout panes are container nodes that arrange their child nodes in a scene graph relative to one another, based on their sizes and positions.
- Most JavaFX layout panes use relative positioning.

Section 13.3.1 Technologies Overview

- `RadioButton`s function as mutually exclusive options.
- You add multiple `RadioButton`s to a `ToggleGroup` to ensure that only one `RadioButton` in a given group is selected at a time.
- If you programmatically create a `ToggleGroup` (rather than declaring it in FXML), you can call `RadioButton`'s `setToggleGroup` method to specify its `ToggleGroup`.
- A `BorderPane` layout container arranges controls into one or more of five regions—top, right, bottom, left and center. The top and bottom areas have the same width as the `BorderPane`. The left, center and right areas fill the vertical space between the top and bottom areas. Each area may contain only one control or one layout container that, in turn, may contain other controls.
- All the areas in a `BorderPane` are optional: If the top or bottom area is empty, the left, center and right areas expand vertically to fill that area. If the left or right area is empty, the center expands horizontally to fill that area.
- A `TitledPane` displays a title at its top and is a collapsible panel containing a layout node, which in turn contains other nodes.
- The `javafx.scene.shape` package contains various classes for creating 2D and 3D shape nodes that can be displayed in a scene graph.
- Nodes are attached to an `Pane` layout at a specified *x*-*y* coordinate measured from the `Pane`'s upper-left corner.
- JavaFX nodes support various mouse events.
- JavaFX supports other types of input events, such as touch-oriented events and key events.

- Each JavaFX control has a `setUserData` method that receives an `Object`. You can use this to store any object you'd like to associate with that control—typically this `Object` is used when responding to the control's events.

Section 13.3.2 Creating the `Painter.fxml` File

- If you already have an FXML file open in Scene Builder, you can choose **File > New** to create a new FXML file, then save it.

Section 13.3.3 Building the GUI

- A `VBox`'s **Spacing** property specifies vertical spacing between its controls.
- Setting a node's **Max Height** property to `MAX_VALUE` enables the node to occupy the full height of its parent node.
- The **Style** `-fx-background-color` specifies a node's background color.
- A `TitledPane`'s **Text** property specifies the title at the top of the `TitledPane`.
- A `RadioButton`'s **Text** property specifies the text that appears next to the `RadioButton`.
- A `RadioButton`'s **Selected** property specifies whether the `RadioButton` is selected.
- Setting a `RadioButton`'s **Toggle Group** property in FXML adds the `RadioButton` to that `ToggleGroup`.
- Setting a control's **Max Width** property to `MAX_VALUE` enables the control to fill its parent node's width.
- A control's **On Mouse Dragged** event handler (located under the **Mouse** heading in the **Code** section) specifies what to do when the user drags the mouse on the control.
- To specify what to do when a user interacts with a `RadioButton`, set its **On Action** event handler.
- To specify what to do when a user interacts with a `Button`, set its **On Action** event handler.

Section 13.3.5 `PainterController` Class

- Top-level types are not declared inside another type.
- Java allows you to declare classes, interfaces and `enum`s inside other classes—these are called nested types.
- Each mouse event handler you define must provide one parameter of type `MouseEvent` (package `javafx.scene.input`). When the event occurs, this parameter contains information about the event, such as its location, whether any mouse buttons were pressed, which node the user interacted with and more.
- Each layout pane has a `getChildren` method that returns an `ObservableList<Node>` collection containing the layout's child nodes. An `ObservableList` provides methods for adding and removing elements.
- All JavaFX shapes inherit indirectly from class `Node` in the `javafx.scene` package.
- `ToggleGroup` method `getSelectedToggle` returns the `Toggle` that's currently selected. Class `RadioButton` is one of several controls (others are `RadioButtonMenuItem` and `ToggleButton`) that implements interface `Toggle`.
- `Toggle`'s `getUserData` method gets the user data `Object` that's associated with a control.

Section 13.4.1 Technologies Overview

- In the RGBA color system, every color is represented by its red, green and blue color values, each ranging from 0 to 255, where 0 denotes no color and 255 full color. The alpha value (A)—which ranges from 0.0 to 1.0—represents a color's opacity, with 0.0 being completely transparent and 1.0 completely opaque.
- A property is defined by creating *set* and *get* methods with specific naming conventions. Typically, such methods manipulate a corresponding `private` instance variable that has the same name as the property, but this is not required. If the property represents a `boolean` value, its *get* method name typically begins with "`is`" rather than "`get`."
- JavaFX properties are observable—when a property's value changes, other objects can respond accordingly.
- One way to respond to a property change is via a property binding, which enables a property of one object to be updated when a property of another object changes.
- Property bindings are not limited to JavaFX controls. Package `javafx.beans.property` contains many classes that you can use to define bindable properties in your own classes.
- A property listener is an event handler that's invoked when a property's value changes. In the event handler, you can respond to the property change in a manner appropriate for your app.

Section 13.4.2 Building the GUI

- `Circle` and `Rectangle` are located in the Scene Builder **Library**’s **Shapes** section.

Section 13.4.4 `ColorChooserController` Class

- A controller class's `initialize` method often configures property bindings and property listeners.
- Each `TextField` has a `text` property that's returned by its `textProperty` method as a `StringProperty` (package `javafx.beans.property`).
- `StringProperty` method `bind` receives an `ObservableValue` as an argument. When the `ObservableValue` changes, the bound property is updated accordingly.
- `Slider` method `valueProperty` returns a `Slider`'s `value` property as an object of class `DoubleProperty`—an observable `double` value.
- `DoubleProperty` method `asString` returns a `StringBinding` object (which is an `ObservableValue`) that produces a `String` representation of the `DoubleProperty`.
- To perform an arbitrary task when a property's value changes, you can register a property listener.

Section 13.5 Cover Viewer App: Data-Driven GUIs with JavaFX Collections

- JavaFX provides a comprehensive model for allowing GUIs to interact with data.

Section 13.5.1 Technologies Overview

- A `ListView` control displays a collection of objects.
- Though you can individually add items to a `ListView`, you'll often bind an `ObservableList` object to the `ListView`.
- If you make changes to an `ObservableList`, its observer (such as a `ListView`) will automatically be notified of those changes.
- Package `javafx.collections` defines `ObservableList` (similar to an `ArrayList`) and other observable collection interfaces.
- Class `FXCollections` provides `static` methods for creating and manipulating observable collections.

Section 13.5.5 `CoverViewerController` Class

- `FXCollections static` method `observableArrayList` returns an empty collection object (similar to an `ArrayList`) that implements the `ObservableList` interface.
- `ListView` method `setItems` receives an `ObservableList` and binds the `ListView` to it. This data binding allows the `ListView` to display the `ObservableList`'s objects automatically—as `String`s by default.
- By default a `ListView` supports single selection—one item at a time may be selected. `ListView`s also support multiple selection. The type of selection is managed by the `ListView`'s `MultipleSelectionModel` (a subclass of `SelectionModel` from package `javafx.scene.control`), which contains observable properties and various methods for manipulating the corresponding `ListView`'s items.
- To respond to selection changes, register a listener for the `MultipleSelectionModel`'s `selectedItem` property.
- `ListView` method `getSelectionModel` returns a `MultipleSelectionModel` object.
- `MultipleSelectionModel`'s `selectedItemProperty` method returns a `ReadOnlyObjectProperty`, and the corresponding `ChangeListener` receives as its `oldValue` and `newValue` parameters the previously selected and newly selected objects, respectively.

Section 13.6.1 Technologies Overview

- To create a custom `ListView` cell format, you must first define a subclass of the `ListCell` generic class (package `javafx.scene.control`) that specifies how to create a `ListView` cell.
- As the `ListView` displays items, it gets `ListView` cells from its cell factory.
- You'll use the `ListView`'s `setCellFactory` method to replace the default cell factory with one that returns objects of the `ListCell` subclass. You override this class's `updateItem` method to specify the cells' custom layout and contents.
- Everything you can do in Scene Builder also can be accomplished in Java code.

Section 13.6.3 `ImageTextCell` Custom Cell Factory Class

- A custom `ListView` cell layout is defined as a subclass of `ListCell`<*Type*>, where *Type* is the type of the object displayed in a `ListView` cell.
- The `ListCell`<*Type*> subclass's `updateItem` method creates the custom presentation. This method is called by the `ListView`'s cell factory when a `ListView` cell is required—that is, when the `ListView` is first displayed and when `ListView` cells are about to scroll onto the screen.
- Method `updateItem` receives the object to display and a `boolean` indicating whether the cell that's about to be created is empty. You must call the superclass's version of `updateItem` to ensure that the custom cells display correctly.
- `ListCell`<*Type*>'s `setGraphic` method receives a JavaFX `Node` representing the customized cell's appearance.

Section 13.6.4 `CoverViewerController` Class

- Once you’ve defined the custom cell layout, you must set the `ListView`’s cell factory.
- The argument to `ListView` method `setCellFactory` is an implementation of interface `CallBack` (package `javafx.util`). This generic interface provides a `call` method that receives one argument and returns an object of the custom `ListCell`<*Type*> subclass.

Section 13.7 Additional JavaFX Capabilities

- JavaFX's `TableView` control (package `javafx.scene.control`) displays tabular data in rows and columns, and supports user interactions with that data.
- In a Java SE 8 update, JavaFX added accessibility features to help people with visual impairments use their devices. These features include screen-reader support, GUI navigation via the keyboard and a high-contrast mode to make controls more readable.
- See your operating system's documentation for information on enabling its screen reader and high-contrast mode.
- Every JavaFX `Node` subclass also has accessibility-related properties.
- The `accessibleTextProperty` is a `String` that a screen reader speaks for a control.
- The `accessibleHelpProperty` is a more detailed control description `String` than that provided by the `accessibleTextProperty`. This property's text should help the user understand the purpose of the control in the context of your app.
- The `accessibleRoleProperty` is a value from the `enum AccessibleRole` (package `javafx.scene`). A screen reader uses this property value to determine the attributes and actions supported for a given control.
- The `accessibleRoleDescriptionProperty` is a `String` text description of a control that a screen reader typically speaks followed by the control's contents or the value of the `accessibleTextProperty`.
- You can add `Label`s to a GUI that describe other controls. In such cases, you should set each `Label`'s `labelFor` property to the specific control the `Label` describes. If a `Label`'s `labelFor` property references

another control, a screen reader will read the `Label`'s text when describing the control to the user.

- You can create custom controls by extending existing JavaFX control classes to customize them or by extending JavaFX's `Control` class directly.
- **Scenic View** (`http://www.scenic-view.org/`) is a debugging tool for JavaFX scenes and nodes. You embed **Scenic View** directly into your apps or run it as a standalone app. You can inspect your JavaFX scenes and nodes, and modify them dynamically to see how changes affect their presentation on the screen—without having to edit your code, recompile it and re-run it for each change.

Section 13.8 JavaFX 9: Java SE 9 JavaFX Updates

- JavaFX 9's biggest new feature is modularization.
- In your apps, if you use modularization and JDK 9, only the modules required by your app will be loaded at runtime. Otherwise, your app will continue to work as it did previously, provided that you did not use so-called internal APIs—that is, undocumented Java APIs that are not meant for public use.
- If a JavaFX GUI's presentation is determined entirely by a style sheet (which specifies the rules for styling the GUI), you can simply swap in a new style sheet—sometimes called a skin—to change the GUI's appearance. This is commonly called skinning.
- Each JavaFX control also has a skin class that determines its default appearance.
- In JavaFX 8, skin classes are defined as internal APIs. In JavaFX 9, the skin classes are now public APIs in the package `javafx.scene.control.skin`.
- You can extend the appropriate skin class to customize the look-and-feel for a given type of control. You then create an object of your custom skin class and set it for a control via its `setSkin` method.
- JavaFX 9 supports GTK+ 3—the latest version of GTK+.
- In a Java SE 8 update, JavaFX added support for High-DPI (dots-per-inch) screens on Windows and macOS. Java SE 9 adds Linux High-DPI support.
- JavaFX 9 adds features to programmatically manipulate the scale at which JavaFX apps are rendered on Windows, macOS and Linux.
- JavaFX implements its audio and video multimedia capabilities using the open-source GStreamer framework (`https://gstreamer.freedesktop.org`). JavaFX 9 incorporates

a more recent version of GStreamer with various bug fixes and performance enhancements.

- JavaFX's `WebView` control enables you to embed web content in your JavaFX apps. `WebView` is based on the open source WebKit framework (`http://www.webkit.org`)—a web browser engine that supports loading and rendering web pages. JavaFX 9 incorporates an updated version of WebKit.

22 JavaFX Graphics and Multimedia

Objectives

In this chapter you'll:

- Use JavaFX graphics and multimedia capabilities to make your apps "come alive" with graphics, animations, audio and video.
- Use external Cascading Style Sheets to customize the look of `Node`s while maintaining their functionality.
- Customize fonts attributes such as font family, size and style.
- Display two-dimensional shape nodes of types `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Path`, `Polyline` and `Polygon`.
- Customize the stroke and fill of shapes with solid colors, images and gradients.
- Use `Transform`s to reposition and reorient nodes.
- Display and control video playback with `Media`, `MediaPlayer` and `MediaView`.
- Animate `Node` properties with `Transition` and `Timeline` animations.
- Use an `AnimationTimer` to create frame-by-frame animations.
- Draw graphics on a `Canvas` node.
- Display 3D shapes.

Outline

22.1 Introduction

In this chapter, we continue our discussion of JavaFX from Chapters 12 and 13. Here, we present various JavaFX graphics and multimedia capabilities. You'll:

- Use external Cascading Style Sheets (CSS) to customize the appearance of JavaFX nodes.
- Customize fonts and font attributes used to display text.
- Display two-dimensional shapes, including lines, rectangles, circles, ellipses, arcs, polylines, polygons and custom paths.
- Apply transforms to `Node`s, such as rotating a `Node` around a particular point, scaling, translating (moving) and more.
- Display video and control its playback (e.g., play, pause, stop, and skip to specific time).
- Animate JavaFX `Node`s with `Transition` and `Timeline` animations that change `Node` property values over time. As you'll see, the built-in `Transition` animations change specific JavaFX `Node` properties (such as a `Node`'s stroke and fill colors), but `Timeline` animations can be used to change *any* modifiable `Node` property.
- Create frame-by-frame animations with an `AnimationTimer`.
- Draw two-dimensional graphics on a `Canvas` `Node`.
- Display three-dimensional shapes, including boxes, cylinders and spheres.

Throughout this chapter, we do not show each example's `Application` subclass, because it performs the same tasks we demonstrated in Chapters 12 and 13. Also, some examples do not have controller classes because they simply display

JavaFX controls or graphics to demonstrate CSS capabilities.

Project Exercises

At the end of this chapter, we provide dozens of project exercises that you’ll find challenging and hopefully entertaining. These will reinforce techniques you’ve learned and encourage you to investigate additional JavaFX graphics and multimedia capabilities in Oracle’s online JavaFX documentation. The **Block Breaker**, **SpotOn**, **Horse Race**, **Cannon** and other exercises will give you experience with game-programming fundamentals.

22.2 Controlling Fonts with Cascading Style Sheets (CSS)

In Chapters 12–13, you built JavaFX GUIs using Scene Builder. You specified a particular JavaFX object's appearance by selecting the object in Scene Builder, then setting its property values in the **Properties** inspector. With this approach, if you want to change the GUI's appearance, you must edit each object. If you have a large GUI in which you want to make the same changes to multiple objects, this can be time consuming and error prone.

In this chapter, we format JavaFX objects using a technology called **Cascading Style Sheets (CSS)** that's typically used to style the elements in web pages. CSS allows you to specify *presentation* (e.g., fonts, spacing, sizes, colors, positioning) separately from the GUI's *structure* and *content* (layout containers, shapes, text, GUI components, etc.). If a JavaFX GUI's presentation is determined entirely by CSS rules, you can simply swap in a new style sheet to change the GUI's appearance.

In this section, you'll use CSS to specify the font properties of several `Label`s and the spacing and padding properties for the `VBox` layout that contains the `Label`s. You'll place **CSS**

rules that specify the font properties, spacing and padding in a separate file that ends with the `.css` **filename extension**, then reference that file from the FXML. As you'll see,

- before referencing the CSS file from the FXML, Scene Builder displays the GUI without styling, and
- after referencing the CSS file from the FXML, Scene Builder renders the GUI with the CSS rules applied to the appropriate objects.

For a complete reference that shows

- all the JavaFX CSS properties,
- the JavaFX `Node` types to which the attributes can be applied, and
- the allowed values for each attribute

visit:

```
https://docs.oracle.com/javase/8/javafx/api/javafx/sc
```

22.2.1 CSS That Styles the GUI

Figure 22.1 presents this app's CSS rules that specify the `VBox`'s and each `Label`'s style. This file is located in the same folder as the rest of the example's files.

`.vbox` CSS Rule—Style Class Selectors

Lines 4–7 define the `.vbox` CSS rule that will be applied to this app’s `VBox` object (lines 8–18 of Fig. 22.2). Each CSS rule begins with a **CSS selector** which specifies the JavaFX objects that will be styled according to the rule. In the `.vbox` CSS rule, `.vbox` is a **style class selector**. The CSS properties in this rule are applied to any JavaFX object that has a `styleClass` property with the value `"vbox"`. In CSS, a style class selector begins with a dot (`.`) and is followed by its **class name** (not to be confused with a Java class). By convention, selector names typically have all lowercase letters, and multi-word names separate each word from the next with a dash (`-`).

```
1   /* Fig. 22.1: FontsCSS.css */
2   /* CSS rules that style the VBox and Labels */
3
4   .vbox {
5      -fx-spacing: 10;
6      -fx-padding: 10;
7   }
8
9   #label1 {
10     -fx-font: bold 14pt Arial;
11  }
12
13  #label2 {
14     -fx-font: 16pt "Times New Roman";
15  }
16
17  #label3 {
```

```
18      -fx-font: bold italic 16pt "Courier New";
19   }
20
21   #label4 {
22      -fx-font-size: 14pt;
23      -fx-underline: true;
24   }
25
26   #label5 {
27      -fx-font-size: 14pt;
28   }
29
30   #label5 .text {
31      -fx-strikethrough: true;
32   }
```

Fig. 22.1

CSS rules that style the `VBox` and `Label`s.

Each CSS rule's body is delimited by a set of required braces (`{}`) containing the CSS properties that are applied to objects matching the CSS selector. Each JavaFX CSS property name begins with `-fx-`[1] followed by the name of the corresponding JavaFX object's property in all lowercase letters. So, `-fx-spacing` in line 5 of Fig. 22.1 defines the value for a JavaFX object's `spacing` property, and `-fx-padding` in line 6 defines the value for a JavaFX object's `padding` property. The value of each property is specified to the right of the required colon (`:`). In this case, we set `-fx-spacing` to `10` to place 10 pixels of vertical space between objects in the

VBox, and -fx-padding to 10 to separate the VBox's contents from the VBox's edges by 10 pixels at the top, right, bottom and left edges. You also can specify the -fx-padding with four values separated by spaces. For example,

1. According to the JavaFX CSS Reference Guide at https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html, JavaFX CSS property names are designed to be processed in style sheets that may also contain HTML CSS. For this reason, JavaFX's CSS property names are prefixed with "-fx-" to ensure that they have distinct names from their HTML CSS counterparts.

```
-fx-padding: 10 5 10 5
```

specifies 10 pixels for the top padding, 5 for the right, 10 for the bottom and 5 for the left. We show how to apply the .vbox CSS rule to the VBox object in Section 22.2.2.

#label1 CSS Rule—ID Selectors

Lines 9–11 define the #label1 CSS rule. Selectors that begin with # are known as **ID selectors**—they are applied to objects with the specified ID. In this case, the #label1 selector matches the object with the fx:id label1—that is, the Label object in line 12 of Fig. 22.2. The #label1 CSS rule specifies the CSS property

```
-fx-font: bold 14pt Arial;
```

This rule sets an object's font property. The object to which this rule applies displays its text in a bold, 14-point, Arial font. The `-fx-font` property can specify all aspects of a font, including its style, weight, size and font family—the size and font family are required. There are also properties for setting each font component: `-fx-font-style`, `-fx-font-weight`, `-fx-font-size` and `-fx-font-family`. These are applied to a JavaFX object's similarly named properties. For more information on specifying CSS font attributes, see

```
https://docs.oracle.com/javase/8/javafx/api/javafx/sc
```

For a complete list of CSS selector types and how you can combine them, see

```
https://www.w3.org/TR/css3-selectors/
```

`#label2` CSS Rule

Lines 13–15 define the `#label2` CSS rule that will be applied to the `Label` with the `fx:id label2`. The CSS property

```
-fx-font: 16pt "Times New Roman";
```

specifies only the required font size (`16pt`) and font family

(`"Times New Roman"`) components—font family names with multiple words must be enclosed in double quotes.

`#label3` CSS Rule

Lines 17–19 define the `#label3` CSS rule that will be applied to the `Label` with the `fx:id label3`. The CSS property

```
-fx-font: bold italic 16pt "Courier New";
```

specifies all the font components—weight (`bold`), style (`italic`), size (`16pt`) and font family (`"Courier New"`).

`#label4` CSS Rule

Lines 21–24 define the `#label4` CSS rule that will be applied to the `Label` with the `fx:id label4`. The CSS property

```
-fx-font-size: 14pt;
```

specifies the font size `14pt`—all other aspects of this `Label`'s font are inherited from the `Label`'s parent container. The CSS property

```
-fx-underline: true;
```

indicates that the text in the `Label` should be *underlined*—the default value for this property is `false`.

`#label5` CSS Rule

Lines 26–28 define the `#label5` CSS rule that will be applied to the `Label` with the `fx:id label5`. The CSS property

```
-fx-font-size: 14pt;
```

specifies the font size `14pt`.

`#label5 .text` CSS Rule

Lines 30–32 define the `#label5 .text` CSS rule that will be applied to the `Text` object within the `Label` that has the `fx:id` value `"label5"`. The selector in this case is a combination of an ID selector and a style class selector. Each `Label` contains a `Text` object with the CSS class `.text`. When applying this CSS rule, JavaFX first locates the object with the ID `label5`, then within that object looks for a nested object that specifies the class `text`.

The CSS property

```
-fx-strikethrough: true;
```

indicates that the text in the `Label` should be displayed with a line through it—the default value for this property is `false`.

22.2.2 FXML That Defines the GUI—Introduction to XML Markup[2]

2. In many of this chapter's examples, after creating a GUI in Scene Builder, we used a text editor to format the FXML, remove unnecessary properties that were inserted by Scene Builder and properties that we specified via CSS rules. For this reason, when you build these examples from scratch, your FXML may differ from what's shown in this chapter. You also can set a property to its default value in Scene Builder to remove it from the FXML.

Figure 22.2 shows the contents of `FontCSS.fxml`—the `FontCSS` app's FXML GUI, which consists of a `VBox` layout element (lines 8–18) containing five `Label` elements (lines 12–16). When you first drag five `Label`s onto the `VBox` and configure their text (Fig. 22.2(a)), all the `Label`s initially have the same appearance in Scene Builder. Also, initially there's no spacing between and around the `Label`s in the `VBox`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 22.2: FontCSS.fxml -->
3  <!-- FontCSS GUI that is styled via external CSS
```

```

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>

<VBox styleClass="vbox" stylesheets="@FontCSS.css
      xmlns="http://javafx.com/javafx/8.0.60"
      xmlns:fx="http://javafx.com/fxml/1">
    <children>
        <Label fx:id="label1" text="Arial 14pt bol
        <Label fx:id="label2" text="Times New Roma
        <Label fx:id="label3" text="Courier New 16
        <Label fx:id="label4" text="Default font 1
        <Label fx:id="label5" text="Default font 1
    </children>
</VBox>
```

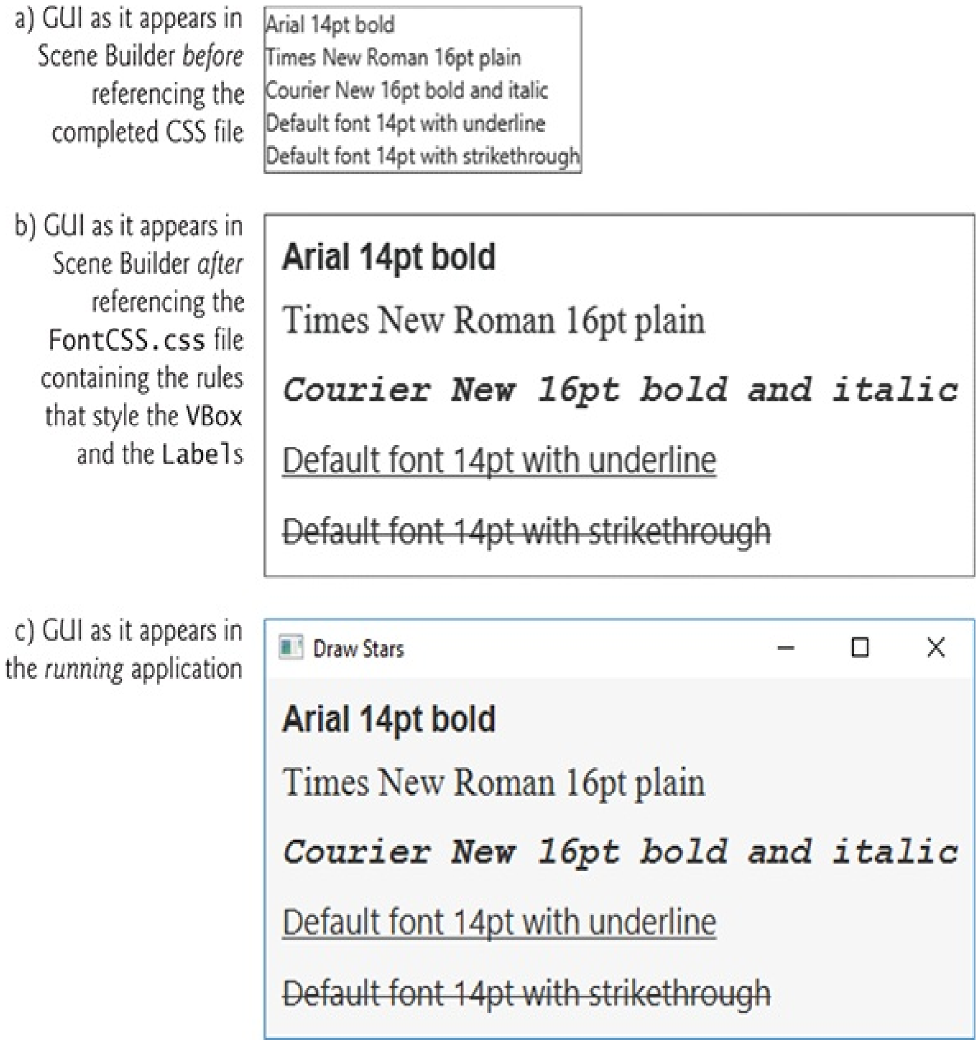


Fig. 22.2

`FontCSS` GUI that is styled via external CSS.

Description

XML Declaration

Each FXML document begins with an **XML declaration** (line

1), which must be the first line in the file and indicates that the document contains XML markup. For FXML documents, line 1 must appear as shown in Fig. 22.2. The XML declaration's `version` **attribute** specifies the XML syntax version (`1.0`) used in the document. The `encoding` **attribute** specifies the format of the document's character—XML documents typically contain Unicode characters in `UTF-8` format (`https://en.wikipedia.org/wiki/UTF-8`).

Attributes

Each XML attribute has the format

```
name="value"
```

The *name* and *value* are separated by `=` and the *value* placed in quotation marks (`""`). Multiple $name = value$ pairs are separated by whitespace.

Comments

Lines 2–3 are XML comments, which begin with `<!--` and end with `-->`, and can be placed almost anywhere in an XML document. XML comments can span to multiple lines.

FXML `import` Declarations

Lines 5–6 are **FXML `import` declarations** that specify the fully qualified names of the JavaFX types used in the document. Such declarations are delimited by `<?import` and `?>`.

Elements

XML documents contain **elements** that specify the document's structure. Most elements are delimited by a **start tag** and an **end tag**:

- A start tag consists of **angle brackets** (`<` and `>`) containing the element's name followed by zero or more attributes. For example, the `VBox` element's start tag (lines 8–10) contains four attributes.
- An end tag consists of the element name preceded by a **forward slash** (`/`) in angle brackets—for example, `</VBox>` in line 18.

An element's start and end tags enclose the element's contents. In this case, lines 11–17 declare other elements that describe the `VBox`'s contents. Every XML document must have exactly one **root element** that contains all the other elements. In Fig. 22.2, `VBox` is the root.

A layout element always contains a `children` element (lines 11–17) containing the child `Node`s that are arranged by that layout. For a `VBox`, the `children` element contains the child `Node`s in the order they're displayed on the screen from top to bottom. The elements in lines 12–16 represent the `VBox`'s five

`Label`s. These are **empty elements** that use the shorthand start-tag-only notation:

```
<ElementName attributes />
```

in which the empty element’s start tag ends with `/>` rather than `>`. The empty element:

```
<Label fx:id="label1" text="Arial 14pt bold" />
```

is equivalent to

```
<Label fx:id="label1" text="Arial 14pt bold">
</Label>
```

which does not have content between the start and end tags. Empty elements often have attributes (such as `fx:id` and `text` for each `Label` element).

XML Namespaces

In lines 9–10, the `VBox` attributes

```
xmlns="http://javafx.com/javafx/8.0.60"
xmlns:fx="http://javafx.com/fxml/1"
```

specify the XML namespaces used in FXML markup. An XML **namespace** specifies a collection of element and attribute names that you can use in the document. The attribute

```
xmlns="http://javafx.com/javafx/8.0.60"
```

specifies the default namespace. FXML `import` declarations (like those in lines 5–6) add names to this namespace for use in the document. The attribute

```
xmlns:fx="http://javafx.com/fxml/1"
```

specifies JavaFX's `fx` namespace. Elements and attributes from this namespace (such as the `fx:id` attribute) are used internally by the `FXMLLoader` class. For example, for each FXML element that specifies an `fx:id`, the `FXMLLoader` initializes a corresponding variable in the controller class. The `fx:` in `fx:id` is a **namespace prefix** that specifies the namespace (`fx`) that defines the attribute (`id`). Every element or attribute name in Fig. 22.2 that does not begin with `fx:` is part of the default namespace.

22.2.3 Referencing the CSS File from FXML

For the `Label`s to appear with the fonts shown in Fig.

22.2(b), we must reference the `FontCSS.css` file from the FXML. This enables Scene Builder to apply the CSS rules to the GUI. To reference the CSS file:

1. Select the `VBox` in the Scene Builder.
2. In the **Properties** inspector, click the + button under the **Stylesheets** heading.
3. In the dialog that appears, select the `FontCSS.css` file and click **Open**.

This adds the `stylesheets` attribute (line 8)

```
stylesheets="@FontCSS.css"
```

to the `VBox`'s opening tag (lines 8–10). The `@` symbol—called the local resolution operator in FXML—indicates that the file `FontCSS.css` is located relative to the FXML file on disk. No path information is specified here, so the CSS file and the FXML file must be in the same folder.

22.2.4 Specifying the VBox's Style Class

The preceding steps apply the font styles to the `Label`s, based on their ID selectors, but do not apply the spacing and padding to the `VBox`. Recall that for the `VBox` we defined a CSS rule using a *style class selector* with the name `.vbox`. To apply the CSS rule to the `VBox`:

1. Select the `VBox` in the Scene Builder.
2. In the **Properties** inspector, under the **Style Class** heading, specify the value `vbox` *without* the dot, then press *Enter* to complete the setting.

This adds the `styleClass` attribute

```
styleClass="vbox"
```

to the `VBox`’s opening tag (line 8). At this point the GUI appears as in Fig. 22.2(b). You can now run the app to see the output in Fig. 22.2(c).

22.2.5 Programmatically Loading CSS

In the `FontCSS` app, the FXML referenced the CSS style sheet directly (line 8). It’s also possible to load CSS files dynamically and add them to a `Scene`’s collection of style sheets. You might do this, for example, in an app that enables users to choose their preferred look-and-feel, such as a light background with dark text vs. a dark background with light text.

To load a stylesheet dynamically, add the following statement to the `Application` subclass’s `start` method:

```
scene.getStylesheets().add(
   getClass().getResource("FontCSS.css").toExternalFo
```

In the preceding statement:

- Inherited `Object` method `getClass` obtains a `Class` object representing the app's `Application` subclass.
- `Class` method `getResource` returns a `URL` representing the location of the file `FontCSS.css`. Method `getResource` looks for the file in the same location from which the `Application` subclass was loaded.
- `URL` method `toExternalForm` returns the `URL`'s `String` representation. This is passed to the `add` method of the `Scene`'s collection of style sheets—this adds the style sheet to the scene.

22.3 Displaying Two-Dimensional Shapes

JavaFX has two ways to draw shapes:

- You can define `Shape` and `Shape3D` (package `javafx.scene.shape`) subclass objects, add them to a container in the JavaFX stage and manipulate them like other JavaFX `Node`s.
- You can add a `Canvas` object (package `javafx.scene.canvas`) to a container in the JavaFX stage, then draw on it using various `GraphicsContext` methods.

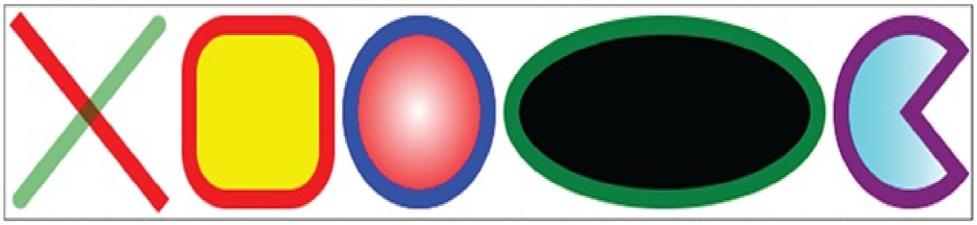
The `BasicShapes` example presented in this section shows you how to display two-dimensional `Shape`s of types `Line`, `Rectangle`, `Circle`, `Ellipse` and `Arc`. Like other `Node` types, you can drag shapes from the Scene Builder **Library**'s **Shapes** category onto the design area, then configure them via the **Inspector**'s **Properties**, **Layout** and **Code** sections—of course, you also may create objects of any JavaFX `Node` type programmatically.

22.3.1 Defining Two-Dimensional Shapes with FXML

Figure 22.3 shows the completed FXML for the `BasicShapes` app, which references the `BasicShapes.css` file (line 13) that we present in Section 22.3.2. For this app we dragged two `Line`s, a `Rectangle`, a `Circle`, an `Ellipse` and an `Arc` onto a `Pane` layout and configured their dimensions and positions in Scene Builder.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 22.3: BasicShapes.fxml -->
3  <!-- Defining Shape objects and styling via CSS -
4
5  <?import javafx.scene.layout.Pane?>
6  <?import javafx.scene.shape.Arc?>
7  <?import javafx.scene.shape.Circle?>
8  <?import javafx.scene.shape.Ellipse?>
9  <?import javafx.scene.shape.Line?>
10 <?import javafx.scene.shape.Rectangle?>
11
12 <Pane id="Pane" prefHeight="110.0" prefWidth="63
13    stylesheets="@BasicShapes.css" xmlns="http://
14    xmlns:fx="http://javafx.com/fxml/1">
15    <children>
16       <Line fx:id="line1" endX="100.0" endY="100
17          startX="10.0" startY="10.0" />
18       <Line fx:id="line2" endX="10.0" endY="100.
19          startX="100.0" startY="10.0" />
20       <Rectangle fx:id="rectangle" height="90.0"
21          layoutY="10.0" width="90.0" />
22       <Circle fx:id="circle" centerX="270.0" cen
23          radius="45.0" />
24       <Ellipse fx:id="ellipse" centerX="430.0" c
25          radiusX="100.0" radiusY="45.0" />
26       <Arc fx:id="arc" centerX="590.0" centerY="
27          radiusX="45.0" radiusY="45.0" startAngl
28    </children>
29 </Pane>
```

a) GUI in Scene Builder with CSS applied—Ellipse's image fill does not show.



b) GUI in running app—Ellipse's image fill displays correctly.

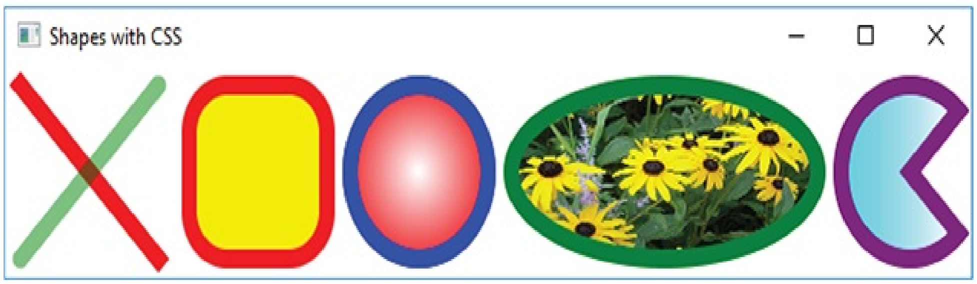


Fig. 22.3

Defining `Shape` objects and styling via CSS.

Description

For each property you can set in Scene Builder, there is a corresponding attribute in FXML. For example, the `Pane` object's **Pref Height** property in Scene Builder corresponds to the `prefHeight` attribute (line 12) in FXML. When you build this GUI in Scene Builder, use the FXML attribute values shown in Fig. 22.3. Note that as you drag each shape onto your design, Scene Builder automatically configures certain properties, such as the **Fill** and **Stroke** colors for the `Rectangle`, `Circle`, `Ellipse` and `Arc`. For each such property that does not have a corresponding attribute shown in Fig. 22.3, you can remove the attribute either by setting the property to its default value in Scene Builder or by manually editing the FXML.

Lines 6–10 import the shape classes used in the FXML. We also specified `fx:id` values (lines 16 and 18) for the two `Line`s—we use these values in CSS rules with ID selectors to define separate styles for each `Line`. We removed the shapes' `fill`, `stroke` and `strokeType` properties that Scene Builder autogenerated. The default `fill` for a shape is black. The default stroke is a one-pixel black line. The default `strokeType` is centered—based on the stroke's thickness, half the thickness appears inside the shape's bounds and half outside. You also may display a shape's stroke completely inside or outside the shape's bounds. We specify the strokes and fills with the styles in Section 22.3.2.

`Line` Objects

Lines 16–17 and 18–19 define two `Line`s. Each connects two endpoints specified by the properties `startX`, `startY`, `endX` and `endY`. The *x*- and *y*-coordinates are measured from the top-left corner of the `Pane`, with *x*-coordinates increasing left to right and *y*-coordinates increasing top to bottom. If you specify a `Line`'s `layoutX` and `layoutY` properties, then the `startX`, `startY`, `endX` and `endY` properties are measured from that point.

`Rectangle` Object

Lines 20–21 define a `Rectangle` object. A `Rectangle` is displayed based on its `layoutX`, `layoutY`, `width` and `height` properties:

- A `Rectangle`'s upper-left corner is positioned at the coordinates specified by the `layoutX` and `layoutY` properties, which are inherited from class `Node`.
- A `Rectangle`'s dimensions are specified by the `width` and `height` properties—in this case they have the same value, so the `Rectangle` defines a square.

`Circle` Object

Lines 22–23 define a `Circle` object with its center at the point specified by the `centerX` and `centerY` properties. The `radius` property determines the `Circle`'s size (two times the `radius`) around its center point.

`Ellipse` Object

Lines 24–25 define an `Ellipse` object. Like a `Circle`, an `Ellipse`'s center is specified by the `centerX` and `centerY` properties. You also specify `radiusX` and `radiusY` properties that help determine the `Ellipse`'s width (left and right of the center point) and height (above and below the center point).

`Arc` Object

Lines 26–27 define an `Arc` object. Like an `Ellipse`, an `Arc`’s center is specified by the `centerX` and `centerY` properties, and the `radiusX` and `radiusY` properties determine the `Arc`’s width and height. For an `Arc`, you also specify:

- `length`—The arc’s length in degrees (0–360). Positive values sweep counterclockwise.
- `startAngle`—The angle in degrees at which the arc should begin.
- `type`—How the arc should be closed. `ROUND` indicates that the starting and ending points of the arc should be connected to the center point by straight lines. You also may choose `OPEN`, which does not connect the start and end points, or `CHORD`, which connects the start and end points with a straight line.

22.3.2 CSS That Styles the Two-Dimensional Shapes

Figure 22.4 shows the CSS for the `BasicShapes` app. In this CSS file, we define two CSS rules with ID selectors (`#line1` and `#line2`) to style the app’s two `Line` objects. The remaining rules use **type selectors**, which apply to all objects of a given type. You specify a type selector by using the JavaFX class name.

```
/* Fig. 22.4: BasicShapes.css */
/* CSS that styles various two-dimensional shape
```

```
Line, Rectangle, Circle, Ellipse, Arc {
    -fx-stroke-width: 10;
}

#line1 {
    -fx-stroke: red;
}

#line2 {
    -fx-stroke: rgba(0%, 50%, 0%, 0.5);
    -fx-stroke-line-cap: round;
}

Rectangle {
    -fx-stroke: red;
    -fx-arc-width: 50;
    -fx-arc-height: 50;
    -fx-fill: yellow;
}

Circle {
    -fx-stroke: blue;
    -fx-fill: radial-gradient(center 50% 50%, rad
}

Ellipse {
    -fx-stroke: green;
    -fx-fill: image-pattern("yellowflowers.png");
}

Arc {
    -fx-stroke: purple;
    -fx-fill: linear-gradient(to right, cyan, whi
}
```

Fig. 22.4

CSS that styles various two-dimensional shapes.

Specifying Common Attributes for Various Objects

The CSS rule in lines 4–6 defines the `-fx-stroke-width` CSS property for all the shapes in the app—this property specifies the thickness of the `Line`s and the border thickness of all the other shapes. To apply this rule to multiple shapes we use CSS type selectors in a comma-separated list. So, line 4 indicates that the rule in lines 4–6 should be applied to all objects of types `Line`, `Rectangle`, `Circle`, `Ellipse` and `Arc` in the GUI.

Styling the `Line`s

The CSS rule in lines 8–10 sets the `-fx-stroke` to the solid color `red`. This rule applies to the `Line` with the `fx:id` `"line1"`. This rule is in addition to the rule at lines 4–6, which sets the stroke width for all `Line`s (and all the other shapes). When JavaFX renders an object, it combines all the CSS rules that apply to the object to determine its appearance. This rule applies to the `Line` with the `fx:id "line1"`.

Colors may be specified as

- named colors (such as `"red"`, `"green"` and `"blue"`),
- colors defined by their red, green, blue and alpha (transparency) components,
- colors defined by their hue, saturation, brightness and alpha components,

and more. For details on all the ways to specify color in CSS, see

```
https://docs.oracle.com/javase/8/javafx/api/javafx/sc
```

The CSS rule in lines 12–15 applies to the `Line` with the `fx:id "line2"`. For this rule, we specified the `-fx-stroke` property's color using the CSS function `rgba`, which defines a color based on its red, green, blue and alpha (transparency) components. Here we used the version of `rgba` that receives percentages from 0% to 100% specifying the amount of red, green and blue in the color, and a value from 0.0 (transparent) to 1.0 (opaque) for the alpha component. Line 13 produces a semitransparent green line. You can see the interaction between the two `Line`s' colors at the intersection point in Fig. 22.3's output windows. The `-fx-stroke-line-cap` CSS property (line 14) indicates that the ends of the `Line` should be *rounded*—the rounding effect becomes more noticeable with thicker strokes.

Styling the `Rectangle`

For `Rectangle`s, `Circle`s, `Ellipse`s and `Arc`s you can specify both the `-fx-stroke` for the shapes' borders and the `-fx-fill`, which specifies the color or pattern that appears inside the shape. The rule in lines 17–22 uses a CSS type selector to indicate that all `Rectangle`s should have `red` borders (line 18) and `yellow` fill (line 21). Lines 19–20 define the `Rectangle`'s `-fx-arc-width` and `-fx-arc-height` properties, which specify the width and height of an ellipse that's divided in half horizontally and vertically, then used to round the `Rectangle`'s corners. Because these properties have the same value (`50`) in this app, the four corners are each one quarter of a circle with a diameter of 50.

Styling the `Circle`

The CSS rule at lines 24–27 applies to all `Circle` objects. Line 25 sets the `Circle`'s stroke to `blue`. Line 26 sets the `Circle`'s fill with a **gradient**—colors that transition gradually from one color to the next. You can transition between as many colors as you like and specify the points at which to change colors, called **color stops**. You can use gradients for any property that specifies a color. In this case, we use the CSS function `radial-gradient` in which the color changes gradually from a center point outward. The fill

```
-fx-fill: radial-gradient(center 50% 50%, radius 60%,
```

indicates that the gradient should begin from a `center` point at `50% 50%`—the middle of the shape horizontally and the middle of the shape vertically. The `radius` specifies the distance from the center at which an even mixture of the two colors appears. This radial gradient begins with the color `white` in the center and ends with `red` at the outer edge of the `Circle`'s fill. We'll discuss a linear gradient momentarily.

Styling the `Ellipse`

The CSS rule at lines 29–32 applies to all `Ellipse` objects. Line 30 specifies that an `Ellipse` should have a `green` stroke. Line 31 specifies that the `Ellipse`'s fill should be the image in the file `yellowflowers.png`, which is located in this app's folder. This image is provided in the `images` folder with the chapter's examples—if you're building this app from scratch, copy the video into the app's folder on your system. To specify an image as fill, you use the CSS function `image-pattern`. [*Note:* At the time of this writing, Scene Builder does not display a shape's fill correctly if it's specified with a CSS `image-pattern`. You must run the example to see the fill, as shown in Fig. 22.3(b).]

Styling the `Arc`

The CSS rule at lines 34–37 applies to all `Arc` objects. Line 35 specifies that an `Arc` should have a `purple` stroke. In this case, line 36

```
-fx-fill: linear-gradient(to right, cyan, white);
```

specifies that the `Arc` should be filled with a **linear gradient**—such gradients gradually transition from one color to the next horizontally, vertically or diagonally. You can transition between as many colors as you like and specify the points at which to change colors. To create a linear gradient, you use the CSS function `linear-gradient`. In this case, `to right` indicates that the gradient should start from the shape's left edge and transition through colors to the shape's right edge. We specified only two colors here—`cyan` at the left edge of the gradient and `white` at the right edge—but two or more colors can be specified in the comma-separated list. For more information on all the options for configuring radial gradients, linear gradients and image patterns, see

```
https://docs.oracle.com/javase/8/javafx/api/javafx/sc
```

22.4 `Polylines`, `Polygons` and `Paths`

There are several kinds of JavaFX shapes that enable you to create custom shapes:

- `Polyline`—draws a series of connected lines defined by a set of points.
- `Polygon`—draws a series of connected lines defined by a set of points and connects the last point to the first point.
- `Path`—draws a series of connected `PathElements` by moving to a given point, then drawing lines, arcs and curves.

In the `PolyShapes` app, you select which shape you want to display by selecting one of the `RadioButton`s in the left column. You specify a shape's points by clicking throughout the `AnchoredPane` in which the shapes are displayed.

For this example, we do not show the `PolyShapes` subclass of `Application` (located in the example's `PolyShapes.java` file), because it loads the FXML and displays the GUI, as demonstrated in Chapters 12 and 13.

22.4.1 GUI and CSS

This app's GUI (Fig. 22.5) is similar to that of the `Painter` app in Section 13.3. For that reason, we show only the key

GUI elements' `fx:id` property values, rather than the complete FXML—each `fx:id` property value ends with the GUI element's type. In this GUI:

- The three `RadioButton`s are part of a `ToggleGroup` with the `fx:id` `"toggleGroup"`. The **Polyline** `RadioButton` should be **Selected** by default. We also set each `RadioButton`'s **On Action** event handler to `shapeRadioButtonSelected`.
- We dragged a `Polyline`, a `Polygon` and a `Path` from the Scene Builder **Library**'s **Shapes** section onto the `Pane` that displays the shapes, and we set their `fx:id`s to `polyline`, `polygon` and `path`, respectively. We set each shape's `visible` property to `false` by selecting the shape in Scene Builder, then unchecking the **Visible** checkbox in the **Properties** inspector. We display only the shape with the selected `RadioButton` at runtime.
- We set the `Pane`'s **On Mouse Clicked** event handler to `drawingAreaMouseClicked`.
- We set the **Clear** `Button`'s **On Action** event handler to `clearButtonPressed`.
- We set the controller class to `PolyShapesController`.
- Finally, we edited the FXML to remove the `Path` object's `<elements>` and the `Polyline` and `Polygon` objects' `<points>`, as we'll set these programmatically in response to the user's mouse-click events.

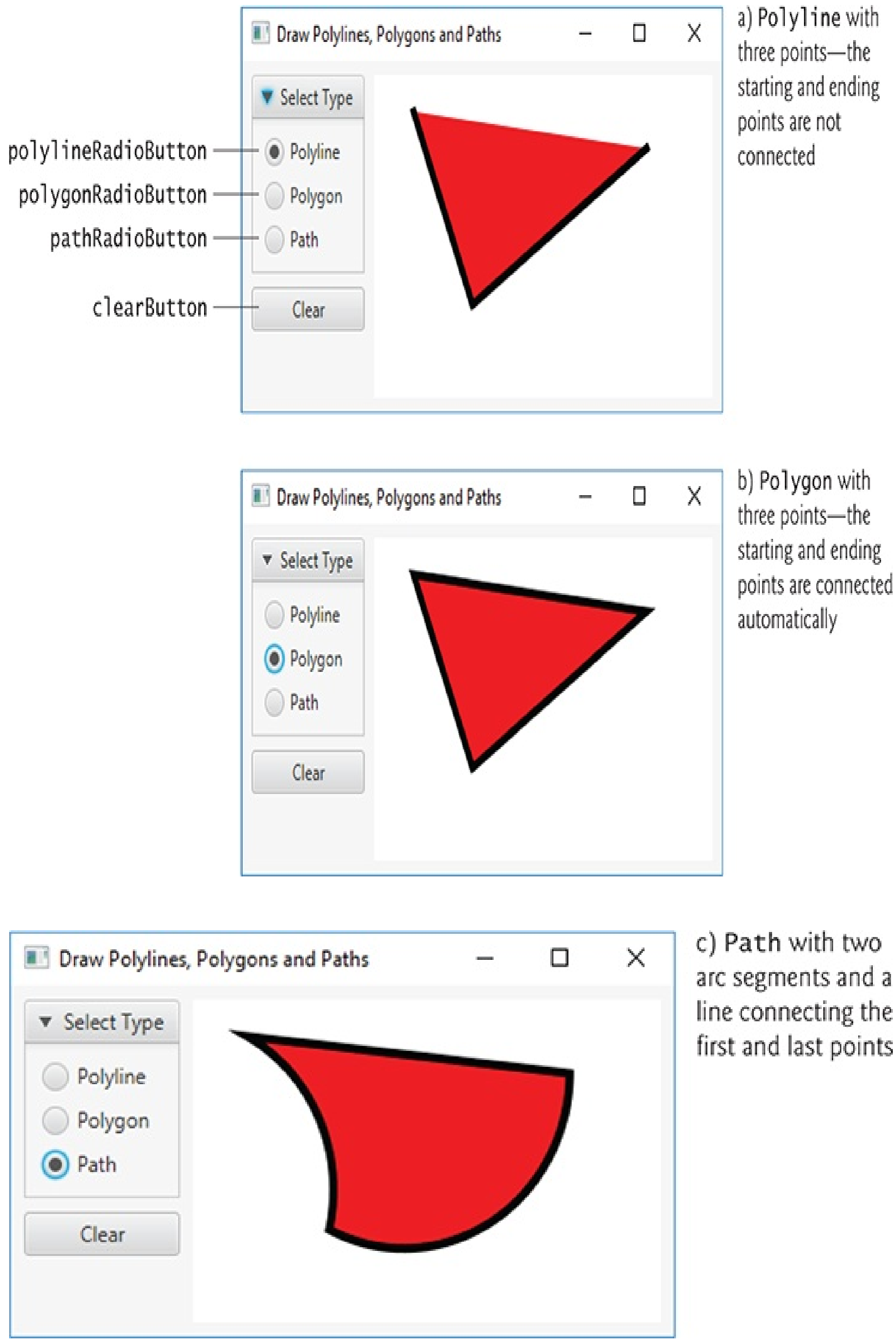

Draw Polylines, Polygons and Paths
Select Type
Polyline
Polygon
Path
Clear
polylineRadioButton
polygonRadioButton
pathRadioButton
clearButton
a) Polyline with three points—the starting and ending points are not connected
b) Polygon with three points—the starting and ending points are connected automatically
c) Path with two arc segments and a line connecting the first and last points

Fig. 22.5

`Polyline`s, `Polygon`s and `Path`s.

Description

The `PolyShapes.css` file defines the properties `-fx-stroke`, `-fx-stroke-width` and `-fx-fill` that are applied to all three shapes in this example. As you can see in Fig. 22.5, the stroke is a thick black line (5 pixels wide) and the fill is red.

```
Polyline, Polygon, Path {
    -fx-stroke: black;
    -fx-stroke-width: 5;
    -fx-fill: red;
}
```

22.4.2 `PolyShapesController` Class

Figure 22.6 shows this app's `PolyShapesController` class, which responds to the user's interactions. The `enum ShapeType` (line 17) defines three constants that we use to determine which shape to display. Lines 20–26 declare the variables that correspond to the GUI components and shapes with `fx:id`s in the FXML. The `shapeType` variable (line 29) stores whichever shape type is currently selected in the GUI's `RadioButton`s—by default, the `Polyline` will be displayed. As you’ll soon see, the `sweepFlag` variable is used to determine whether an arc in a `Path` is drawn with a negative or positive sweep angle.

```
// Fig. 22.6: PolyShapesController.java
// Drawing Polylines, Polygons and Paths.
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.input.MouseEvent;
import javafx.scene.shape.ArcTo;
import javafx.scene.shape.ClosePath;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Polyline;

public class PolyShapesController {
   // enum representing shape types
   private enum ShapeType {POLYLINE, POLYGON, PA

   // instance variables that refer to GUI compo
   @FXML private RadioButton polylineRadioButton
   @FXML private RadioButton polygonRadioButton;
   @FXML private RadioButton pathRadioButton;
   @FXML private ToggleGroup toggleGroup;
   @FXML private Polyline polyline;
   @FXML private Polygon polygon;
   @FXML private Path path;

   // instance variables for managing state
   private ShapeType shapeType = ShapeType.POLYL
   private boolean sweepFlag = true; // used wit

   // set user data for the RadioButtons and dis
   public void initialize() {
      // user data on a control can be any Objec
```

```
      polylineRadioButton.setUserData(ShapeType.
      polygonRadioButton.setUserData(ShapeType.P
      pathRadioButton.setUserData(ShapeType.PATH

      displayShape(); // sets polyline's visibil
   }

   // handles drawingArea's onMouseClicked event
   @FXML
   private void drawingAreaMouseClicked(MouseEve
      polyline.getPoints().addAll(e.getX(), e.ge
      polygon.getPoints().addAll(e.getX(), e.get

      // if path is empty, move to first click p
      if (path.getElements().isEmpty()) {
         path.getElements().add(new MoveTo(e.get
         path.getElements().add(new ClosePath())
      }
      else { // insert a new path segment before
         // create an arc segment and insert it
         ArcTo arcTo = new ArcTo();
         arcTo.setX(e.getX());
         arcTo.setY(e.getY());
         arcTo.setRadiusX(100.0);
         arcTo.setRadiusY(100.0);
         arcTo.setSweepFlag(sweepFlag);
         sweepFlag = !sweepFlag;
         path.getElements().add(path.getElements
      }
   }

   // handles color RadioButton's ActionEvents
   @FXML
   private void shapeRadioButtonSelected(ActionE
      // user data for each color RadioButton is
      shapeType =
         (ShapeType) toggleGroup.getSelectedTogg
      displayShape(); // display the currently s
   }

```

```
75     // displays currently selected shape
76     private void displayShape() {
77        polyline.setVisible(shapeType == ShapeType
78        polygon.setVisible(shapeType == ShapeType.
79        path.setVisible(shapeType == ShapeType.PAT
80     }
81
82     // resets each shape
83     @FXML
84     private void clearButtonPressed(ActionEvent e
85        polyline.getPoints().clear();
86        polygon.getPoints().clear();
87        path.getElements().clear();
88     }
89  }
```

Fig. 22.6

Drawing `Polyline`s, `Polygon`s and `Path`s.

Method `initialize`

Recall from Section 13.3.1 that you can associate any `Object` with each JavaFX control via its `setUserData` method. For the shape `RadioButton`s in this app, we store the specific `ShapeType` that the `RadioButton` represents (lines 35–37). We use these values when handling the `RadioButton` events to set the `shapeType` instance variable. Line 39 then calls method `displayShape` to display the currently selected shape (the `Polyline` by default). Initially, the shape is not visible because it does not yet have any points.

Method `drawingAreaMouseClicked`

When the user clicks the app's `Pane`, method `drawingAreaMouseClicked` (lines 43–64) modifies all three shapes to incorporate the new point at which the user clicked. `Polyline`s and `Polygon`s store their points as a collection of `Double` values in which the first two values represent the first point's location, the next two values represent the second point's location, etc. Line 45 gets the `polyline` object's collection of points, then adds the new click point to the collection by calling its `addAll` method and passing the `MouseEvent`'s *x*- and *y*-coordinate values. This adds the new point's information to the end of the collection. Line 46 performs the same task for the `polygon` object.

Lines 49–63 manipulate the `path` object. A `Path` is represented by a collection of `PathElements`. The subclasses of `PathElement` used in this example are:

- `MoveTo`—Moves to a specific position without drawing anything.
- `ArcTo`—Draws an arc from the previous `PathElement`'s endpoint to the specified location. We'll discuss this in more detail momentarily.
- `ClosePath`—Closes the path by drawing a straight line from the end

point of the last `PathElement` to the start point of the first `PathElement`.

Other `PathElement`s not covered here include `LineTo`, `HLineTo`, `VLineTo`, `CubicCurveTo` and `QuadCurveTo`.

When the user clicks the `Pane`, line 49 checks whether the `Path` contains elements. If not, line 50 moves the starting point of the `path` to the mouse-click location by adding a `MoveTo` element to the path's `PathElement`s collection. Then line 51 adds a new `ClosePath` element to complete the path. For each subsequent mouse-click event, lines 55–60 create an `ArcTo` element and line 62 inserts it before the `ClosePath` element by calling the `PathElement`s collection's `add` method that receives an index as its first argument.

Lines 56–57 set the `ArcTo` element's end point to the `MouseEvent`'s coordinates. The arc is drawn as a piece of an ellipse for which you specify the horizontal radius and vertical radius (lines 58–59). Line 60 sets the `ArcTo`'s `sweepFlag`, which determines whether the arc sweeps in the positive angle direction (`true`; counter clockwise) or the negative angle direction (`false`; clockwise). By default an `ArcTo` element is drawn as the shortest arc between the last `PathElement`'s end point and the point specified by the `ArcTo` element. To sweep the arc the long way around the ellipse, set the `ArcTo`'s `largeArcFlag` to `true`. For each mouse click, line 61 reverses the value of our controller class's `sweepFlag` instance variable so that the `ArcTo` elements toggle between positive and negative angles for variety.

Method `shapeRadioButtonSelected`

When the user clicks a shape `RadioButton`, lines 70–71 set the controller's `shapeType` instance variable, then line 72 calls method `displayShape` to display the selected shape. Try creating a `Polyline` of several points, then changing to the `Polygon` and `Path` to see how the points are used in each shape.

Method `displayShape`

Lines 77–79 simply set the visibility of the three shapes, based on the current `shapeType`. The currently selected shape's visibility is set to `true` to display the shape, and the other shapes' visibility is set to `false` to hide those shapes.

Method `clearButtonPressed`

When the user clicks the **Clear** `Button`, lines 85–86 clear the `polyline`’s and `polygon`’s collections of points, and line 87 clears the `path`’s collection of `PathElement`s. The user can then begin drawing a new shape by clicking the `Pane`.

22.5 Transforms

A **transform** can be applied to any UI element to *reposition* or *reorient* the element. The built-in JavaFX transforms are subclasses of `Transform`. Some of these subclasses include:

- `Translate`—*moves* an object to a new location.
- `Rotate`—*rotates* an object around a point and by a specified rotation angle.
- `Scale`—*scales* an object's size by the specified amounts.

The next example draws stars using the `Polygon` control and uses `Rotate` transforms to create a circle of randomly colored stars. The FXML for this app consists of an empty 300-by-300 `Pane` layout with the `fx:id "pane"`. We also set the controller class to `DrawStarsController`. Figure 22.7 shows the app's controller and a sample output.

Method initialize (lines 14–37) defines the stars, applies the transforms and attaches the stars to the app's `pane`. Lines 16–18 define the points of a star as an array of type `Double`—the collection of points stored in a `Polygon` is implemented with a generic collection, so you must use type `Double` rather than `double` (recall that primitive types cannot be used in Java generics). Each pair of values in the array represents the *x*- and *y*-coordinates of one point in the `Polygon`. We defined ten points in the array.

```
// Fig. 22.7: DrawStarsController.java
// Create a circle of stars using Polygons and R
import java.security.SecureRandom;
import javafx.fxml.FXML;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;
import javafx.scene.transform.Transform;

public class DrawStarsController {
    @FXML private Pane pane;
    private static final SecureRandom random = ne

    public void initialize() {
        // points that define a five-pointed star
        Double[] points = {205.0,150.0, 217.0,186.
            223.0,204.0, 233.0,246.0, 205.0,222.0,
            223.0,204.0, 233.0,246.0, 205.0,222.0,

        // create 18 stars
        for (int count = 0; count < 18; ++count) {
            // create a new Polygon and copy existi
            Polygon newStar = new Polygon();
            newStar.getPoints().addAll(points);

            // create random Color and set as newSt
            newStar.setStroke(Color.GREY);
            newStar.setFill(Color.rgb(random.nextIn
                random.nextInt(255), random.nextInt(
                random.nextDouble()));

            // apply a rotation to the shape
            newStar.getTransforms().add(
                Transform.rotate(count * 20, 150, 15
            pane.getChildren().add(newStar);
        }
    }
}
```

Fig. 22.7

Create a circle of stars using `Polygon`s and `Rotate` transforms.

Description

During each iteration of the loop, lines 23–34 create a `Polygon` using the points in the `points` array and apply a different `Rotate` transform. This results in the circle of `Polygon`s in the screen capture. To generate the random colors for each star, we use a `Secure-Random` object to create three random values from 0–255 for the red, green and blue components of the color, and one random value from 0.0–1.0 for the color’s alpha transparency value. We pass those values to class `Color`’s `static rgb` method to create a `Color`.

To apply a rotation to the new `Polygon`, we add a `Rotate` transform to the `Polygon`'s collection of `Transform`s (lines 33–34). To create the `Rotate` transform object, we invoke class `Transform`'s `static` method `rotate` (line 34), which returns a `Rotate` object. The method's first argument is the rotation angle. Each iteration of the loop assigns a new rotation-angle value by using the control variable multiplied by 20 as the `rotate` method's first argument. The method's next two arguments are the *x*- and *y*-coordinates of the point of rotation around which the `Polygon` rotates. The center of the circle of stars is the point *(150, 150)*, because we rotated all 18 stars around that point. Adding each `Polygon` as a new child element of the `pane` object allows the `Polygon` to be rendered on screen.

22.6 Playing Video with `Media`, `MediaPlayer` and `MediaViewer`

Many of today's most popular apps are multimedia intensive. JavaFX provides audio and video multimedia capabilities via the classes of package `javafx.scene.media`:

- For simple audio playback you can use class `AudioClip`.
- For audio playback with more playback controls and for video playback you can use classes `Media`, `MediaPlayer` and `MediaView`.

In this section, you'll build a basic video player. We'll explain classes `Media`, `MediaPlayer` and `MediaView` as we encounter them in the project's controller class (Section 22.6.2). The video used in this example is from NASA's multimedia library[3] and was downloaded from

3. For NASA's terms of use, visit `http://www.nasa.gov/multimedia/guidelines/`.

```
http://www.nasa.gov/centers/kennedy/multimedia/HD-ind
```

The video file `sts117.mp4` is provided in the `video` folder with this chapter's examples. When building the app from scratch, copy the video onto the app's folder.

Media Formats

For video, JavaFX supports MPEG-4 (also called MP4) and Flash Video formats. We downloaded a Windows WMV version of the video file used in this example, then converted it to MP4 via a free online video converter.[4]

4. There are many free online and downloadable video-format conversion tools. We used the one at `https://convertio.co/video-converter/`.

ControlsFX Library's `ExceptionDialog`

`ExceptionDialog` is one of many additional JavaFX controls available through the open-source project ControlsFX at

```
http://controlsfx.org
```

We use an `ExceptionDialog` in this app to display a message to the user if an error occurs during media playback.

You can download the latest version of ControlsFX from the preceding web page, then extract the contents of the ZIP file. Place the extracted ControlsFX JAR file (named `controlsfx-8.40.12.jar` at the time of this writing) in your project's folder—a JAR file is a compressed archive like a ZIP file, but contains Java class files and their corresponding resources. We included a copy of the JAR file with the final example.

Compiling and Running the App with ControlsFX

To compile this app, you must specify the JAR file as part of the app's classpath. To do so, use the `javac` command's `-classpath` option, as in:

```
javac -classpath .;controlsfx-8.40.12.jar *.java
```

Similary, to run the app, use the `java` command's `-cp` option, as in

```
java -cp .;controlsfx-8.40.12.jar VideoPlayer
```

In the preceding commands, Linux and macOS users should use a colon (`:`) rather than a semicolon(`;`). The classpath in each command specifies the current folder containing the app's files—this is represented by the dot (`.`)—and the name of the JAR file containing the ControlsFX classes (including `ExceptionDialog`).

22.6.1 VideoPlayer GUI

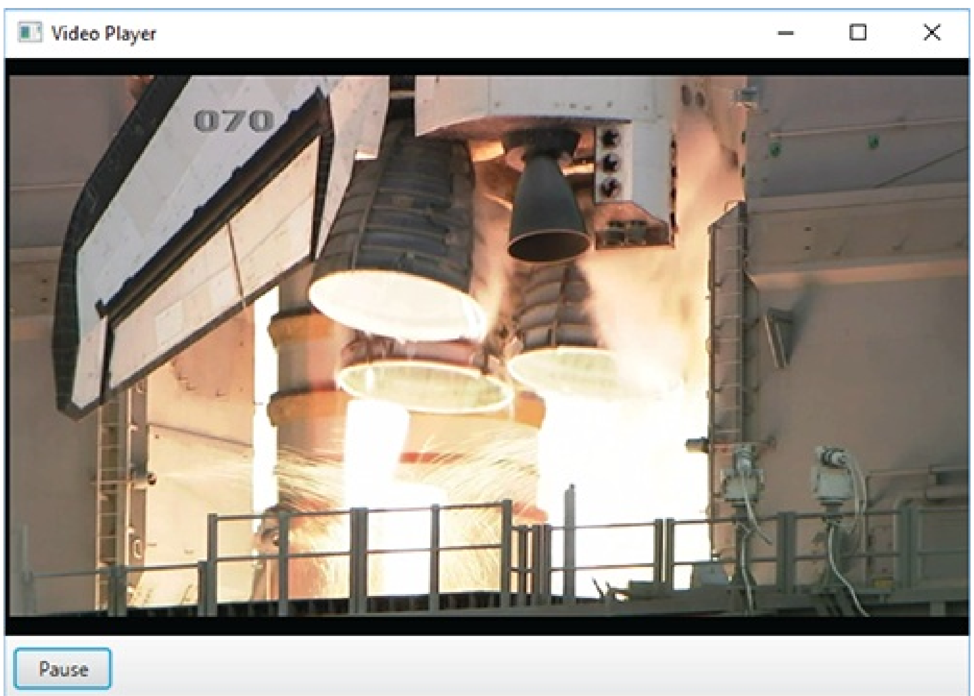
Figure 22.8 shows the completed `VideoPlayer.fxml` file and two sample screen captures of the final running `VideoPlayer` app. The GUI’s layout is a `BorderPane` consisting of

- a `MediaView` (located in the Scene Builder **Library**’s **Controls** section) with the `fx:id mediaView` and
- a `ToolBar` (located in the Scene Builder **Library**’s **Containers** section) containing one `Button` with the `fx:id playPauseButton` and the text `"Play"`. The controller method `playPauseButtonPressed` responds when the `Button` is pressed.

We placed the `MediaView` in the `BorderPane`’s center region (lines 25–27) so that it occupies all available space in the `BorderPane`, and we placed the `ToolBar` in the `BorderPane`’s bottom region (lines 15–24). By default, Scene Builder adds one `Button` to the `ToolBar` when you drag the `ToolBar` onto your layout. You can then add other controls to the `ToolBar` as necessary. We set the controller class to `VideoPlayerController`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 22.8: VideoPlayer.fxml -->
3  <!-- VideoPlayer GUI with a MediaView and a Butt
4
5  <?import javafx.scene.control.Button?>
6  <?import javafx.scene.control.ToolBar?>
7  <?import javafx.scene.layout.BorderPane?>
8  <?import javafx.scene.media.MediaView?>
9
10 <BorderPane prefHeight="400.0" prefWidth="600.0"
11       style="-fx-background-color: black;"
12       xmlns="http://javafx.com/javafx/8.0.60"
13       xmlns:fx="http://javafx.com/fxml/1"
```

```
      fx:controller="VideoPlayerController">
   <bottom>
      <ToolBar prefHeight="40.0" prefWidth="200.
         BorderPane.alignment="CENTER">
         <items>
            <Button fx:id="playPauseButton"
               onAction="#playPauseButtonPressed"
               prefWidth="60.0" text="Play" />
         </items>
      </ToolBar>
   </bottom>
   <center>
      <MediaView fx:id="mediaView" BorderPane.al
   </center>
</BorderPane>
```



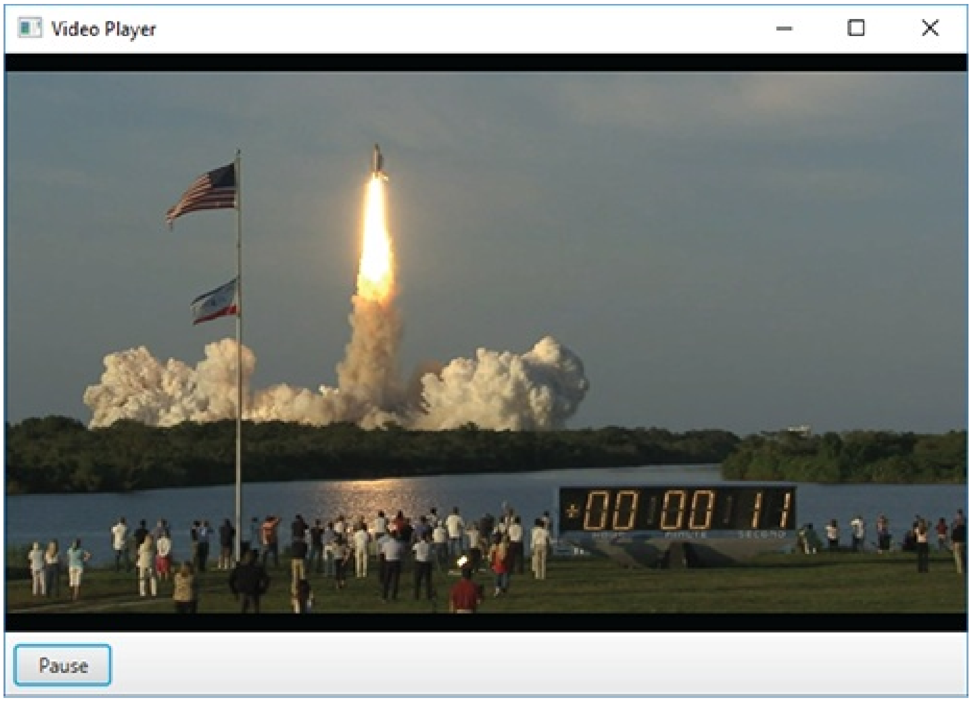


Fig. 22.8

`VideoPlayer` GUI with a `MediaView` and a `Button`. Video courtesy of NASA—see `http://www.nasa.gov/multimedia/guidelines/` for usage guidelines.

Description

22.6.2 `VideoPlayerController` Class

Figure 22.9 shows the completed `VideoPlayerController` class, which configures video playback and responds to state changes from the `MediaPlayer` and the events when the user presses the `playPauseButton`. The controller uses classes `Media`, `MediaPlayer` and `MediaView` as follows:

- A `Media` object specifies the location of the media to play and provides access to various information about the media, such as its duration, dimensions and more.
- A `MediaPlayer` object loads a `Media` object and controls playback. In addition, a `MediaPlayer` transitions through its various states (*ready*, *playing*, *paused*, etc.) during media loading and playback. As you'll see, you can provide `Runnable`s that execute in response to these state transitions.
- A `MediaView` object displays the `Media` being played by a given `MediaPlayer` object.

```
// Fig. 22.9: VideoPlayerController.java
// Using Media, MediaPlayer and MediaView to pla
import java.net.URL;
import javafx.beans.binding.Bindings;
import javafx.beans.property.DoubleProperty;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.util.Duration;
import org.controlsfx.dialog.ExceptionDialog;

public class VideoPlayerController {
   @FXML private MediaView mediaView;
   @FXML private Button playPauseButton;
   private MediaPlayer mediaPlayer;
```

```
   private boolean playing = false;

   public void initialize() {
      // get URL of the video file
      URL url = VideoPlayerController.class.getR

      // create a Media object for the specified
      Media media = new Media(url.toExternalForm

      // create a MediaPlayer to control Media p
      mediaPlayer = new MediaPlayer(media);

      // specify which MediaPlayer to display in
      mediaView.setMediaPlayer(mediaPlayer);

      // set handler to be called when the video
      mediaPlayer.setOnEndOfMedia(
         new Runnable() {
            public void run() {
               playing = false;
               playPauseButton.setText("Play");
               mediaPlayer.seek(Duration.ZERO);
               mediaPlayer.pause();
            }
         }
      );

      // set handler that displays an ExceptionD
      mediaPlayer.setOnError(
         new Runnable() {
            public void run() {
               ExceptionDialog dialog =
                  new ExceptionDialog(mediaPlaye
               dialog.showAndWait();
            }
         }
      );

      // set handler that resizes window to vide
      mediaPlayer.setOnReady(
```

```
            new Runnable() {
               public void run() {
                  DoubleProperty width = mediaView.
                  DoubleProperty height = mediaView
                  width.bind(Bindings.selectDouble(
                     mediaView.sceneProperty(), "wi
                  height.bind(Bindings.selectDouble
                     mediaView.sceneProperty(), "he
               }
            }
         );
      }

      // toggle media playback and the text on the
      @FXML
      private void playPauseButtonPressed(ActionEve
         playing = !playing;

         if (playing) {
            playPauseButton.setText("Pause");
            mediaPlayer.play();
         }
         else {
            playPauseButton.setText("Play");
            mediaPlayer.pause()
         }
      }
   }
```

Fig. 22.9

Using `Media`, `MediaPlayer` and `MediaView` to play a video.

Instance Variables

Lines 16–19 declare the controller's instance variables. When the app loads, the `mediaView` variable (line 16) is assigned a reference to the `MediaView` object declared in the app's FXML. The `mediaPlayer` variable (line 18) is configured in method `initialize` to load the video specified by a `Media` object and used by method `playPauseButtonPressed` (lines 73–85) to play and pause the video.

Creating a `Media` Object Representing the Video to Play

Method `initialize` configures media playback and registers event handlers for `MediaPlayer` events. Line 23 gets a `URL` representing the location of the `sts117.mp4` video file. The notation

```
VideoPlayerController.class
```

creates a `Class` object representing the `VideoPlayerController` class. This is equivalent to calling inherited method `getClass()`. Next line 26 creates a `Media` object representing the video. The argument to the

`Media` constructor is a `String` representing the video's location, which we obtain with `URL` method `toExternalForm`. The URL `String` can represent a local file on your computer or can be a location on the web. The `Media` constructor throws various exceptions, including `MediaException`s if the media cannot be found or is not of a supported media format.

Creating a `MediaPlayer` Object to Load the Video and Control Playback

To load the video and prepare it for playback, you must associate it with a `MediaPlayer` object (line 29). Playing multiple videos requires a separate `MediaPlayer` for each `Media` object. However, a given `Media` object can be associated with multiple `MediaPlayer`s. The `MediaPlayer` constructor throws a `NullPointerException` if the `Media` is `null` or a `MediaException` if a problem occurs during construction of the `MediaPlayer` object.

Attaching the `MediaPlayer` Object to the

MediaView to Display the Video

A MediaPlayer does not provide a view in which to display video. For this purpose, you must associate a MediaPlayer with a MediaView. When the MediaView already exists—such as when it's created in FXML—you call the MediaView's setMediaPlayer method (line 32) to perform this task. When creating a MediaView object programmatically, you can pass the MediaPlayer to the MediaView's constructor. A MediaView is like any other Node in the scene graph, so you can apply CSS styles, transforms and animations (Sections 22.7–22.9) to it as well.

Configuring Event Handlers for MediaPlayer Events

A MediaPlayer transitions through various states. Some common states include *ready*, *playing* and *paused*. For these and other states, you can execute a task as the MediaPlayer enters the corresponding state. In addition, you can specify tasks that execute when the end of media playback is reached or when an error occurs during playback. To perform a task for a given state, you specify an object that implements the Runnable interface (package java.lang). This interface contains a no-parameter run method that returns void.

For example, lines 35–44 call the `MediaPlayer`'s `setOnEndOfMedia` method, passing an object of an anonymous inner class that implements interface `Runnable` to execute when video playback completes. Line 38 sets the `boolean` instance variable `playing` to `false` and line 39 changes the text on the `playPauseButton` to `"Play"` to indicate that the user can click the `Button` to play the video again. Line 40 calls `MediaPlayer` method `seek` to move to the beginning of the video and line 41 pauses the video.

Lines 47–55 call the `MediaPlayer`'s `setOnError` method to specify a task to perform if the `MediaPlayer` enters the *error* state, indicating that an error occurred during playback. In this case, we display an `ExceptionDialog` containing the `MediaPlayer`'s error message. Calling the `ExceptionDialog`'s `showAndWait` method indicates that the app must wait for the user to dismiss the dialog before continuing.

Binding the `MediaViewer`'s Size to the Scene's Size

Lines 58–69 call the `MediaPlayer`'s `setOnReady` method to specify a task to perform if the `MediaPlayer` enters the *ready* state. We use property bindings to bind the `MediaView`'s `width` and `height` properties to the scene's `width` and `height` properties so that the `MediaView` resizes with app's window. A `Node`'s `sceneProperty` returns a `ReadOnlyObjectProperty<Scene>` that you can use to access to the `Scene` in which the `Node` is displayed. The `ReadOnlyObjectProperty<Scene>` represents an object that has many properties. To bind to a specific properties of that object, you can use the methods of class `Bindings` (package `javafx.beans.binding`) to select the corresponding properties. The `Scene`'s `width` and `height` are each `DoubleProperty` objects. `Bindings` method `selectDouble` gets a reference to a `DoubleProperty`. The method's first argument is the object that contains the property and the second argument is the name of the property to which you'd like to bind.

Method `playPauseButtonPressed`

The event handler `playPauseButtonPressed` (lines 73–85) toggles video playback. When `playing` is `true`, line 78 sets the `playPauseButton`'s text to `"Pause"` and line 79 calls the `MediaPlayer`'s `play` method; otherwise, line 82 sets the `playPauseButton`'s text to `"Play"` and line 83 calls the `MediaPlayer`'s `pause` method.

Using Java SE 8 Lambdas

to Implement the `Runnable`s

8

Each of the anonymous inner classes in this controller’s `initialize` method can be implemented more concisely using lambdas as shown in Section 17.16.

22.7 Transition Animations

Animations in JavaFX apps transition a Node's property values from one value to another in a specified amount of time. Most properties of a Node can be animated. This section focuses on several of JavaFX's predefined Transition animations from the javafx.animations package. By default, the subclasses that define Transition animations change the values of specific Node properties. For example, a FadeTransition changes the value of a Node's opacity property (which specifies whether the Node is opaque or transparent) over time, whereas a PathTransition changes a Node's location by moving it along a Path over time. Though we show sample screen captures for all the animation examples, the best way to experience each is to run the examples yourself.

22.7.1 TransitionAnimations.fxml

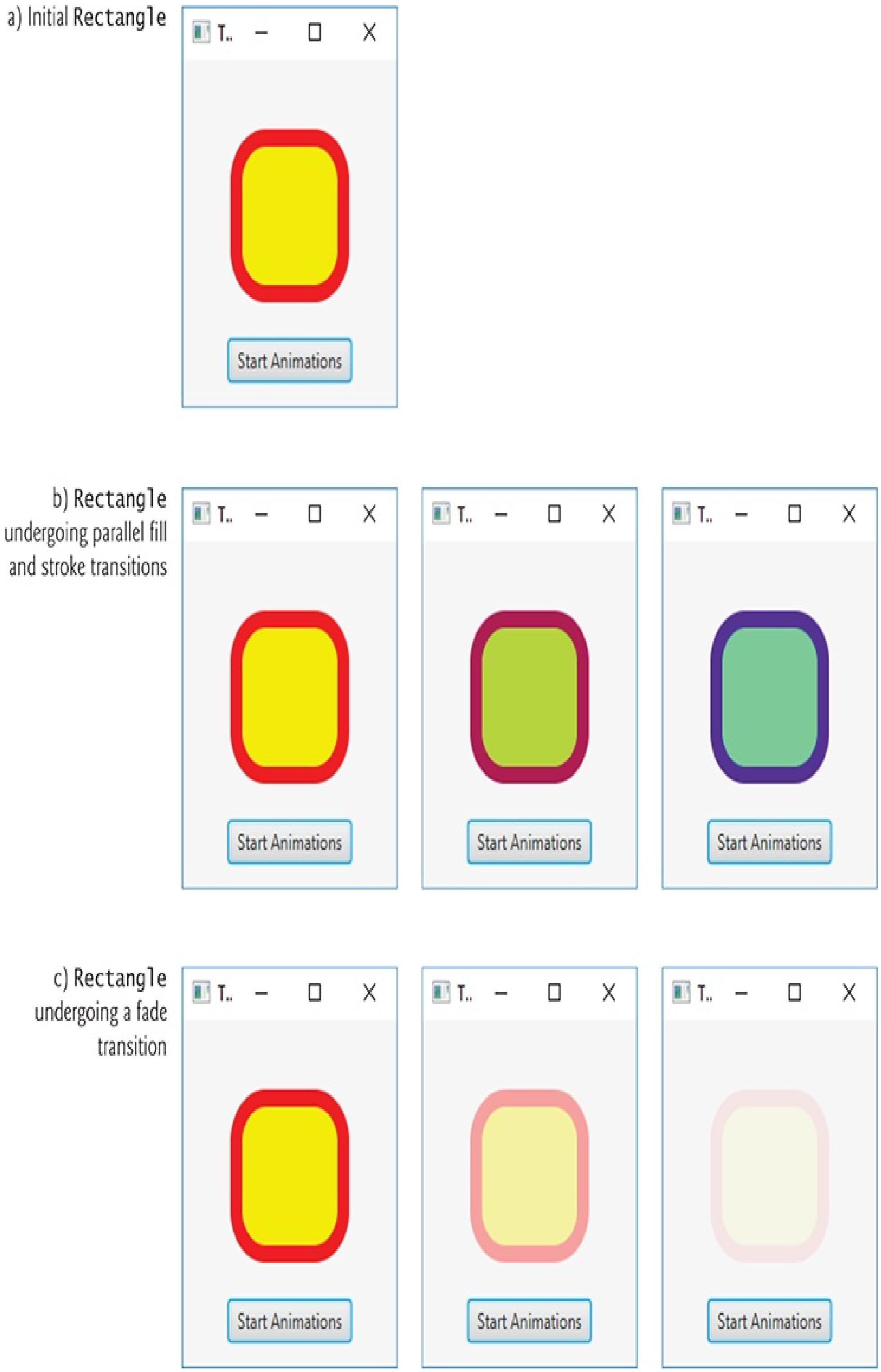
Figure 22.10 shows this app's GUI and screen captures of the running application. When you click the startButton (lines 17–19), its `startButtonPressed` event handler in the app's controller creates a sequence of `Transition` animations for the `Rectangle` (lines 15–16) and plays them. The `Rectangle` is styled with the following CSS from the file `TransitionAnimations.css`:

```
Rectangle {
   -fx-stroke-width: 10;
   -fx-stroke: red;
   -fx-arc-width: 50;
   -fx-arc-height: 50;
   -fx-fill: yellow;
}
```

which produces a rounded rectangle with a 10-pixel red border and yellow fill.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <!-- Fig. 22.10: TransitionAnimations.fxml -->
3   <!-- FXML for a Rectangle and Button -->
4
5   <?import javafx.scene.control.Button?>
6   <?import javafx.scene.layout.Pane?>
7   <?import javafx.scene.shape.Rectangle?>
8
9   <Pane id="Pane" prefHeight="200.0" prefWidth="180
10        stylesheets="@TransitionAnimations.css"
11        xmlns="http://javafx.com/javafx/8.0.60"
12        xmlns:fx="http://javafx.com/fxml/1"
13        fx:controller="TransitionAnimationsController
14     <children>
15        <Rectangle fx:id="rectangle" height="90.0"
16           layoutY="45.0" width="90.0" />
17        <Button fx:id="startButton" layoutX="38.0"
```

```
            mnemonicParsing="false"
            onAction="#startButtonPressed" text="St
      </children>
   </Pane>
```


a) Initial Rectangle
T..
Start Animations
b) Rectangle undergoing parallel fill and stroke transitions
T..
Start Animations
T..
Start Animations
T..
Start Animations
c) Rectangle undergoing a fade transition
T..
Start Animations
T..
Start Animations
T..
Start Animations

d) `Rectangle` undergoing a rotate transition

e) `Rectangle` undergoing a path transition

f) `Rectangle` undergoing a scale transition

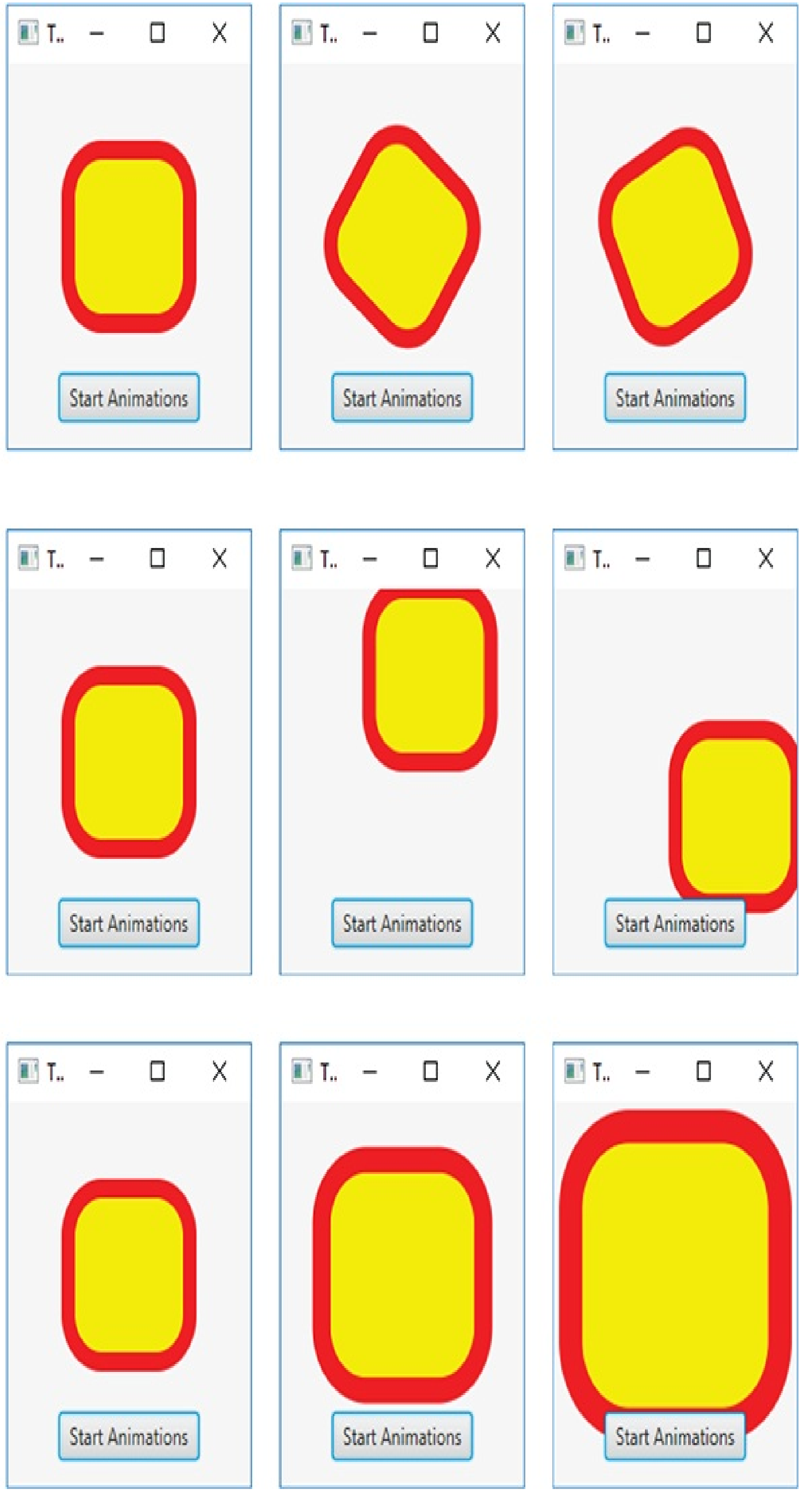


Fig. 22.10

FXML for a `Rectangle` and `Button`.

Description

22.7.2 `TransitionAnimationsController` Class

Figure 22.11 shows this app's controller class, which defines the `startButton`'s event handler (lines 25–87). This event handler defines several animations that are played in sequence.

```
1   // Fig. 22.11: TransitionAnimationsController.jav
2   // Applying Transition animations to a Rectangle.
3   import javafx.animation.FadeTransition;
4   import javafx.animation.FillTransition;
5   import javafx.animation.Interpolator;
6   import javafx.animation.ParallelTransition;
7   import javafx.animation.PathTransition;
8   import javafx.animation.RotateTransition;
9   import javafx.animation.ScaleTransition;
10  import javafx.animation.SequentialTransition;
11  import javafx.animation.StrokeTransition;
12  import javafx.event.ActionEvent;
13  import javafx.fxml.FXML;
14  import javafx.scene.paint.Color;
15  import javafx.scene.shape.LineTo;
16  import javafx.scene.shape.MoveTo;
17  import javafx.scene.shape.Path;
18  import javafx.scene.shape.Rectangle;
19  import javafx.util.Duration;
20
21  public class TransitionAnimationsController {
```

```
    @FXML private Rectangle rectangle;

    // configure and start transition animations
    @FXML
    private void startButtonPressed(ActionEvent e
       // transition that changes a shape's fill
       FillTransition fillTransition =
          new FillTransition(Duration.seconds(1))
       fillTransition.setToValue(Color.CYAN);
       fillTransition.setCycleCount(2);

       // each even cycle plays transition in rev
       fillTransition.setAutoReverse(true);

       // transition that changes a shape's strok
       StrokeTransition strokeTransition =
          new StrokeTransition(Duration.seconds(1
       strokeTransition.setToValue(Color.BLUE);
       strokeTransition.setCycleCount(2);
       strokeTransition.setAutoReverse(true);

       // parallelizes multiple transitions
       ParallelTransition parallelTransition =
          new ParallelTransition(fillTransition,

       // transition that changes a node's opacit
       FadeTransition fadeTransition =
          new FadeTransition(Duration.seconds(1))
       fadeTransition.setFromValue(1.0); // opaqu
       fadeTransition.setToValue(0.0); // transpa
       fadeTransition.setCycleCount(2);
       fadeTransition.setAutoReverse(true);

       // transition that rotates a node
       RotateTransition rotateTransition =
          new RotateTransition(Duration.seconds(1
       rotateTransition.setByAngle(360.0);
       rotateTransition.setCycleCount(2);
       rotateTransition.setInterpolator(Interpola
       rotateTransition.setAutoReverse(true);
```

```
        // transition that moves a node along a Pa
        Path path = new Path(new MoveTo(45, 45), n
           new LineTo(90, 0), new LineTo(90, 90),
        PathTransition translateTransition =
           new PathTransition(Duration.seconds(2),
        translateTransition.setCycleCount(2);
        translateTransition.setInterpolator(Interp
        translateTransition.setAutoReverse(true);

        // transition that scales a shape to make
        ScaleTransition scaleTransition =
           new ScaleTransition(Duration.seconds(1)
        scaleTransition.setByX(0.75);
        scaleTransition.setByY(0.75);
        scaleTransition.setCycleCount(2);
        scaleTransition.setInterpolator(Interpolat
        scaleTransition.setAutoReverse(true);

        // transition that applies a sequence of t
        SequentialTransition sequentialTransition
           new SequentialTransition (rectangle, pa
              fadeTransition, rotateTransition, tr
              scaleTransition);
        sequentialTransition.play(); // play the t
    }
}
```

Fig. 22.11

Applying `Transition` animations to a `Rectangle`.

`FillTransition`

Lines 28–34 configure a one-second `FillTransition` that changes a shape's fill color. Line 30 specifies the color (`CYAN`) to which the fill will transition. Line 31 sets the animations cycle count to 2—this specifies the number of iterations of the transition to perform over the specified duration. Line 34 specifies that the animation should automatically play itself in reverse once the initial transition is complete. For this animation, during the first cycle the fill color changes from the original fill color to `CYAN`, and during the second cycle the animation transitions back to the original fill color.

StrokeTransition

Lines 37–41 configure a one-second `StrokeTransition` that changes a shape's stroke color. Line 39 specifies the color (`BLUE`) to which the stroke will transition. Line 40 sets the animations cycle count to 2, and line 41 specifies that the animation should automatically play itself in reverse once the initial transition is complete. For this animation, during the first cycle the stroke color changes from the original stroke color to `BLUE`, and during the second cycle the animation transitions back to the original stroke color.

ParallelTransition

Lines 44–45 configure a `ParallelTransition` that performs multiple transitions at the same time (that is, in parallel). The `ParallelTransition` constructor receives a variable number of `Transition`s as a comma-separated list. In this case, the `FillTransition` and `StrokeTransition` will be performed in parallel on the app's `Rectangle`.

FadeTransition

Lines 48–53 configure a one-second `FadeTransition` that changes a `Node`'s opacity. Line 50 specifies the initial opacity—1.0 is fully opaque. Line 51 specifies the final opacity—0.0 is fully transparent. Once again, we set the cycle count to 2 and specified that the animation should auto-reverse itself.

RotateTransition

Lines 56–61 configure a one-second `RotateTransition` that rotates a `Node`. You can rotate a `Node` by a specified number of degrees (line 58) or you can use other `RotateTransition` methods to specify a start angle and end angle. Each `Transition` animation uses an `Interpolator` to calculate new property values throughout the animation's duration. The default is a `LINEAR Interpolator` which evenly divides the property value changes over the animation's duration. For the `RotateTransition`, line 60 uses the `Interpolator EASE_BOTH`, which changes the rotation slowly at first (known as "easing in"), speeds up the rotation in the middle of the animation, then slows the rotation again to complete the animation (known as “easing out”). For a list of all the predefined `Interpolator`s, see

```
https://docs.oracle.com/javase/8/javafx/api/javafx/an
```

`PathTransition`

Lines 64–70 configure a two-second `PathTransition` that changes a shape’s position by moving it along a `Path`. Lines 64–65 create the `Path`, which is specified as the second argument to the `PathTransition` constructor. A `LineTo` object draws a straight line from the previous `PathElement`’s endpoint to the specified location. Line 69 specifies that this animation should use the `Interpolator EASE_IN`, which changes the position slowly at first, before performing the animation at full speed.

`ScaleTransition`

Lines 73–79 configure a one-second `ScaleTransition` that changes a `Node`’s size. Line 75 specifies that the object will be scaled 75% larger along the *x*-axis (i.e., horizontally), and line 76 specifies that the object will be scaled 75% larger along the *y*-axis (i.e., vertically). Line 78 specifies that this animation should use the `Interpolator EASE_OUT`, which begins scaling the shape at full speed, then slows down as the animation completes.

SequentialTransition

Lines 82–86 configure a `SequentialTransition` that performs a sequence of transitions—as each completes, the next one in the sequence begins executing. The `SequentialTransition` constructor receives the `Node` to which the sequence of animations will be applied, followed by a comma-separated list of `Transition`s to perform. In fact, every transition animation class has a constructor that enables you to specify a `Node`. For this example, we did not specify `Node`s when creating the other transitions, because they're all applied by the `SequentialTransition` to the `Rectangle`. Every `Transition` has a `play` method (line 86) that begins the animation. Calling `play` on the `SequentialTransition` automatically calls `play` on each animation in the sequence.

22.8 Timeline Animations

In this section, we continue our animation discussion with a `Timeline` animation that bounces a `Circle` object around the app's `Pane` over time. A `Timeline` animation can change any `Node` property that's modifiable. You specify how to change property values with one or more `KeyFrame` objects that the `Timeline` animation performs in sequence. For this app, we'll specify a single `KeyFrame` that modifies a `Circle`'s location, then we'll play that `KeyFrame` indefinitely. Figure 22.12 shows the app's FXML, which defines a `Circle` object with a five-pixel black border and the fill color `DODGERBLUE`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 22.12: TimelineAnimation.fxml -->
<!-- FXML for a Circle that will be animated by

<?import javafx.scene.layout.Pane?>
<?import javafx.scene.shape.Circle?>

<Pane id="Pane" fx:id="pane" prefHeight="400.0"
   prefWidth="600.0" xmlns:fx="http://javafx.com
   xmlns="http://javafx.com/javafx/8.0.60"
   fx:controller="TimelineAnimationController">
   <children>
      <Circle fx:id="c" fill="DODGERBLUE" layout
         radius="40.0" stroke="BLACK" strokeType
         strokeWidth="5.0" />
   </children>
```

```
</Pane>
```

Fig. 22.12

FXML for a `Circle` that will be animated by the controller.

The application’s controller (Fig. 22.13) configures then plays the `Timeline` animation in the `initialize` method. Lines 22–45 define the animation, line 48 specifies that the animation should cycle indefinitely (until the program terminates or the animation’s `stop` method is called) and line 49 plays the animation.

```
1  // Fig. 22.13: TimelineAnimationController.java
2  // Bounce a circle around a window using a Timel
3  import java.security.SecureRandom;
4  import javafx.animation.KeyFrame;
5  import javafx.animation.Timeline;
6  import javafx.event.ActionEvent;
7  import javafx.event.EventHandler;
8  import javafx.fxml.FXML;
9  import javafx.geometry.Bounds;
10 import javafx.scene.layout.Pane;
11 import javafx.scene.shape.Circle;
12 import javafx.util.Duration;
13
14 public class TimelineAnimationController {
15    @FXML Circle c;
16    @FXML Pane pane;
17
18    public void initialize() {
19       SecureRandom random = new SecureRandom();
```

```

      // define a timeline animation
      Timeline timelineAnimation = new Timeline(
         new KeyFrame(Duration.millis(10),
            new EventHandler<ActionEvent>() {
               int dx = 1 + random.nextInt(5);
               int dy = 1 + random.nextInt(5);

               // move the circle by the dx and
               @Override
               public void handle(final ActionEv
                  c.setLayoutX(c.getLayoutX() +
                  c.setLayoutY(c.getLayoutY() +
                  Bounds bounds = pane.getBounds

                  if (hitRightOrLeftEdge(bounds)
                     dx *= -1;
                  }

                  if (hitTopOrBottom(bounds)) {
                     dy *= -1;
                  }
               }
            }
         )
      );

      // indicate that the timeline animation sh
      timelineAnimation.setCycleCount(Timeline.I
      timelineAnimation.play();
   }

   // determines whether the circle hit the left
   private boolean hitRightOrLeftEdge(Bounds bou
      return (c.getLayoutX() <= (bounds.getMinX(
         (c.getLayoutX() >= (bounds.getMaxX() -
   }

   // determines whether the circle hit the top
   private boolean hitTopOrBottom(Bounds bounds)
```

```
        return (c.getLayoutY() <= (bounds.getMinY(
           (c.getLayoutY() >= (bounds.getMaxY() -
     }
  }
```

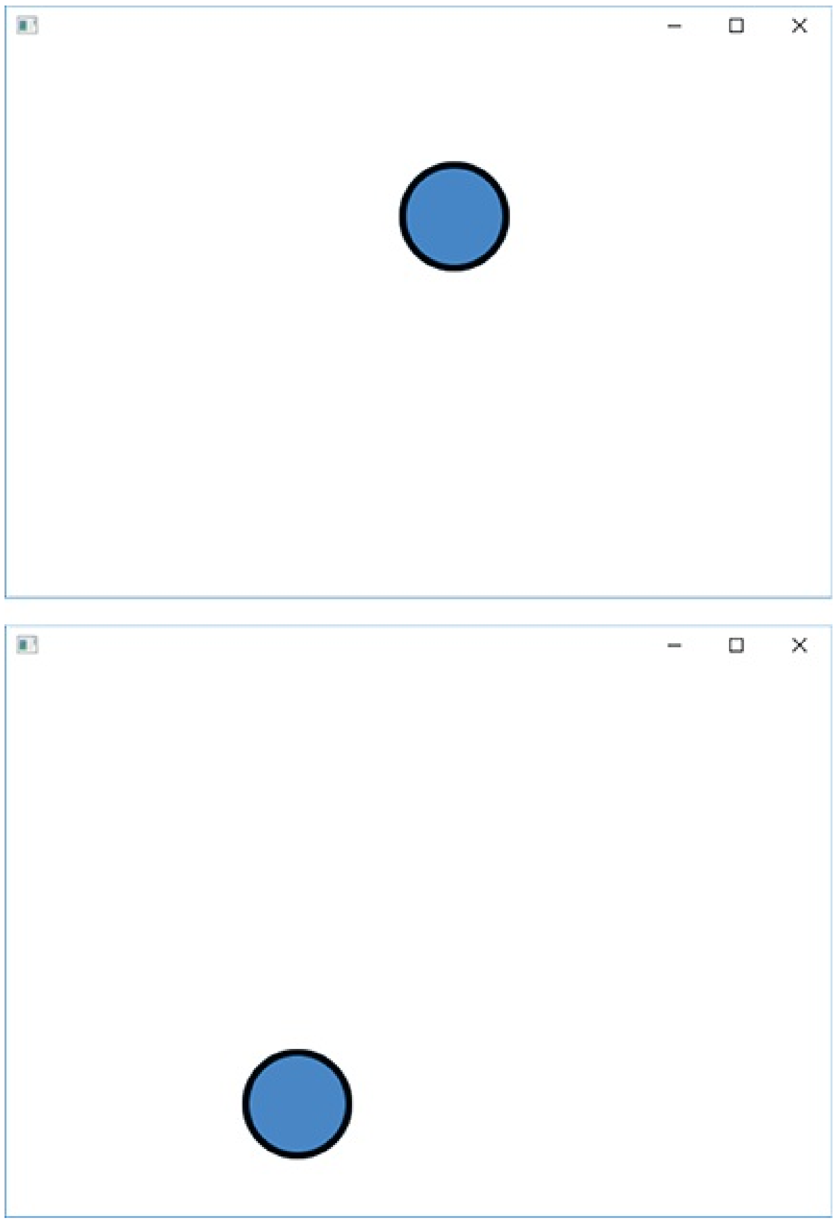


Fig. 22.13

Bounce a circle around a window using a `Timeline` animation.

Description

Creating the `Timeline`

The `Timeline` constructor used in lines 22–45 can receive a comma-separated list of `KeyFrame`s as arguments—in this case, we pass a single `KeyFrame`. Each `KeyFrame` issues an `ActionEvent` at a particular time in the animation. The app can respond to the event by changing a `Node`’s property values. The `KeyFrame` constructor used here specifies that, after 10 milliseconds, the `ActionEvent` will occur. Because we set the `Timeline`’s cycle count to `Timeline.INDEFINITE`, the `Timeline` will perform this `KeyFrame` every 10 milliseconds. Lines 24–43 define the `EventHandler` for the `KeyFrame`’s `ActionEvent`.

`KeyFrame`’s `EventHandler`

In the `KeyFrame`’s `EventHandler` we define instance variables `dx` and `dy` (lines 25–26) and initialize them with randomly chosen values that will be used to change the `Circle`’s *x*- and *y*-coordinates each time the `KeyFrame` plays. The `EventHandler`’s `handle` method (lines 29–42) adds these values to the `Circle`'s *x*- and *y*-coordinates (lines 31–32). Next, lines 35–41 perform bounds checking to determine whether the `Circle` has collided with any of the `Pane`'s edges. If the `Circle` hits the left or right edge, line 36 multiplies the value of `dx` by `-1` to reverse the `Circle`'s horizontal direction. If the `Circle` hits the top or bottom edge, line 40 multiplies the value of `dx` by `-1` to reverse the `Circle`'s horizontal direction.

22.9 Frame-by-Frame Animation with `AnimationTimer`

A third way to implement JavaFX animations is via an `AnimationTimer` (package `javafx.animation`), which enables you to define frame-by-frame animations. You specify how your objects should move in a given frame, then JavaFX aggregates all of the drawing operations and displays the frame. This can be used with objects in the scene graph or to draw shapes in a `Canvas`. JavaFX calls the `handle` method of every `AnimationTimer` before it draws an animation frame.

For smooth animation, JavaFX tries to display animation frames at 60 frames per second. This frame rate varies based on the animation's complexity, the processor speed and how busy the processor is at a given time. For this reason, method `handle` receives a time stamp in nanoseconds (billionths of a second) that you can use to determine the elapsed time since the last animation frame, then you can scale the movements of your objects accordingly. This enables you to define animations that operate at the same overall speed, regardless of the frame rate on a given device.

Figure 22.14 reimplements the animation in Fig. 22.13 using an `AnimationTimer`. The FXML is identical (other than the filename and controller class name). Much of the code is identical to Fig. 22.13—we've highlighted the key changes, which we discuss below.

```
// Fig. 22.14: BallAnimationTimerController.java
// Bounce a circle around a window using an Anim
import java.security.SecureRandom;
import javafx.animation.AnimationTimer;
import javafx.fxml.FXML;
import javafx.geometry.Bounds;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.util.Duration;

public class BallAnimationTimerController {
   @FXML private Circle c;
   @FXML private Pane pane;

   public void initialize() {
      SecureRandom random = new SecureRandom();

      // define a timeline animation
      AnimationTimer timer = new AnimationTimer(
         int dx = 1 + random.nextInt(5);
         int dy = 1 + random.nextInt(5);
         int velocity = 60; // used to scale dis
         long previousTime = System.nanoTime();

         // specify how to move Circle for curre
         @Override
         public void handle(long now) {
            double elapsedTime = (now - previous
            previousTime = now;
            double scale = elapsedTime * velocit

            Bounds bounds = pane.getBoundsInLoca
            c.setLayoutX(c.getLayoutX() + dx * s
```

```
            c.setLayoutY(c.getLayoutY() + dy * s

            if (hitRightOrLeftEdge(bounds)) {
               dx *= -1;
            }

            if (hitTopOrBottom(bounds)) {
               dy *= -1;
            }
         }
      };

      timer.start();
   }

   // determines whether the circle hit left/rig
   private boolean hitRightOrLeftEdge(Bounds bou
      return (c.getLayoutX() <= (bounds.getMinX(
         (c.getLayoutX() >= (bounds.getMaxX() -
   }

   // determines whether the circle hit top/bott
   private boolean hitTopOrBottom(Bounds bounds)
      return (c.getLayoutY() <= (bounds.getMinY(
         (c.getLayoutY() >= (bounds.getMaxY() -
   }
}
```

Fig. 22.14

Bounce a circle around a window using an `AnimationTimer` subclass.

Extending `abstract` Class `AnimationTimer`

Class `AnimationTimer` is an `abstract` class, so you must create a subclass. In this example, lines 19–44 create an anonymous inner class that extends `AnimationTimer`. Lines 20–23 define the anonymous inner class's instance variables:

- As in Fig. 22.13, `dx` and `dy` incrementally change the `Circle`'s position and are chosen randomly so the `Circle` moves at different speeds during each execution.
- Variable `velocity` is used as a multiplier to determine the actual distance moved in each animation frame—we discuss this again momentarily.
- Variable `previousTime` represents the time stamp (in nanoseconds) of the previous animation frame—this will be used to determine the elapsed time between frames. We initialized `previousTime` to `System.nanoTime()`, which returns the number of nanoseconds since the JVM launched the app. Each call to `handle` also receives as its argument the number of nanoseconds since the JVM launched the app.

Overriding Method `handle`

Lines 26–43 override `AnimationTimer` method `handle`, which specifies what to do during each animation frame:

- Line 28 calculates the `elapsedTime` in *seconds* since the last animation frame. If method `handle` truly is called 60 times per second, the elapsed time between frames will be approximately 0.0167 seconds—that is, 1/60 of a second.

- Line 29 stores the time stamp in `previousTime` for use in the *next* animation frame.
- When we change the `Circle`'s `layoutX` and `layoutY` (lines 33–34), we multiply `dx` and `dy` by the `scale` (line 30). In Fig. 22.13, the `Circle`'s speed was determined by moving between one and five pixels along the *x*- and *y*-axes every 10 milliseconds—the larger the values, the faster the `Circle` moved. If we scale `dx` or `dy` by just `elapsedTime`, we'd move the `Circle` only small fractions of `dx` and `dy` during each frame—approximately 0.0167 seconds (1/60 of a second) to 0.083 seconds (5/60 of a second), based on their randomly chosen values. For this reason, we multiply the `elapsedTime` by the `velocity` (60) to scale the movement in each frame. This results in values that are approximately one to five pixels, as in Fig. 22.13.

22.10 Drawing on a `Canvas`

So far, you've displayed and manipulated JavaFX two-dimensional shape objects that reside in the scene graph. In this section, we demonstrate similar drawing capabilities using the `javafx.scene.canvas` package, which contains two classes:

- Class `Canvas` is a subclass of `Node` in which you can draw graphics.
- Class `GraphicsContext` performs the drawing operations on a `Canvas`.

As you'll see, a `GraphicsContext` object enables you to specify the same drawing characteristics that you've previously used on `Shape` objects. However, with a `GraphicsContext`, you must set these characteristics and draw the shapes programmatically. To demonstrate various `Canvas` capabilities, Fig. 22.15 re-implements Section 22.3's `BasicShapes` example. Here, you'll see various JavaFX classes and `enums` (from packages `javafx.scene.image`, `javafx.scene.paint` and `javafx.scene.shape`) that JavaFX's CSS capabilities use behind the scenes to style `Shapes`.

Performance Tip 22.1

A `Canvas` *typically is preferred for performance-oriented graphics, such as those in games with moving elements.*

```
// Fig. 22.15: CanvasShapesController.java
// Drawing on a Canvas.
import javafx.fxml.FXML;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import javafx.scene.paint.Color;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.ImagePattern;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.RadialGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.StrokeLineCap;

public class CanvasShapesController {
   // instance variables that refer to GUI compo
   @FXML private Canvas drawingCanvas;

   // draw on the Canvas
   public void initialize() {
      GraphicsContext gc = drawingCanvas.getGrap
      gc.setLineWidth(10); // set all stroke wid

      // draw red line
      gc.setStroke(Color.RED);
      gc.strokeLine(10, 10, 100, 100);

      // draw green line
      gc.setGlobalAlpha(0.5); // half transparen
      gc.setLineCap(StrokeLineCap.ROUND);
      gc.setStroke(Color.GREEN);
      gc.strokeLine(100, 10, 10, 100);

      gc.setGlobalAlpha(1.0); // reset alpha tra

```

```
        // draw rounded rect with red border and y
        gc.setStroke(Color.RED);
        gc.setFill(Color.YELLOW);
        gc.fillRoundRect(120, 10, 90, 90, 50, 50);
        gc.strokeRoundRect(120, 10, 90, 90, 50, 50

        // draw circle with blue border and red/wh
        gc.setStroke(Color.BLUE);
        Stop[] stopsRadial =
           {new Stop(0, Color.RED), new Stop(1, Co
        RadialGradient radialGradient = new Radial
           0.6, true, CycleMethod.NO_CYCLE, stopsR
        gc.setFill(radialGradient);
        gc.fillOval(230, 10, 90, 90);
        gc.strokeOval(230, 10, 90, 90);

        // draw ellipse with green border and imag
        gc.setStroke(Color.GREEN);
        gc.setFill(new ImagePattern(new Image("yel
        gc.fillOval(340, 10, 200, 90);
        gc.strokeOval(340, 10, 200, 90);

        // draw arc with purple border and cyan/wh
        gc.setStroke(Color.PURPLE);
        Stop[] stopsLinear =
           {new Stop(0, Color.CYAN), new Stop(1, C
        LinearGradient linearGradient = new Linear
           true, CycleMethod.NO_CYCLE, stopsLinear
        gc.setFill(linearGradient);
        gc.fillArc(560, 10, 90, 90, 45, 270, ArcTy
        gc.strokeArc(560, 10, 90, 90, 45, 270, Arc
    }
}
```

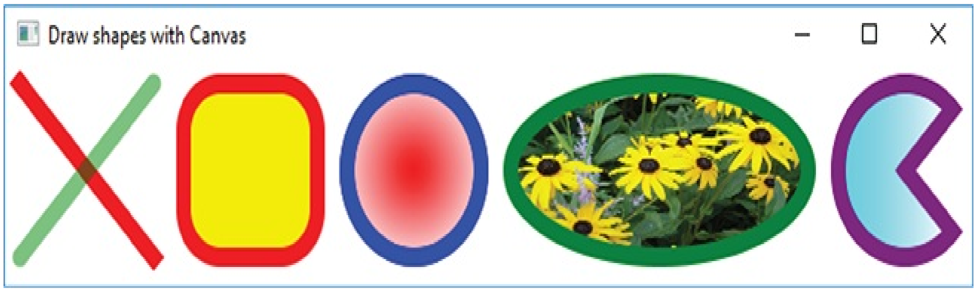


Fig. 22.15

Drawing on a `Canvas`.

Description

Obtaining the `GraphicsContext`

To draw on a `Canvas`, you first obtain its `GraphicsContext` by calling `Canvas` method `getGraphicsContext2D` (line 22).

Setting the Line Width for All the Shapes

When you set a `GraphicsContext`'s drawing characteristics, they're applied (as appropriate) to all subsequent shapes you draw. For example, line 23 calls setLineWidth to specify the GraphicsContext's line thickness (10). All subsequent GraphicsContext method calls that draw lines or shape borders will use this setting. This is similar to the -fx-strokewidth CSS attribute we specified for all shapes in Fig. 22.4.

Drawing Lines

Lines 26–33 draw the red and green lines:

- GraphicsContext's setStroke method (lines 26 and 32) specifies the Paint object (package javafx.scene.paint) used to draw the line. The Paint can be any of the subclasses Color, ImagePattern, LinearGradient or RadialGradient (all from package javafx.scene.paint). We demonstrate each of these in this example —Color for the lines and Color, ImagePattern, LinearGradient or RadialGradient as the fills for other shapes.
- GraphicsContext's strokeLine method (lines 27 and 33) draws a line using the current Paint object that's set as the stroke. The four arguments are the *x*-*y* coordinates of the start and end points, respectively.
- GraphicsContext's setLineCap method (line 31) sets line cap, like the CSS property -fx-stroke-line-cap in Fig. 22.4. The argument to this method must be constant from the enum StrokeLineCap (package javafx.scene.shape). Here we round the line ends.
- GraphicsContext's setGlobalAlpha method (line 30) sets the alpha transparency of all subsequent shapes you draw. For the green line we used 0.5, which is 50% transparent. After drawing the green line, we reset this to the default 1.0 (line 35), so that subsequent shapes are fully opaque.

Drawing a Rounded Rectangle

Lines 38–41 draw a rounded rectangle with a red border:

- Line 38 sets the border color to `Color.RED`.
- `GraphicsContext`'s `setFill` method (lines 39, 49, 55 and 65) specifies the `Paint` object that fills a shape. Here we fill the rectangle with `Color.YELLOW`.
- `GraphicsContext`'s `fillRoundRect` method draws a *filled* rectangle with rounded corners using the current `Paint` object set as the fill. The method's first four arguments represent the rectangle's upper-left *x*-coordinate, upper-left *y*-coordinate, width and height, respectively. The last two arguments represent the arc width and arc height that are used to round the corners. These work identically to the CSS properties `-fx-arc-width` and `-fx-arc-height` properties in Fig. 22.4. `GraphicsContext` also provides a `fillRect` method that draws a rectangle without rounded corners.
- `GraphicsContext`'s `strokeRoundRect` method has the same arguments as `fillRoundRect`, but draws a hollow rectangle with rounded corners. `GraphicsContext` also provides a `strokeRect` method that draws a rectangle without rounded corners.

Drawing a Circle with a `RadialGradient` Fill

Lines 44–51 draw a circle with a blue border and a red-white, radial-gradient fill. Line 44 sets the border color to `Color.BLUE`. Lines 45–48 configure the `RadialGradient`—these lines perform the same tasks as the CSS function `radial-gradient` in Fig. 22.4.

First, lines 45–46 create an array of `Stop` objects (package `javafx.scene.paint`) representing the color stops. Each `Stop` has an offset from 0.0 to 1.0 representing the offset (as a percentage) from the gradient's start point and a `Color`. Here the `Stop`s indicate that the radial gradient will transition from red at the gradient's start point to white at its end point.

The `RadialGradient` constructor (lines 47–48) receives as arguments:

- the focus angle, which specifies the direction of the radial gradient's focal point from the gradient's center,
- the distance of the focal point as a percentage (0.0–1.0),
- the center point's *x* and *y* location as percentages (0.0–1.0) of the width and height for the shape being filled,
- a `boolean` indicating whether the gradient should scale to fill its shape,
- a constant from the `CycleMethod` `enum` (package `javafx.scene.paint`) indicating how the color stops are applied, and
- an array of `Stop` objects—this can also be a comma-separated list of `Stop`s or a `List<Stop>` object.

This creates a red-white radial gradient that starts with solid red at the center of the shape and—at 60% of the radial gradient's radius—transitions to white. Line 49 sets the fill to the new `radialGradient`, then lines 50–51 call `GraphicsContext`'s `fillOval` and `strokeOval` methods to draw a filled oval and hollow oval, respectively. Each method receives as arguments the upper-left *x*-

coordinate, upper-left *y*-coordinate, width and height of the rectangular area (that is, the bounding box) in which the oval should be drawn. Because the width and height are the same, these calls draw circles.

Drawing an Oval with an `ImagePattern` Fill

Lines 54–57 draw an oval with a green border and containing an image:

- Line 54 sets the border color to `Color.GREEN`.
- Line 55 sets the fill to an `ImagePattern`—a subclass of `Paint` that loads an `Image`, either from the local system or from a URL specified as a `String`. `ImagePattern` is the class used by the CSS function `image-pattern` in Fig. 22.4.
- Lines 56–57 draw a filled oval and a hollow oval, respectively.

Drawing an Arc with a `LinearGradient` Fill

Lines 60–67 draw an arc with a purple border and filled with a cyan-white linear gradient:

- Line 60 sets the border color to `Color.PURPLE`.
- Lines 61–64 configure the `LinearGradient`, which is the class used by CSS function `linear-gradient` in Fig. 22.4. The constructor's first four arguments are the endpoint coordinates that represent the

direction and angle of the gradient—if the *x*-coordinates are the same, the gradient is vertical; if the *y*-coordinates are the same, the gradient is horizontal and all other linear gradients follow a diagonal line. When these values are specified in the range 0.0 to 1.0 and the constructor's fifth argument is `true`, the gradient is scaled to fill the shape. The next argument is the `CycleMethod`. The last argument is an array of `Stop` objects—again, this can be a comma-separated list of `Stop`s or a `List<Stop>` object.

Lines 66–67 call `GraphicsContext`'s `fillArc` and `strokeArc` methods to draw a filled arc and hollow arc, respectively. Each method receives as arguments

- the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height of the rectangular area (that is, the bounding box) in which the oval should be drawn,
- the start angle and sweep of the arc in degrees, and
- a constant from the `ArcType` `enum` (package `javafx.scene.shape`)

Additional `GraphicsContext` Features

There are many additional `GraphicsContext` features, which you can explore at

```
https://docs.oracle.com/javase/8/javafx/api/javafx/sc
```

Some of the capabilities that we did not discuss here include:

- Drawing and filling text—similar to the font features in Section 22.2.
- Drawing and filling polylines, polygons and paths—similar to the corresponding `Shape` subclasses in Section 22.4.
- Applying effects and transforms—similar to the transforms in Section 22.5.
- Drawing images.
- Manipulating the individual pixels of a drawing in a `Canvas` via a `PixelWriter`.
- Saving and restoring graphics characteristics via the `save` and `restore` methods.

22.11 Three-Dimensional Shapes

8

Throughout this chapter, we've demonstrated many two-dimensional graphics capabilities. In Java SE 8, JavaFX added several three-dimensional shapes and corresponding capabilities. The three-dimensional shapes are subclasses of `Shape3D` from the package `javafx.scene.shape`. In this section, you'll use Scene Builder to create a `Box`, a `Cylinder` and a `Sphere` and specify several of their properties. Then, in the app's controller, you'll create so-called *materials* that apply color and images to the 3D shapes.

FXML for the `Box`, `Cylinder` and `Sphere`

Figure 22.16 shows the completed FXML that we created with Scene Builder:

- Lines 16–21 define the `Box` object.
- Lines 22–27 define the `Cylinder` object.
- Lines 28–29 define the `Sphere` object.

We dragged objects of each of these `Shape3D` subclasses from the Scene Builder **Library**’s **Shapes** section onto the design area and gave them the **fx:id** values `box`, `cylinder` and `sphere`, respectively. We also set the controller to `ThreeDimensionalShapesController`.[5]

5. At the time of this writing, when you drag three-dimensional shapes onto the Scene Builder design area, their dimensions are set to small values by default—a `Box`’s **Width**, **Height** and **Depth** are set to `2`, a `Cylinder`’s **Height** and **Radius** are set to `2` and `0.77`, and a `Sphere`’s **Radius** is set to `0.77`. You may need to select them in the **Hierarchy** pane to set their properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ThreeDimensionalShapes.fxml -->
<!-- FXML that displays a Box, Cylinder and Sphe

<?import javafx.geometry.Point3D?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.shape.Box?>
<?import javafx.scene.shape.Cylinder?>
<?import javafx.scene.shape.Sphere?>

<Pane prefHeight="200.0" prefWidth="510.0"
   xmlns="http://javafx.com/javafx/8.0.60"
   xmlns:fx="http://javafx.com/fxml/1"
   fx:controller="ThreeDimensionalShapesControll
   <children>
      <Box fx:id="box" depth="100.0" height="100
         layoutY="100.0" rotate="30.0" width="10
         <rotationAxis>
            <Point3D x="1.0" y="1.0" z="1.0" />
         </rotationAxis>
      </Box>
      <Cylinder fx:id="cylinder" height="100.0"
         layoutY="100.0" radius="50.0" rotate="-
         <rotationAxis>
            <Point3D x="1.0" y="1.0" z="1.0" />
         </rotationAxis>
```

```
        </Cylinder>
      <Sphere fx:id="sphere" layoutX="430.0" lay
            radius="60.0" />
      </children>
   </Pane<
```

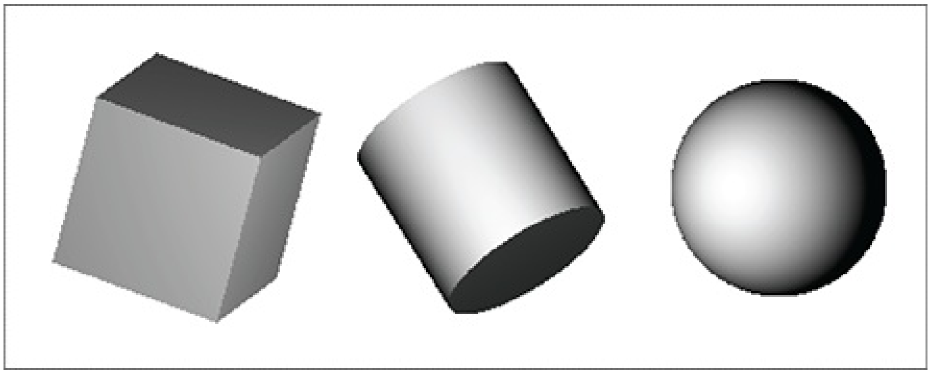


Fig. 22.16

FXML that displays a `Box`, `Cylinder` and `Sphere`.

Description

As you can see in the screen capture of Figure 22.16, all three shapes initially are gray. The shading you see in Scene Builder comes from the scene’s default lighting. Though we do not use them in this example, package `javafx.scene`’s `AmbientLight` and `PointLight` classes can be used to add your own lighting effects. You can also use camera objects to view the scene from different angles and distances. These are located in the Scene Builder **Library**’s **3D** section. For more information on lighting and cameras, see

```
https://docs.oracle.com/javase/8/javafx/graphics-tuto
```

We ask you to investigate these capabilities and use them in Exercise 22.11.

Box Properties

Configure the `Box`'s properties in Scene Builder as follows:

- Set **Width**, **Height** and **Depth** to `100`, making a cube. The depth is measured along the *z*-axis which runs perpendicular to your screen—when you move objects along the *z*-axis they get bigger as they're brought toward you and smaller as they're moved away from you.
- Set **Layout X** and **Layout Y** to `100` to specify the location of the cube.
- Set **Rotate** to `30` to specify the rotation angle in degrees. Positive values rotate counter-clockwise.
- For **Rotation Axis**, set the **X**, **Y** and **Z** values to `1` to indicate that the **Rotate** angle should be used to rotate the cube 30 degrees around *each* axis.

To see how the **Rotate** angle and **Rotation Axis** values affect the `Box`'s rotation, try setting two of the three **Rotation Axis** values to `0`, then changing the **Rotate** angle.

`Cylinder` Properties

Configure the `Cylinder`'s properties in Scene Builder as follows:

- Set **Height** to `100.0` and **Radius** to `50`.
- Set **Layout X** and **Layout Y** to `265` and `100`, respectively.
- Set **Rotate** to `-45` to specify the rotation angle in degrees. Negative values rotate clockwise.
- For **Rotation Axis**, set the **X**, **Y** and **Z** values to `1` to indicate that the **Rotate** angle should be applied to all three axes.

`Sphere` Properties

Configure the `Sphere`'s properties in Scene Builder as follows:

- Set **Radius** to `60`.
- Set **Layout X** and **Layout Y** to `430` and `100`, respectively.

`ThreeDimensionalShapesController` Class

Figure 22.17 shows this app's controller and final output. The colors and images you see on the final shapes are created by applying so-called materials to the shapes. JavaFX class `PhongMaterial` (package `javafx.scene.paint`) is used to define materials. The name "Phong" is a 3D graphics term—*phong shading* is technique for applying color and shading to 3D surfaces. For more details on this technique, visit

```
https://en.wikipedia.org/wiki/Phong_shading
```

```
// Fig. 22.17: ThreeDimensionalShapesController.
// Setting the material displayed on 3D shapes.
import javafx.fxml.FXML;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.image.Image;
import javafx.scene.shape.Box;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.Sphere;

public class ThreeDimensionalShapesController {
   // instance variables that refer to 3D shapes
   @FXML private Box box;
   @FXML private Cylinder cylinder;
   @FXML private Sphere sphere;

   // set the material for each 3D shape
   public void initialize() {
      // define material for the Box object
      PhongMaterial boxMaterial = new PhongMater
      boxMaterial.setDiffuseColor(Color.CYAN);
      box.setMaterial(boxMaterial);

      // define material for the Cylinder object
      PhongMaterial cylinderMaterial = new Phong
      cylinderMaterial.setDiffuseMap(new Image("
      cylinder.setMaterial(cylinderMaterial);

      // define material for the Sphere object
      PhongMaterial sphereMaterial = new PhongMa
      sphereMaterial.setDiffuseColor(Color.RED);
      sphereMaterial.setSpecularColor(Color.WHIT
      sphereMaterial.setSpecularPower(32);
      sphere.setMaterial(sphereMaterial);
   }
```

```
36 }
```

Fig. 22.17

Setting the material displayed on 3D shapes.

Description

`PhongMaterial` for the `Box`

Lines 20–22 configure and set the `Box` object's `PhongMaterial`. Method `setDiffuseColor` sets the color that's applied to the `Box`'s surfaces (that is, sides). The scene's lighting effects determine the shades of the color applied to each visible surface. These shades change, based on the angle from which the light shines on the objects.

`PhongMaterial` for the `Cylinder`

Lines 25–27 configure and set the `Cylinder` object's `PhongMaterial`. Method `setDiffuseMap` sets the `Image` that's applied to the `Cylinder`'s surfaces. Again, the scene's lighting affects how the image is shaded on the surfaces. In the output, notice that the image is darker at the left and right edges (where less light reaches) and barely visible on the bottom (where almost no light reaches).

`PhongMaterial` for the `Sphere`

Lines 30–34 configure and set the `Sphere` object's `PhongMaterial`. We set the diffuse color to red. Method `setSpecularColor` sets the color of a bright spot that makes a 3D shape appear shiny. Method `setSpecularPower` determines the intensity of that spot. Try experimenting with different specular powers to see changes in the bright spot's intensity.

22.12 Wrap-Up

In this chapter, we completed our discussion of JavaFX that began in Chapters 12 and 13. Here, we presented various JavaFX graphics and multimedia capabilities.

We used external Cascading Style Sheets (CSS) to customize the appearance of JavaFX `Node`s, including `Label`s and objects of various `Shape` subclasses. We displayed two-dimensional shapes, including lines, rectangles, circles, ellipses, arcs, polylines, polygons and custom paths.

We showed how to apply a transform to a `Node`, rotating 18 `Polygon` objects around a specific point to create a circle of star shapes. We created a simple video player using class `Media` to specify the video's location, class `MediaPlayer` to load the video and control its playback and class `MediaView` to display the video.

We animated `Node`s with `Transition` and `Timeline` animations that change `Node` properties to new values over time. We used built-in `Transition` animations to change specific JavaFX `Node` properties (such as a `Node`'s stroke and fill colors, opacity, angle of rotation and scale). We used `Timeline` animations with `KeyFrame`s to bounce a `Circle` around a window, and showed that such animations can be used to change any modifiable `Node` property. We also showed how to create frame-by-frame animations with `AnimationTimer`.

Next, we presented various capabilities for drawing on a `Canvas Node` using a `GraphicsContext` object. You saw that `GraphicsContext` supports many of the same drawing characteristics and shapes that you can implement with `Shape Node`s. Finally, we showed the three-dimensional shapes `Box`, `Cylinder` and `Sphere`, and demonstrated how to use materials to apply color and images to them. For more information on JavaFX, visit the FX Experience blog at

```
http://fxexperience.com/
```

Summary

Section 22.2 Controlling Fonts with Cascading Style Sheets (CSS)

- JavaFX objects can be formatted using Cascading Style Sheets (CSS).
- CSS allows you to specify presentation (e.g., fonts, spacing, sizes, colors, positioning) separately from the GUI's structure and content (layout containers, shapes, text, GUI components, etc.).

Section 22.2.1 CSS That Styles the GUI

- Each CSS rule begins with a CSS selector which specifies the JavaFX objects that will be styled according to the rule.
- A rule with a style class selector applies to any object that has a `styleClass` property with the class name. In CSS, a style class selector begins with a dot (`.`) and is followed by its class name.
- Each CSS rule's body is delimited by a set of required braces (`{}`) containing the CSS properties that are applied to objects matching the CSS selector.
- Each JavaFX CSS property name begins with `-fx-` followed by the name of the corresponding JavaFX object's property in all lowercase letters.
- The `-fx-spacing` property specifies vertical space between objects.
- The `-fx-padding` property separates an object from its container's edges.
- Selectors that begin with # are known as ID selectors—they are applied to objects with the specified ID.
- The `-fx-font` property can specify all aspects of a font, including its style, weight, size and font family—the size and font family are required.
- Font properties also may be specified with `-fx-font-style`, `-fx-font-weight`, `-fx-font-size` and `-fx-font-family`. These are applied to a JavaFX object's similarly named properties.

Section 22.2.2 FXML That Defines the GUI—Introduction to XML Markup

- Each FXML document begins with an XML declaration, which must be the first line in the file and indicates that the document contains XML markup.
- Each XML attribute has a *name* and *value* separated by `=`, and the *value* is placed in quotation marks (`""`). Multiple $name = value$ pairs are separated by whitespace.
- XML comments begin with `<!--` and end with `-->` and can span multiple lines.
- An FXML `import` declaration specifies the fully qualified name of a JavaFX type used in the document. Such declarations are delimited by `<?import` and `?>`.
- XML documents contain elements that specify the document's structure. Most elements are delimited by a start tag and an end tag.
- A start tag consists of angle brackets (`<` and `>`) containing the element's name followed by zero or more attributes.
- An end tag consists of the element name preceded by a forward slash (`/`) in angle brackets.
- Every XML document must have exactly one root element that contains all the other elements.
- An FXML layout element's `children` element contains the child `Node`s arranged by that layout.
- Empty elements use the shorthand start-tag-only notation in which the empty element's start tag ends with `/>`, rather than `>`.

- An XML namespace specifies a collection of element and attribute names that you can use in the document.

Section 22.2.3 Referencing the CSS File from FXML

- If you reference a CSS file from FXML, Scene Builder can apply the CSS rules to the GUI.
- The `@` symbol—called the local resolution operator in FXML—indicates that a file is located relative to the FXML file on disk.

Section 22.2.4 Specifying the VBox’s Style Class

- To apply a CSS class to an object in Scene Builder, set the object’s **Style Class** to the CSS class name without the dot (`.`).

Section 22.2.5 Programmatically Loading CSS

- It's possible to load CSS files dynamically and add them to a `Scene`'s collection of style sheets, which you can access via the `getStyleSheets` method.

Section 22.3 Displaying Two-Dimensional Shapes

- You can add objects of subclasses of `Shape` and `Shape3D` (package `javafx.scene.shape`) to a container in the JavaFX stage.
- You can add a `Canvas` object (package `javafx.scene.canvas`) to a container in the JavaFX stage, then draw on it using various `Canvas` methods.
- Like other `Node` types, you can drag shapes from the Scene Builder **Library**'s **Shapes** category onto the design area, then configure them via the **Properties**, **Layout** and **Code Inspector**s. You also may create objects of any JavaFX `Node` type programmatically.

Section 22.3.1 Defining Two-Dimensional Shapes with FXML

- For each property you can set in Scene Builder, there is a corresponding attribute in FXML.
- As you drag each shape onto your design, Scene Builder automatically configures certain properties, such as the **Fill** and **Stroke** colors for `Rectangle`s, `Circle`s, `Ellipse`s and `Arc`s.
- You can remove an attribute either by setting the property to its default value in Scene Builder or by manually editing the FXML.
- The default `fill` for a shape is black.
- The default stroke is a one-pixel black line.
- The default `strokeType` is centered, based on the stroke's thickness—half the thickness appears inside the shape's bounds and half outside. You also may display a shape's stroke completely inside or outside the shape's bounds.
- A `Line` connects two endpoints specified by the properties `startX`, `startY`, `endX` and `endY`.
- The *x*- and *y*-coordinate values are measured by default from the top-left corner of the layout, with *x*-coordinates increasing left to right and *y*-coordinates increasing top to bottom.
- If you specify a `Line`'s `layoutX` and `layoutY` properties, then the `startX`, `startY`, `endX` and `endY` properties are measured from that point.
- A `Rectangle` is displayed based on its `layoutX`, `layoutY`, `width` and `height` properties. The upper-left corner is positioned at the coordinates specified by the `layoutX` and `layoutY` properties.

- A `Circle` object is centered at the point specified by the `centerX` and `centerY` properties. The `radius` property determines the `Circle`'s size around its center point.
- An `Ellipse`'s center is specified by the `centerX` and `centerY` properties. Its `radiusX` and `radiusY` properties determine the `Ellipse`'s width and height.
- An `Arcs`'s center is specified by the `centerX` and `centerY` properties, and the properties `radiusX` and `radiusY` determine the `Arc`'s width and height. You must also specify the `Arc`'s `length`, `startAngle` and `type`.

Section 22.3.2 CSS That Styles the Two-Dimensional Shapes

- You specify a CSS type selector by using the JavaFX class name.
- When JavaFX renders an object, it combines all the CSS rules that apply to the object to determine its appearance.
- Colors may be specified as named colors (such as `"red"`, `"green"` and `"blue"`), RGBA colors, colors defined by their hue, saturation, brightness and alpha components and more.
- CSS function `rgba` defines a color based on its red, green, blue and alpha components.
- The `-fx-stroke-line-cap` CSS property indicates how lines should be terminated.
- The `-fx-fill` CSS property specifies the color or pattern that appears inside a shape.
- A `Rectangle`’s `-fx-arc-width` and `-fx-arc-height` properties specify the width and height of an ellipse that’s divided in half horizontally and vertically, then used to round the `Rectangle`’s corners.
- A gradient defines colors that transition gradually from one color to the next.
- CSS function `radial-gradient` produces color changes gradually from a center point outward.
- To specify an image as fill, you use the CSS function `image-pattern`.
- A linear gradient gradually transitions between colors horizontally, vertically or diagonally.

Section 22.4 `Polylines`, `Polygons` and `Paths`

- A `Polyline` draws a series of connected lines defined by a set of points.
- A `Polygon` draws a series of connected lines defined by a set of points and connects the last point to the first point.
- A `Path` draws a series of connected `PathElement`s by moving to a given point, then drawing lines, arcs and curves.

Section 22.4.2 `PolyshapesController` Class

- A `Path` is represented by a collection of `PathElement`s.
- The `MoveTo` subclass of `PathElement` moves to a specific position without drawing anything.
- The `ArcTo` subclass of `PathElement` draws an arc from the previous `PathElement`'s endpoint to the specified location.
- The `ClosePath` subclass of `PathElement` closes the path by drawing a straight line from the end point of the last `PathElement` to the start point of the first `PathElement`.
- An `ArcTo`'s `sweepFlag` determines whether the arc sweeps in the positive angle direction (`true`; counter clockwise) or the negative angle direction (`false`; clockwise).
- By default an `ArcTo` element is drawn as the shortest arc between the last `PathElement`'s end point and the point specified by the `ArcTo` element. To sweep the arc the long way around the ellipse, set the `ArcTo`'s `largeArcFlag` to `true`.

Section 22.5 Transforms

- A transform can be applied to any UI element to reposition or reorient the graphic.
- A `Translate` transform moves an object to a new location.
- A `Rotate` transform rotates an object around a point and by a specified rotation angle.
- A `Scale` transform scales an object's size by the specified amounts.
- To create a `Rotate` transform, invoke class `Transform`'s `static` method `rotate`, which returns a `Rotate` object. The method's first argument is the rotation angle. The method's next two arguments are the *x*- and *y*-coordinates of the point of rotation around which the `Shape` rotates.

Section 22.6 Playing Video with `Media`, `MediaPlayer` and `MediaViewer`

- JavaFX's audio and video capabilities are located in package `javafx.scene.media`.
- For simple audio playback you can use class `AudioClip`. For audio playback with more playback controls and for video playback you can use classes `Media`, `MediaPlayer` and `MediaView`.
- For video, JavaFX supports MPEG-4 (also called MP4) and Flash Video formats.

Section 22.6.1 VideoPlayer GUI

- `MediaView` is located in the Scene Builder **Library**'s **Controls** section.
- `ToolBar` is located in the Scene Builder **Library**'s **Containers** section. By default, Scene Builder adds one `Button` to a `ToolBar` when you drag the `ToolBar` onto your layout.

Section 22.6.2 `VideoPlayerController` Class

- A `Media` object specifies the location of the media to play and provides access to various information about the media, such as its duration, dimensions and more.
- A `MediaPlayer` object loads a `Media` object and controls playback. In addition, a `MediaPlayer` transitions through its various states during media loading and playback. You can provide `Runnable`s that execute in response to state transitions.
- A `MediaView` object displays the `Media` being played by a given `MediaPlayer` object.
- To load a video and prepare it for playback, you must associate it with a `MediaPlayer` object.
- Playing multiple videos requires a separate `MediaPlayer` for each `Media` object. However, a given `Media` object can be associated with multiple `MediaPlayer`s.
- A `MediaPlayer` does not provide a view in which to display video. For this purpose, you must associate a `MediaPlayer` with a `MediaView`. When a `MediaView` already exists—such as when it's created in FXML—you call the `MediaView`'s `setMediaPlayer` method to perform this task.
- When creating a `MediaView` object programmatically, you can pass a `MediaPlayer` to the `MediaView`'s constructor.
- A `MediaView` is like any other `Node` in the scene graph, so you can apply CSS styles, transforms and animations to it.
- Some common `MediaPlayer` states include *ready*, *playing* and *paused*.

- To perform a task for a given state, you specify an object that implements the `Runnable` interface (package `java.lang`). This interface contains a no-parameter `run` method that returns `void`.
- `MediaPlayer`’s `setOnEndOfMedia` method receives a `Runnable` object that should execute when video playback completes.
- `MediaPlayer` method `seek` moves to a specified time in the media clip.
- `MediaPlayer`’s `setOnError` method receives a `Runnable` object that should execute if the `MediaPlayer` enters the *error* state, indicating that an error occurred during playback.
- `MediaPlayer`’s `setOnReady` method receives a `Runnable` object that should execute if the `MediaPlayer` enters the *ready* state.
- A `Node`’s `sceneProperty` returns a `ReadOnlyObjectProperty<Scene>` that you can use to access to the `Scene` in which the `Node` is displayed.
- To bind to a specific properties of an object, you can use the methods of class `Bindings` (package `javafx.beans.binding`) to select the corresponding properties.
- Bindings method `selectDouble` gets a reference to a `DoubleProperty`.

Section 22.7 `Transition` Animations

- Transition animations change a `Node`'s property values from one value to another in a specified amount of time. Most properties of a `Node` can be animated.
- By default, the subclasses that define `Transition` animations (package `javafx.animations`) change the values of specific `Node` properties.

Section 22.7.2 `TransitionAnimationsController` Class

- A `FillTransition` changes a shape's fill color.
- A `StrokeTransition` changes a shape's stroke color.
- A `ParallelTransition` performs multiple transitions at the same time (that is, in parallel).
- A `FadeTransition` changes a `Node`'s opacity.
- A `RotateTransition` rotates a `Node`.
- Each `Transition` animation uses an `Interpolator` to calculate new property values throughout the animation's duration.
- The `Interpolator EASE_BOTH` begins the animation slowly at first (known as "easing in"), speeds up the in the middle, then slows again to complete the animation (known as "easing out").
- A `PathTransition` changes a shape's position by moving it along a `Path`.
- The `LineTo` subclass of `PathElement` draws a straight line from the previous `PathElement`'s endpoint to the specified location.
- The `Interpolator EASE_IN` begins an animation slowly at first, then continues at full speed.
- A `ScaleTransition` changes a `Node`'s size.
- The `Interpolator EASE_OUT` begins an animation at full speed, then slows down as the animation completes.
- A `SequentialTransition` performs a sequence of transitions—as each completes, the next one in the sequence begins executing.

- Every `Transition` has a `play` method that begins the animation.

Section 22.8 `Timeline` Animations

- A `Timeline` animation can change any `Node` property that's modifiable. You specify how to change property values with one or more `KeyFrame` objects that the `Timeline` animation performs in sequence.
- Each `KeyFrame` issues an `ActionEvent` at a particular time in the animation. The app can respond to the event by changing a `Node`'s property values.
- Setting an animation's cycle count to `Timeline.INDEFINITE` performs an animation until its `stop` method is called or the program terminates.

Section 22.9 Frame-By-Frame Animation with `AnimationTimer`

- An `AnimationTimer` (package `javafx.animation`) enables you to define frame-by-frame animations. You specify how your objects should move in a given frame, then JavaFX aggregates all of the drawing operations and displays the frame.
- JavaFX calls the `handle` method of every `AnimationTimer` before it draws an animation frame.
- For smooth animation, JavaFX tries to display animation frames at 60 frames per second.
- Method `handle` receives a time stamp in nanoseconds (billionths of a second) that you can use to determine the elapsed time since the last animation frame, then you can scale the movements of your objects accordingly. This enables you to define animations that operate at the same overall speed, regardless of the frame rate on a given device.
- Class `AnimationTimer` is an `abstract` class, so you must create a subclass to use it.

Section 22.10 Drawing on a Canvas

- Package `javafx.scene.canvas` contains two classes: `Canvas` is a subclass of `Node` in which you can draw graphics, and `GraphicsContext` performs the drawing operations on a `Canvas`.
- To draw on a `Canvas`, you must first obtain its `GraphicsContext` by calling `Canvas` method `getGraphicsContext2D`.
- When you set a `GraphicsContext`'s drawing characteristics, they're applied (as appropriate) to all subsequent shapes you draw. For example, if you call `setLineWidth` to specify the `GraphicsContext`'s line thickness, all subsequent `GraphicsContext` method calls that draw lines or shape borders will use this setting.
- `GraphicsContext`'s `setStroke` method specifies the `Paint` object (package `javafx.scene.paint`) used to draw the line. The `Paint` can be any of the subclasses `Color`, `ImagePattern`, `LinearGradient` or `RadialGradient`.
- `GraphicsContext`'s `strokeLine` method draws a line using the current `Paint` object that's set as the stroke. The four arguments are the *x*-*y* coordinates of the start and end points, respectively.
- `GraphicsContext`'s `setLineCap` method sets line cap.
- `GraphicsContext`'s `setGlobalAlpha` method sets the alpha transparency of all subsequent shapes you draw.
- `GraphicsContext`'s `setFill` method specifies the `Paint` object used to fill a shape.
- `GraphicsContext`'s `strokeRoundRect` method draws a hollow rectangle with rounded corners.
- `GraphicsContext`'s `fillRoundRect` method has the same arguments as `strokeRoundRect`, but uses the current `Paint` object

set as the fill to draw a filled rectangle with rounded corners.

- `GraphicsContext`'s `strokeRect` method draws a hollow rectangle without rounded corners.
- `GraphicsContext`'s `fillRect` method draws a filled rectangle without rounded corners.
- A `RadialGradient` specifies a gradient that radiates outward from a focal point.
- `Stop` objects represent color stops in a gradient.
- `GraphicsContext`'s `fillOval` and `strokeOval` methods draw a filled and hollow oval, respectively.
- `ImagePattern` is a subclass of `Paint` that loads an `Image`, either from the local system or from a URL specified as a `String`.
- A `LinearGradient` specifies a gradient that transitions between colors along a straight line.
- `GraphicsContext`'s `fillArc` and `strokeArc` methods draw filled and hollow arcs, respectively.

Section 22.11 Three-Dimensional Shapes

- The three-dimensional shapes are subclasses of `Shape3D` from the package `javafx.scene.shape`. These include `Box`, `Cylinder` and `Sphere`.
- The shading you see for three-dimensional shapes in Scene Builder comes from the scene's default lighting. Package `javafx.scene`'s `AmbientLight` and `PointLight` classes can be used to add your own lighting effects.
- You can use camera objects to view a scene from different angles and distances.
- An object's depth is measured along the *z*-axis, which runs perpendicular to your screen—when you move objects along the *z*-axis they get bigger as they're brought toward you and smaller as they're moved away from you.
- Colors and images on three-dimensional shapes are created by applying materials to the shapes.
- JavaFX class `PhongMaterial` is used to define materials.
- The name "Phong" is a 3D graphics term—phong shading is a technique for applying color and shading to 3D surfaces.
- `PhongMaterial` method `setDiffuseColor` sets the color that's applied to a three-dimensional shape's surface(s). The scene's lighting effects determine the shades of the color that are applied to each visible surface.
- `PhongMaterial` method `setDiffuseMap` sets the `Image` that's applied to a three-dimensional shape's surface(s).
- `PhongMaterial` method `setSpecularColor` sets the color of a bright spot that makes a 3D shape appear shiny. `PhongMaterial` method `setSpecularPower` determines the intensity of that spot.