

JavaFX GUI: Part 2

Based on Chapter 13 of
Java How to Program, 11/e

Introduction

- ▶ In this chapter, you'll:
 - 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 `Rectangles` and `Circles` 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.

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*
 - Doing so often creates a design that looks pleasing when it first loads, but deteriorates when the app is resized or the content updates

Laying Out Nodes in a Scene Graph (cont.)

- ▶ 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.

Laying Out Nodes in a Scene Graph (cont.)

- ▶ 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

Laying Out Nodes in a Scene Graph (cont.)

- ▶ Figure 13.1 describes each of the JavaFX layout panes, including those presented in Chapter 12 (previous lecture)
- ▶ 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.

Fig. 13.1 | JavaFX layout panes. (Part I of 2.)

Layout	Description
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. (Part 2 of 2.)

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.

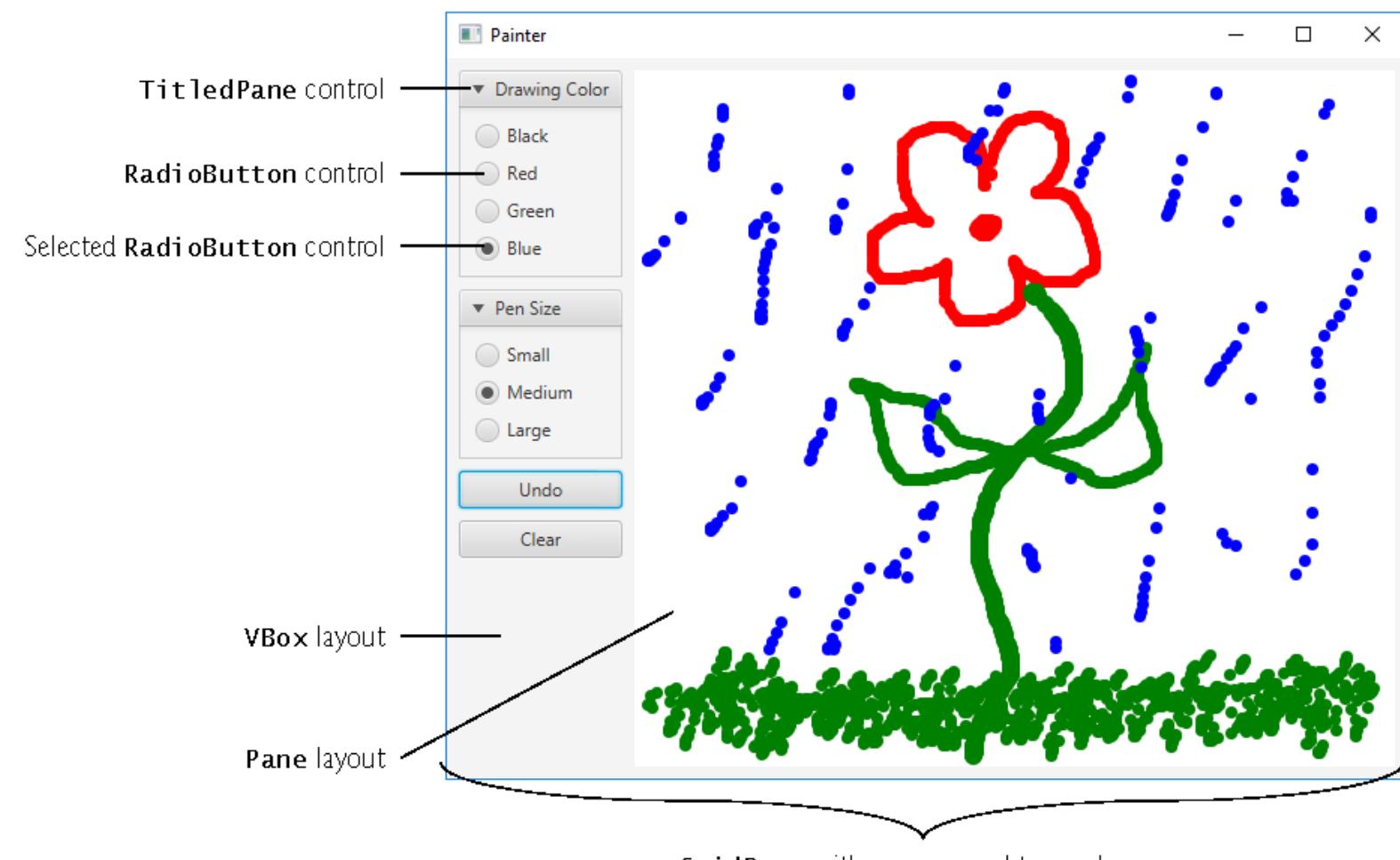


Fig. 13.2 | Painter app.

Technologies Overview

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
 - Can also create a ToggleGroup in Java, then use a RadioButton's **setToggleGroup** method to specify its ToggleGroup.

Technologies Overview (cont.)

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

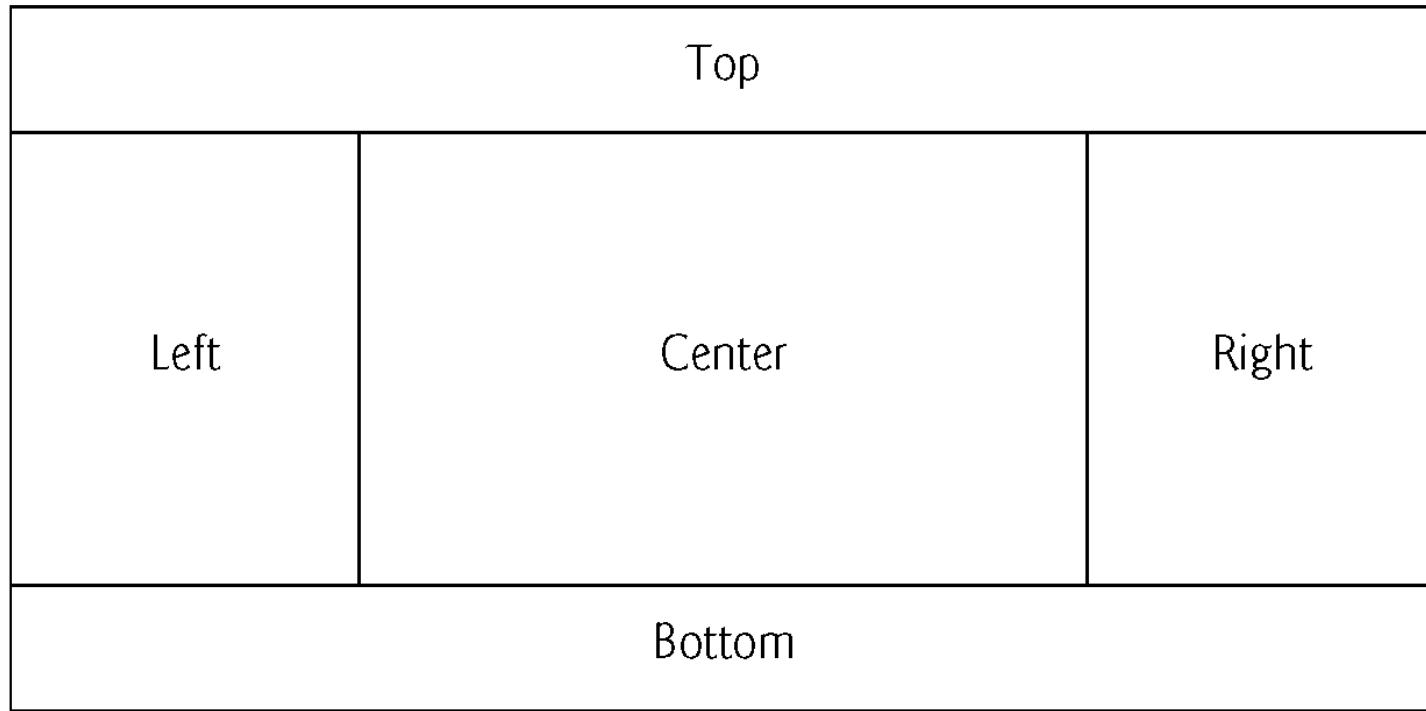


Fig. 13.3 | BorderPane's five areas.



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.

Technologies Overview (cont.)

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 (Fig. 13.4)
- ▶ You'll configure an **onMouseDragged** event handler for this app's **Pane**
- ▶ JavaFX supports other 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/scene/Node.html>

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 <i>different</i> node continues with the mouse cursor over the given node.

Fig. 13.4 | Mouse events. (Part I of 2.)

Mouse events	When the event occurs for a 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. (Part 2 of 2.)

Technologies Overview (cont.)

Setting a Control's User Data

- ▶ Each control has a `setUserData` method that receives an Object
- ▶ 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.

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

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

Technologies Overview (cont.)

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 TitledPanes to organize the app's Radio-Buttons and to help the user understand the purpose of each RadioButton group

Technologies Overview (cont.)

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
- ▶ 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.



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.

Building the GUI (cont.)

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

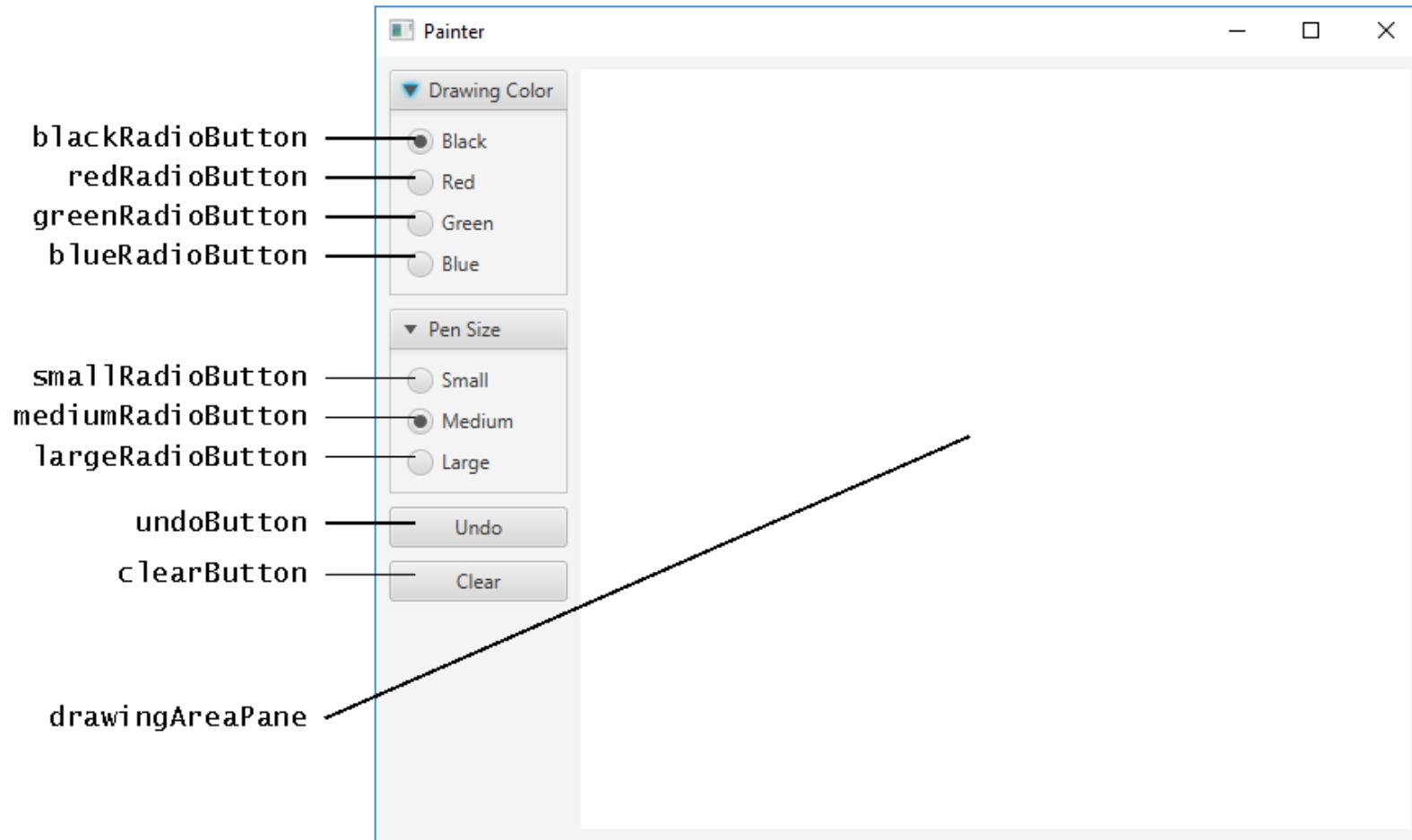
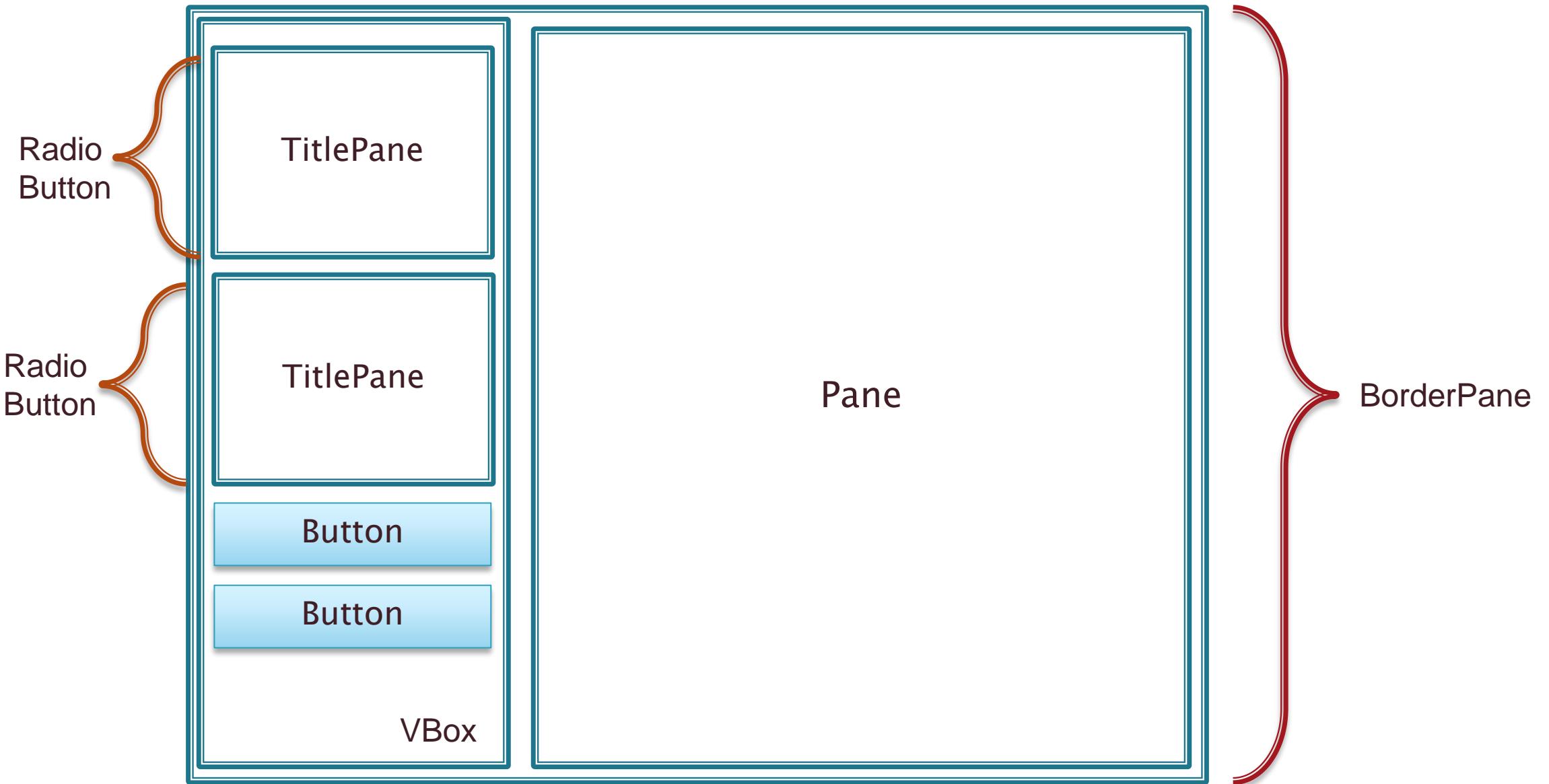


Fig. 13.5 | Painter GUI labeled with **fx:id**s for the programmatically manipulated controls.



Building the GUI (cont.)

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 BorderPane'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

Building the GUI (cont.)

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

Building the GUI (cont.)

Step 3: Adding the VBox and Pane (cont.)

- ▶ 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

Building the GUI (cont.)

Step 3: Adding the VBox and Pane (cont.)

- ▶ 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`.

Building the GUI (cont.)

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

Building the GUI (cont.)

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

Building the GUI (cont.)

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 Titled-Pane, then configure their Text properties and fx:ids as shown in Fig. 13.5
- ▶ Select the `blackRadioButton` and ensure that its `Selected` property is checked, then do the same for the `mediumRadioButton`.

Building the GUI (cont.)

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

Building the GUI (cont.)

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 Titled-Panes 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:ids as shown in Fig. 13.5
- ▶ Set each Button's Max Width property to `MAX_VALUE` so that they fill the VBox's width

Building the GUI (cont.)

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.

Building the GUI (cont.)

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:
 - Expand Scene Builder's Controller window (located below the Hierarchy window).
 - In the Controller Class field, type PainterController

Building the GUI (cont.)

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 mouse-dragged event.
 - For the four Drawing Color RadioButtons, specify `colorRadioButtonSelected` as each RadioButton's On Action event handler
 - This method will set the current drawing color, based on the user's selection.

Building the GUI (cont.)

Step 12: Specifying the Event-Handler Method Names (cont.)

- ▶ For the three Pen Size RadioButtons, 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.

Building the GUI (cont.)

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.

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

```
1 // Fig. 13.5: Painter.java
2 // Main application class that loads and displays the Painter's GUI.
3 import javafx.application.Application;
4 import javafx.fxml.FXMLLoader;
5 import javafx.scene.Parent;
6 import javafx.scene.Scene;
7 import javafx.stage.Stage;
8
9 public class Painter extends Application {
10     @Override
11     public void start(Stage stage) throws Exception {
12         Parent root =
13             FXMLLoader.load(getClass().getResource("Painter.fxml"));
```

Fig. 13.6 | Main application class that loads and displays the **Painter**'s GUI.

```
14  
15     Scene scene = new Scene(root);  
16     stage.setTitle("Painter"); // displayed in window's title bar  
17     stage.setScene(scene);  
18     stage.show();  
19 }  
20  
21 public static void main(String[] args) {  
22     launch(args);  
23 }  
24 }
```

Fig. 13.6 | Main application class that loads and displays the **Painter**'s GUI.

PainterController Class

- ▶ Figure 13.7 shows the final version of class PainterController with this app's new features highlighted

```
1 // Fig. 13.6: PainterController.java
2 // Controller for the Painter app
3 import javafx.event.ActionEvent;
4 import javafx.fxml.FXML;
5 import javafx.scene.control.RadioButton;
6 import javafx.scene.control.ToggleGroup;
7 import javafx.scene.input.MouseEvent;
8 import javafx.scene.layout.Pane;
9 import javafx.scene.paint.Color;
10 import javafx.scene.paint.Paint;
11 import javafx.scene.shape.Circle;
12
```

Fig. 13.7 | Controller for the **Painter** app. (Part 1 of 7.)

```
I3  public class PainterController {  
I4      // enum representing pen sizes  
I5      private enum PenSize {  
I6          SMALL(2),  
I7          MEDIUM(4),  
I8          LARGE(6);  
I9  
I20     private final int radius;  
I21  
I22     PenSize(int radius) {this.radius = radius;} // constructor  
I23  
I24     public int getRadius() {return radius;}  
I25 };  
I26
```

Fig. 13.7 | Controller for the **Painter** app. (Part 2 of 7.)

```
27 // instance variables that refer to GUI components
28 @FXML private RadioButton blackRadioButton;
29 @FXML private RadioButton redRadioButton;
30 @FXML private RadioButton greenRadioButton;
31 @FXML private RadioButton blueRadioButton;
32 @FXML private RadioButton smallRadioButton;
33 @FXML private RadioButton mediumRadioButton;
34 @FXML private RadioButton largeRadioButton;
35 @FXML private Pane drawingAreaPane;
36 @FXML private ToggleGroup colorToggleGroup;
37 @FXML private ToggleGroup sizeToggleGroup;
38
39 // instance variables for managing Painter state
40 private PenSize radius = PenSize.MEDIUM; // radius of circle
41 private Paint brushColor = Color.BLACK; // drawing color
42
```

Fig. 13.7 | Controller for the **Painter** app. (Part 3 of 7.)

```
43     // set user data for the RadioButtons
44     public void initialize() {
45         // user data on a control can be any Object
46         blackRadioButton.setUserData(Color.BLACK);
47         redRadioButton.setUserData(Color.RED);
48         greenRadioButton.setUserData(Color.GREEN);
49         blueRadioButton.setUserData(Color.BLUE);
50         smallRadioButton.setUserData(PenSize.SMALL);
51         mediumRadioButton.setUserData(PenSize.MEDIUM);
52         largeRadioButton.setUserData(PenSize.LARGE);
53     }
54 }
```

Fig. 13.7 | Controller for the **Painter** app. (Part 4 of 7.)

```
55 // handles drawingArea's onMouseDragged MouseEvent
56 @FXML
57 private void drawingAreaMouseDragged(MouseEvent e) {
58     Circle newCircle = new Circle(e.getX(), e.getY(),
59         radius.getRadius(), brushColor);
60     drawingAreaPane.getChildren().add(newCircle);
61 }
62
63 // handles color RadioButton's ActionEvents
64 @FXML
65 private void colorRadioButtonSelected(ActionEvent e) {
66     // user data for each color RadioButton is the corresponding Color
67     brushColor =
68         (Color) colorToggleGroup.getSelectedToggle().getUserData();
69 }
70
```

Fig. 13.7 | Controller for the **Painter** app. (Part 5 of 7.)

```
71 // handles size RadioButton's ActionEvents
72 @FXML
73 private void sizeRadioButtonSelected(ActionEvent e) {
74     // user data for each size RadioButton is the corresponding PenSize
75     radius =
76         (PenSize) sizeToggleGroup.getSelectedToggle().getUserData();
77 }
78
79 // handles Undo Button's ActionEvents
80 @FXML
81 private void undoButtonPressed(ActionEvent event) {
82     int count = drawingAreaPane.getChildren().size();
83
84     // if there are any shapes remove the last one added
85     if (count > 0) {
86         drawingAreaPane.getChildren().remove(count - 1);
87     }
88 }
```

Fig. 13.7 | Controller for the **Painter** app. (Part 6 of 7.)

```
89
90     // handles Clear Button's ActionEvents
91     @FXML
92     private void clearButtonPressed(ActionEvent event) {
93         drawingAreaPane.getChildren().clear(); // clear the canvas
94     }
95 }
```

Fig. 13.7 | Controller for the **Painter** app. (Part 7 of 7.)

PainterController Class

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 enums as **nested types** inside other classes
 - Except for the anonymous inner class introduced in Section 12.5.5, all the classes, interfaces and enums 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

PainterController Class (cont.)

Instance Variables

- ▶ Lines 28–37 declare the @FXML instance variables that the controller uses to programmatically interact with the GUI
- ▶ These variables must match the corresponding fx:id values that you specified in Painter.fxml
- ▶ Two of the @FXML instance variables are ToggleGroups—in the RadioButton event handlers
 - Used 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.

PainterController Class (cont.)

Method initialize

- ▶ 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

PainterController Class (cont.)

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`)
- ▶ 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

PainterController Class (cont.)

drawingAreaMouseDragged Event Handler (cont.)

- ▶ 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`

PainterController Class (cont.)

drawingAreaMouseDragged Event Handler (cont.)

- ▶ 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.

PainterController Class (cont.)

colorRadioButtonSelected Event Handler

- ▶ Lines 64–69 define `colorRadioButtonSelected`, which responds to the `ActionEvents` of the Drawing Color RadioButtons—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 RadioButtons
- ▶ Lines 67–68 set the current drawing Color

PainterController Class (cont.)

colorRadioButtonSelected Event Handler (cont.)

- ▶ 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 RadioButtons, this Object is always a Color, so we cast the Object to a Color and assign it to `brushColor`

PainterController Class (cont.)

sizeRadioButtonSelected Event Handler

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

PainterController Class (cont.)

undoButtonPressed Event Handler

- ▶ Lines 80–88 define `undoButtonPressed`, which responds to an `ActionEvent` from the `undo`-Button 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.

PainterController Class (cont.)

clearButtonPressed Event Handler

- ▶ Lines 91–94 define `clearButtonPressed`, which responds to the `ActionEvent` from the `clear`-Button 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.

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

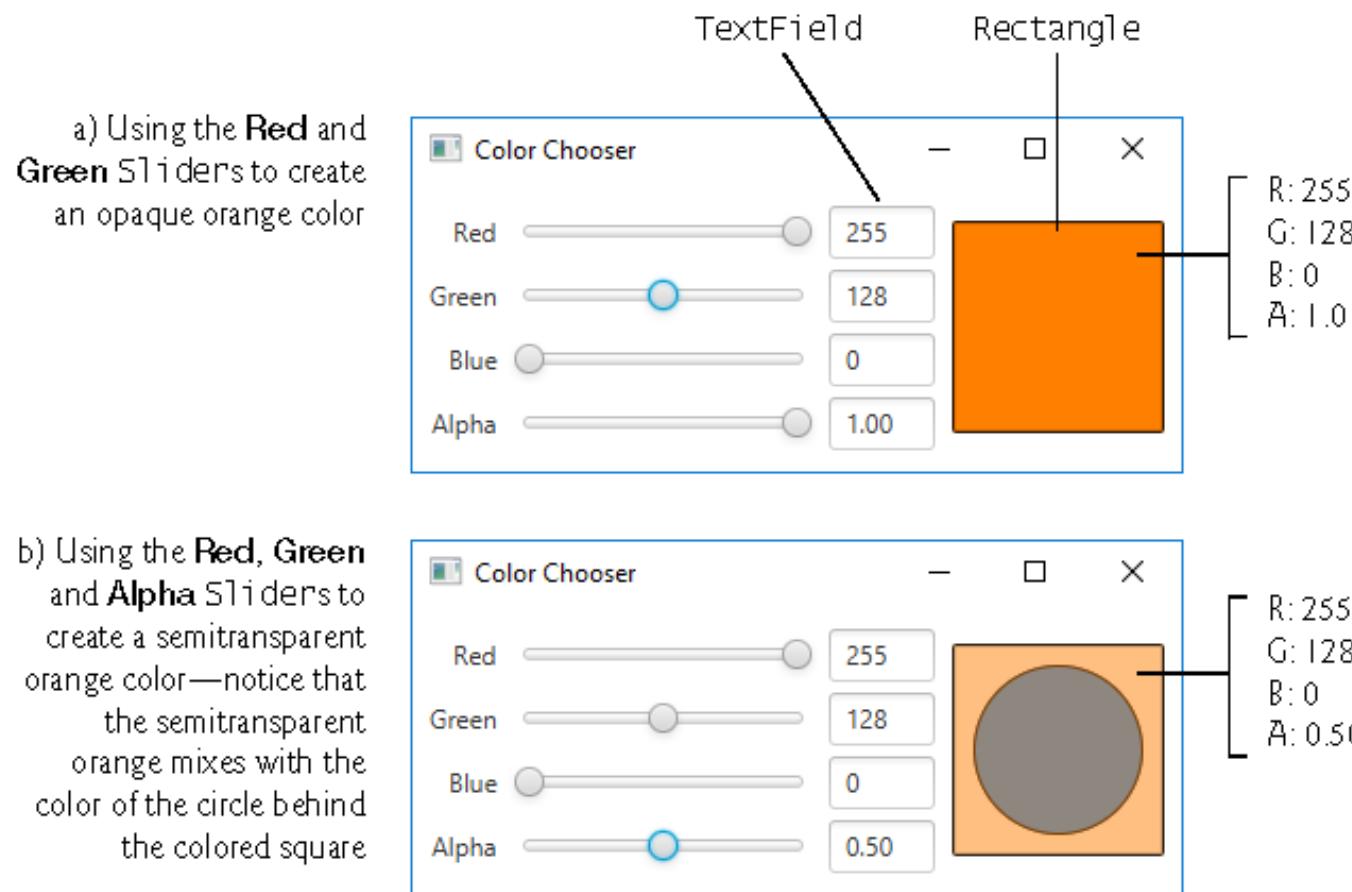


Fig. 13.8 | Color Chooser app with opaque and semitransparent orange colors.

Technologies Overview

RGBA Colors

- ▶ The app uses the **RGBA color system** to display a rectangle of color based on the values of four **Sliders**
- ▶ 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
- ▶ 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 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.

Technologies Overview (cont.)

Properties of a Class

- ▶ JavaFX makes extensive use of properties
- ▶ A **property** is defined by creating *set/get* methods named as follows

```
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
 - 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.

Technologies Overview (cont.)

Property Bindings

- ▶ JavaFX properties are *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**—enables a property of one object to be updated when a property of another changes
 - 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

Technologies Overview (cont.)

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 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 **Sliders** and set that **Color** as the fill color of a **Rectangle** object that displays the current color
- ▶ More on properties, property bindings and property listeners:
 - <http://docs.oracle.com/javase/8/javafx/properties-binding-tutorial/binding.htm>

Building the GUI

- ▶ In this section, we'll discuss the Color Chooser app's GUI
- ▶ 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`.

Building the GUI (cont.)

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

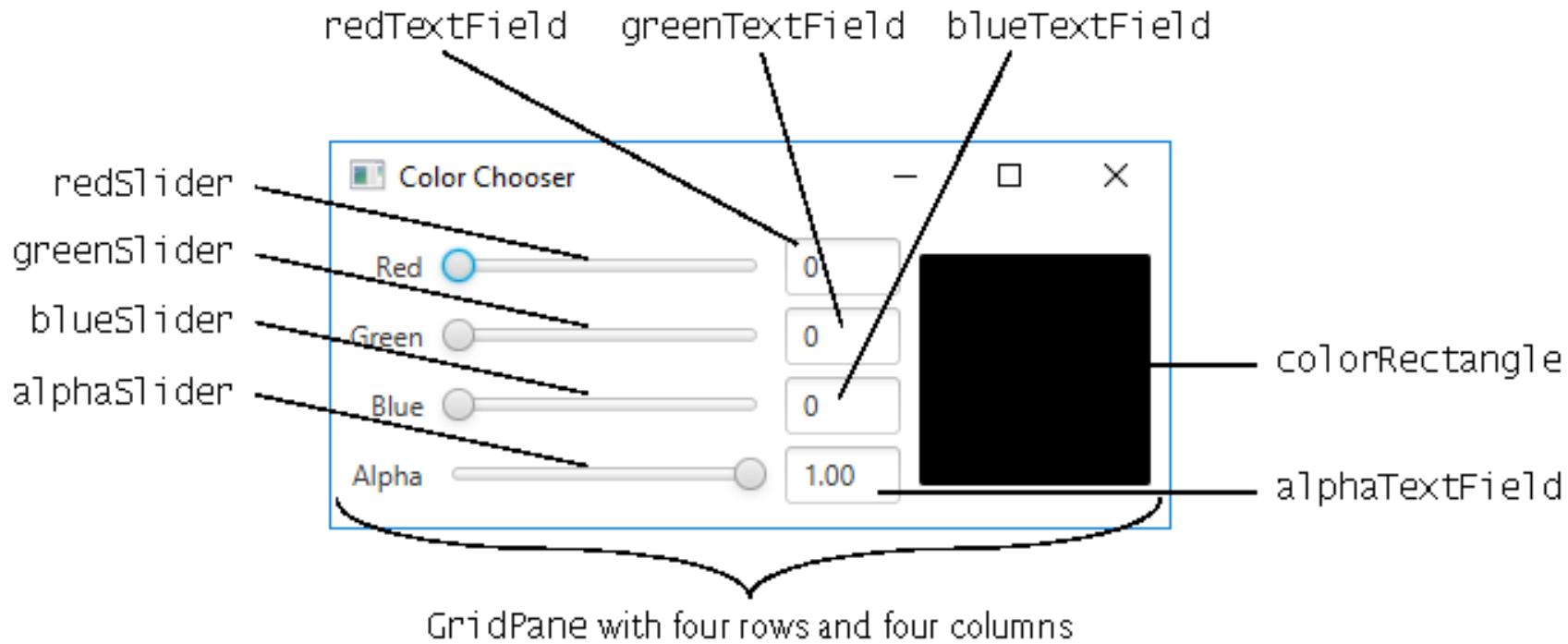


Fig. 13.9 | Color Chooser app's programmatically manipulated controls labeled with their **fx:id**s.

Building the GUI (cont.)

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

Building the GUI (cont.)

Step 3: Adding the Controls

- ▶ Using Fig. 13.9 as a guide, add the Labels, Sliders, TextFields, 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 Labels and TextFields as shown and set all the appropriate fx:id properties as you add each control.

Building the GUI (cont.)

Step 4: Configuring the Sliders

- ▶ For the red, green and blue **Sliders**, 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.

Building the GUI (cont.)

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.

Building the GUI (cont.)

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

Building the GUI (cont.)

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`

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

```
1 // Fig. 13.8: ColorChooser.java
2 // Main application class that loads and displays the ColorChooser's GUI.
3 import javafx.application.Application;
4 import javafx.fxml.FXMLLoader;
5 import javafx.scene.Parent;
6 import javafx.scene.Scene;
7 import javafx.stage.Stage;
8
9 public class ColorChooser extends Application {
10     @Override
11     public void start(Stage stage) throws Exception {
12         Parent root =
13             FXMLLoader.load(getClass().getResource("ColorChooser.fxml"));
```

Fig. 13.10 | Application class that loads and displays the **Color Chooser**'s GUI. (Part I of 2.)

```
14  
15     Scene scene = new Scene(root);  
16     stage.setTitle("Color Chooser");  
17     stage.setScene(scene);  
18     stage.show();  
19 }  
20  
21 public static void main(String[] args) {  
22     launch(args);  
23 }  
24 }
```

Fig. 13.10 | Application class that loads and displays the **Color Chooser**'s GUI. (Part 2 of 2.)

ColorChooserController Class

- ▶ Figure 13.11 shows the final version of class `ColorChooserController` with this app's new features highlighted

```
1 // Fig. 13.9: ColorChooserController.java
2 // Controller for the ColorChooser app
3 import javafx.beans.value.ChangeListener;
4 import javafx.beans.value.ObservableValue;
5 import javafx.fxml.FXML;
6 import javafx.scene.control.Slider;
7 import javafx.scene.control.TextField;
8 import javafx.scene.paint.Color;
9 import javafx.scene.shape.Rectangle;
10
```

Fig. 13.11 | Controller for the ColorChooser app. (Part I of 5.)

```
11 public class ColorChooserController {  
12     // instance variables for interacting with GUI components  
13     @FXML private Slider redSlider;  
14     @FXML private Slider greenSlider;  
15     @FXML private Slider blueSlider;  
16     @FXML private Slider alphaSlider;  
17     @FXML private TextField redTextField;  
18     @FXML private TextField greenTextField;  
19     @FXML private TextField blueTextField;  
20     @FXML private TextField alphaTextField;  
21     @FXML private Rectangle colorRectangle;  
22
```

Fig. 13.11 | Controller for the ColorChooser app. (Part 2 of 5.)

```
23 // instance variables for managing
24 private int red = 0;
25 private int green = 0;
26 private int blue = 0;
27 private double alpha = 1.0;
28
29 public void initialize() {
30     // bind TextField values to corresponding Slider values
31     redTextField.textProperty().bind(
32         redSlider.valueProperty().asString("%.0f"));
33     greenTextField.textProperty().bind(
34         greenSlider.valueProperty().asString("%.0f"));
35     blueTextField.textProperty().bind(
36         blueSlider.valueProperty().asString("%.0f"));
37     alphaTextField.textProperty().bind(
38         alphaSlider.valueProperty().asString("%.2f"));
39 }
```

Fig. 13.11 | Controller for the ColorChooser app. (Part 3 of 5.)

```
40     // listeners that set Rectangle's fill based on Slider changes
41     redSlider.valueProperty().addListener(
42         new ChangeListener<Number>() {
43             @Override
44             public void changed(ObservableValue<? extends Number> ov,
45                 Number oldValue, Number newValue) {
46                 red = newValue.intValue();
47                 colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
48             }
49         }
50     );
51     greenSlider.valueProperty().addListener(
52         new ChangeListener<Number>() {
53             @Override
54             public void changed(ObservableValue<? extends Number> ov,
55                 Number oldValue, Number newValue) {
56                 green = newValue.intValue();
57                 colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
58             }
59         }
60     );

```

Fig. 13.11 | Controller for the ColorChooser app. (Part 4 of 5.)

```
61     blueSlider.valueProperty().addListener(
62         new ChangeListener<Number>() {
63             @Override
64             public void changed(ObservableValue<? extends Number> ov,
65                 Number oldValue, Number newValue) {
66                 blue = newValue.intValue();
67                 colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
68             }
69         }
70     );
71     alphaSlider.valueProperty().addListener(
72         new ChangeListener<Number>() {
73             @Override
74             public void changed(ObservableValue<? extends Number> ov,
75                 Number oldValue, Number newValue) {
76                 alpha = newValue.doubleValue();
77                 colorRectangle.setFill(Color.rgb(red, green, blue, alpha));
78             }
79         }
80     );
81 }
82 }
```

Fig. 13.11 | Controller for the ColorChooser app. (Part 5 of 5.)

ColorChooserController Class (cont.)

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.

ColorChooserController Class (cont.)

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.

ColorChooserController Class (cont.)

Property-to-Property Bindings

- ▶ Lines 31–38 set up property bindings between a Slider's value and the corresponding Text-Field's text so that changing a Slider updates the corresponding TextField
- ▶ Consider lines 31–32, which bind the redSlider's valueProperty to the redText-Field's textProperty:

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

ColorChooserController Class (cont.)

Property-to-Property Bindings (cont.)

- ▶ 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")`

ColorChooserController Class (cont.)

Property-to-Property Bindings (cont.)

- ▶ 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

ColorChooserController Class (cont.)

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 Sliders' value properties
- ▶ Consider lines 41–50, which register the ChangeListener that executes when the user moves the redSlider's thumb
- ▶ 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

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`.

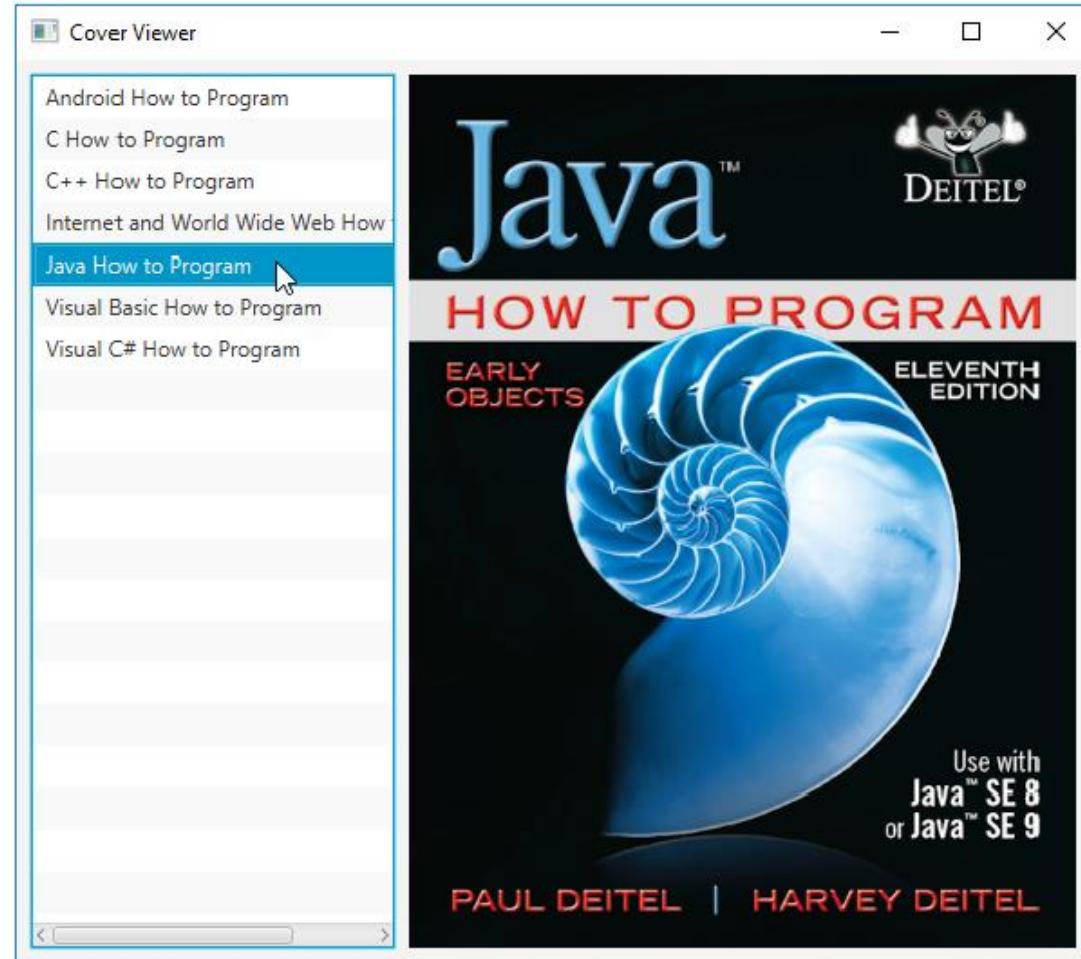


Fig. 13.12 | Cover Viewer with Java How to Program selected.

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

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

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

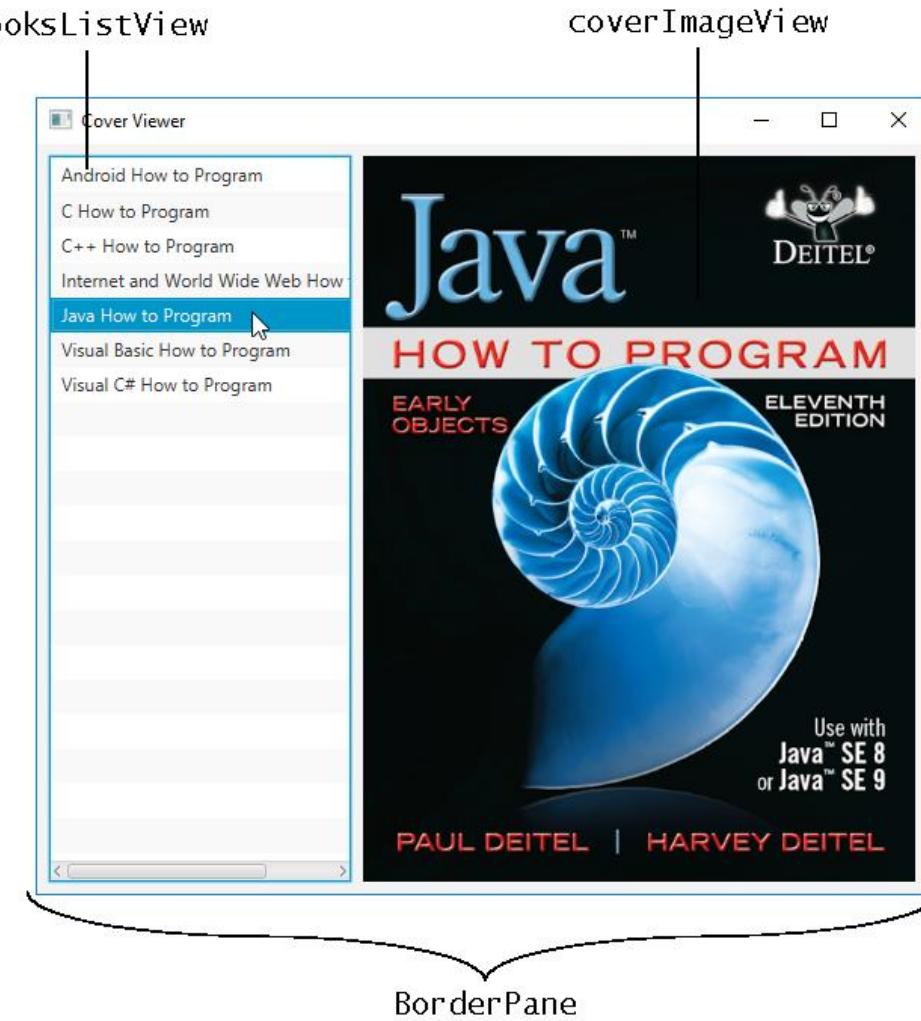


Fig. 13.13 | Cover Viewer app's programmatically manipulated controls labeled with their fx:ids.

Building the GUI (cont.)

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

Building the GUI (cont.)

Adding and Configuring the Controls (cont.)

- ▶ 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.

Building the GUI (cont.)

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`

CoverViewer Subclass of Application

- ▶ Figure 13.14 shows class CoverViewer subclass of Application

```
1 // Fig. 13.13: CoverViewer.java
2 // Main application class that loads and displays the CoverViewer's GUI.
3 import javafx.application.Application;
4 import javafx.fxml.FXMLLoader;
5 import javafx.scene.Parent;
6 import javafx.scene.Scene;
7 import javafx.stage.Stage;
8
9 public class CoverViewer extends Application {
10     @Override
11     public void start(Stage stage) throws Exception {
12         Parent root =
13             FXMLLoader.load(getClass().getResource("CoverViewer.fxml"));
14
15         Scene scene = new Scene(root);
16         stage.setTitle("Cover Viewer");
17         stage.setScene(scene);
18         stage.show();
19     }
20
21     public static void main(String[] args) {
22         launch(args);
23     }
24 }
```

Fig. 13.14 | Main application class that loads and displays the **Cover Viewer**'s GUI.

CoverViewerController Class

- ▶ Figure 13.15 shows the final version of class CoverViewerController with the app's new features highlighted

```
1 // Fig. 13.14: CoverViewerController.java
2 // Controller for Cover Viewer application
3 import javafx.beans.value.ChangeListener;
4 import javafx.beans.value.ObservableValue;
5 import javafx.collections.FXCollections;
6 import javafx.collections.ObservableList;
7 import javafx.fxml.FXML;
8 import javafx.scene.control.ListView;
9 import javafx.scene.image.Image;
10 import javafx.scene.image.ImageView;
11
```

Fig. 13.15 | Controller for **Cover Viewer** application. (Part I of 4.)

```
I2  public class CoverViewerController {  
I3      // instance variables for interacting with GUI  
I4      @FXML private ListView<Book> booksListView;  
I5      @FXML private ImageView coverImageView;  
I6  
I7      // stores the list of Book Objects  
I8      private final ObservableList<Book> books =  
I9          FXCollections.observableArrayList();  
I20
```

Fig. 13.15 | Controller for **Cover Viewer** application. (Part 2 of 4.)

```
21 // initialize controller
22 public void initialize() {
23     // populate the ObservableList<Book>
24     books.add(new Book("Android How to Program",
25                         "/images/small/androidhttp.jpg", "/images/large/androidhttp.jpg"));
26     books.add(new Book("C How to Program",
27                         "/images/small/chtp.jpg", "/images/large/chtp.jpg"));
28     books.add(new Book("C++ How to Program",
29                         "/images/small/cpphttp.jpg", "/images/large/cpphttp.jpg"));
30     books.add(new Book("Internet and World Wide Web How to Program",
31                         "/images/small/iw3http.jpg", "/images/large/iw3http.jpg"));
32     books.add(new Book("Java How to Program",
33                         "/images/small/jhttp.jpg", "/images/large/jhttp.jpg"));
34     books.add(new Book("Visual Basic How to Program",
35                         "/images/small/vbhttp.jpg", "/images/large/vbhttp.jpg"));
36     books.add(new Book("Visual C# How to Program",
37                         "/images/small/vcshttp.jpg", "/images/large/vcshttp.jpg"));
38     booksListView.setItems(books); // bind booksListView to books
```

Fig. 13.15 | Controller for **Cover Viewer** application. (Part 3 of 4.)

```
39
40     // when ListView selection changes, show large cover in ImageView
41     booksListView.getSelectionModel().selectedItemProperty().
42         addListener(
43             new ChangeListener<Book>() {
44                 @Override
45                 public void changed(ObservableValue<? extends Book> ov,
46                     Book oldValue, Book newValue) {
47                     coverImageView.setImage(
48                         new Image(newValue.getLargeImage()));
49                 }
50             }
51         );
52     }
53 }
```

Fig. 13.15 | Controller for **Cover Viewer** application. (Part 4 of 4.)

CoverViewerController Class (cont.)

@FXML Instance Variables

- ▶ Lines 14–15 declare the controller’s **@FXML** instance variables
- ▶ In this case, the **ListView** displays **Book** objects
- ▶ Class **Book** contains three **String** instance variables with *set/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
- ▶ Copy **Book.java** from this chapter’s examples folder into the folder that contains **CoverViewer.fxml**, **CoverViewer.java** and **CoverViewerController.java**

CoverViewerController Class (cont.)

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 `Observable-List` interface

CoverViewerController Class (cont.)

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

CoverViewerController Class (cont.)

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
 - `ListView`s also support multiple selection
- ▶ Selection is managed by the `ListView`'s `MultipleSelectionModel` (a subclass of `SelectionModel` from package `javafx.scene.control`)
 - 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)

CoverViewerController Class (cont.)

Listening for ListView Selection Changes (cont.)

- ▶ ListView method `getSelectionModel` returns a `MultipleSelectionModel` object
- ▶ In this example, `MultipleSelectionModel`'s `selectedItemProperty` method returns a `ReadOnlyObjectProperty<Book>`
- ▶ Corresponding `ChangeListener` receives as `oldValue` and `newValue` 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`

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)

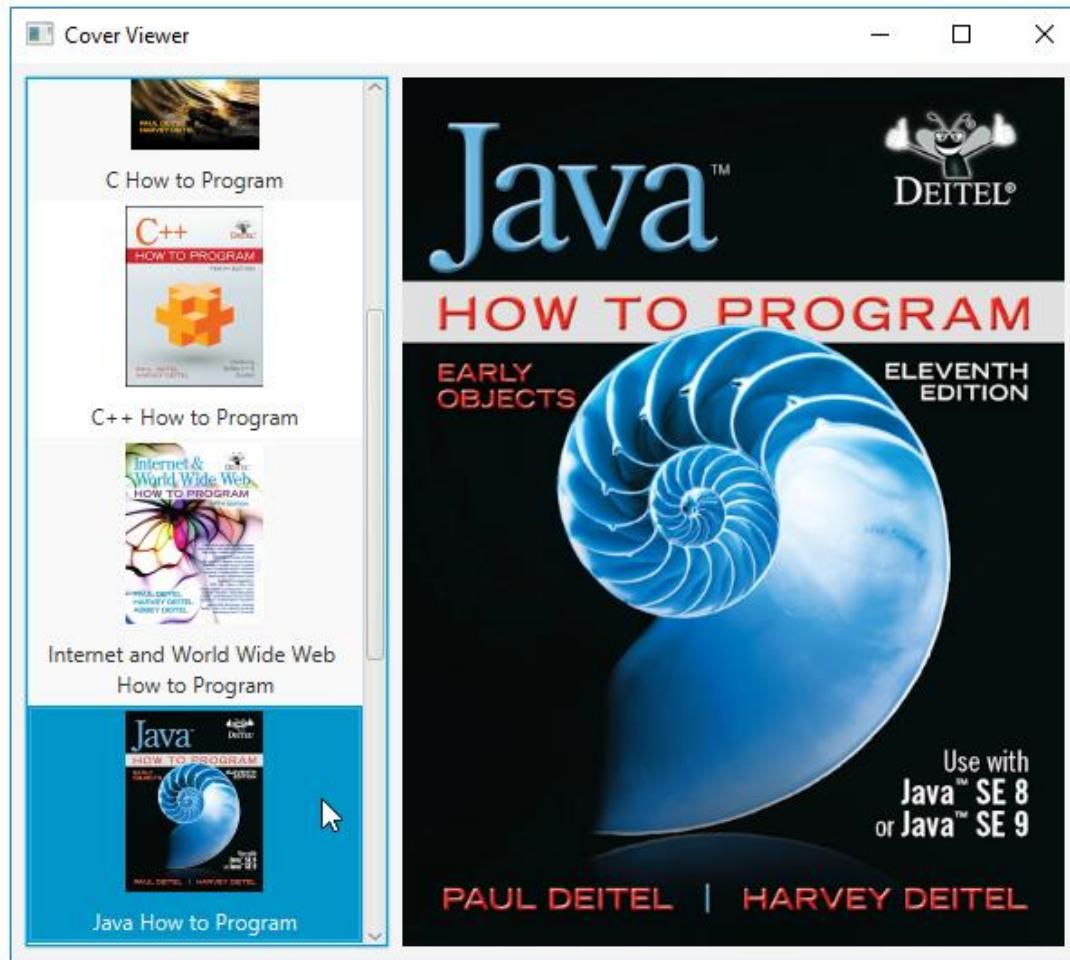


Fig. 13.16 | **Cover Viewer** app with *Java How to Program* selected.

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 ListCells 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

Technologies Overview (cont.)

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

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`

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

```
1 // Fig. 13.16: ImageTextCell.java
2 // Custom ListView cell factory that displays an Image and text
3 import javafx.geometry.Pos;
4 import javafx.scene.control.Label;
5 import javafx.scene.control.ListCell;
6 import javafx.scene.image.Image;
7 import javafx.scene.image.ImageView;
8 import javafx.scene.layout.VBox;
9 import javafx.scene.text.TextAlignment;
10
11 public class ImageTextCell extends ListCell<Book> {
12     private VBox vbox = new VBox(8.0); // 8 points of gap between controls
13     private ImageView thumbImageView = new ImageView(); // initially empty
14     private Label label = new Label();
15 }
```

Fig. 13.17 | Custom ListView cell factory that displays an image and text.

```
16 // constructor configures VBox, ImageView and Label
17 public ImageTextCell() {
18     vbox.setAlignment(Pos.CENTER); // center VBox contents horizontally
19
20     thumbImageView.setPreserveRatio(true);
21     thumbImageView.setFitHeight(100.0); // thumbnail 100 points tall
22     vbox.getChildren().add(thumbImageView); // attach to Vbox
23
24     label.setWrapText(true); // wrap if text too wide to fit in label
25     label.setAlignment(TextAlignment.CENTER); // center text
26     vbox.getChildren().add(label); // attach to VBox
27
28     setPrefWidth(USE_PREF_SIZE); // use preferred size for cell width
29 }
30
```

Fig. 13.17 | Custom ListView cell factory that displays an image and text.

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

Fig. 13.17 | Custom ListView cell factory that displays an image and text.

ImageTextCell Custom Cell Factory Class (cont.)

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

ImageTextCell Custom Cell Factory Class (cont.)

Constructor (cont.)

- ▶ 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.

ImageTextCell Custom Cell Factory Class (cont.)

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

ImageTextCell Custom Cell Factory Class (cont.)

Method updateItem

- ▶ 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

ImageTextCell Custom Cell Factory Class (cont.)

Method updateItem

- ▶ 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`.

CoverViewerController Class

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

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

CoverViewerController Class (cont.)

- ▶ 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 `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

CoverViewerController Class (cont.)

- ▶ 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.

13.7 Additional JavaFX Capabilities (cont.)

Accessibility

- ▶ 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

13.7 Additional JavaFX Capabilities (cont.)

Accessibility (cont.)

- ▶ 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.

13.7 Additional JavaFX Capabilities (cont.)

Accessibility (cont.)

- ▶ Every JavaFX Node subclass also has these accessibility-related properties:
 - **accessibleTextProperty**—A String that a screen reader speaks for a control
 - **accessibleHelpProperty**—A more detailed control description String than that provided by the **accessibleTextProperty**
 - 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`)
 - Screen reader uses this to determine attributes and actions supported for a control
 - **accessibleRoleDescriptionProperty**—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**

13.7 Additional JavaFX Capabilities (cont.)

Accessibility (cont.)

- ▶ In addition, you can add **Labels** 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

13.7 Additional JavaFX Capabilities (cont.)

Third-Party JavaFX Libraries

- ▶ Various libraries for additional JavaFX capabilities
 - ControlsFX (<http://www.controlsfx.org/>) provides common dialogs, additional controls, validation capabilities, `TextField` enhancements, a `SpreadSheetView`, `TableView` enhancements and more
 - JFXtras (<http://jfxtras.org/>) 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>

13.7 Additional JavaFX Capabilities (cont.)

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

- ▶ Oracle officially supports JavaFX only for desktop apps
- ▶ Gluon's open-source JavaFXPorts project (<http://javafxports.org/>) brings desktop JavaFX to mobile devices (iOS and Android) and devices like the Raspberry Pi (<https://www.raspberrypi.org/>)
- ▶ In addition, Gluon Mobile provides a mobile-optimized JavaFX implementation for iOS and Android
 - <http://gluonhq.com/products/mobile/>

13.7 Additional JavaFX Capabilities (cont.)

Scenic View for Debugging JavaFX Scenes and Nodes

- ▶ Scenic View is a debugging tool for JavaFX scenes and nodes
- ▶ Embed Scenic View directly into apps or run it as a standalone app
- ▶ 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 more information, visit
 - <http://www.scenic-view.org>

13.7 Additional JavaFX Capabilities (cont.)

JavaFX Resources and JavaFX in the Real World

- ▶ Visit the following website for a lengthy and growing list of JavaFX resources
 - <http://bit.ly/JavaFXResources>

13.8 JavaFX 9: Java SE 9 JavaFX Updates

Java SE 9 Modularization

- ▶ Java SE 9's biggest new software-engineering feature is the module system
- ▶ The key JavaFX 9 modules are:
 - `javafx.base`—Required by all JavaFX 9 apps
 - `javafx.controls`—Controls, layouts and charts
 - `javafx.fxml`—For working with FXML
 - `javafx.graphics`—For working with graphics, animation, CSS (for styling nodes), text and more
 - `javafx.media`—For incorporating audio and video
 - `javafx.swing`—For integrating into JavaFX 9 apps Swing GUI components
 - `javafx.web`—For integrating web content

13.8 JavaFX 9: Java SE 9 JavaFX Updates (cont.)

- ▶ 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

13.8 JavaFX 9: Java SE 9 JavaFX Updates (cont.)

New Public Skinning APIs

- ▶ 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**

13.8 JavaFX 9: Java SE 9 JavaFX Updates (cont.)

New Public Skinning APIs (cont.)

- ▶ 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

13.8 JavaFX 9: Java SE 9 JavaFX Updates (cont.)

GTK+ 3 Support on Linux

- ▶ 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

- ▶ 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.

13.8 JavaFX 9: Java SE 9 JavaFX Updates (cont.)

Updated GStreamer

- ▶ JavaFX implements its audio and video multimedia capabilities using GStreamer (<https://gstreamer.freedesktop.org>)
- ▶ JavaFX 9 incorporates a more recent version of GStreamer

Updated WebKit

- ▶ WebView control embeds web content in JavaFX apps
- ▶ Based on 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