Chapter 13
JavaFX GUI: Part 2

Java How to Program, 11/e

Questions? E-mail paul.deitel@deitel.com

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.

OBJECTIVES (cont.)

- 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

OUTLINE (cont.)

OUTLINE (cont.)

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 1 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.)

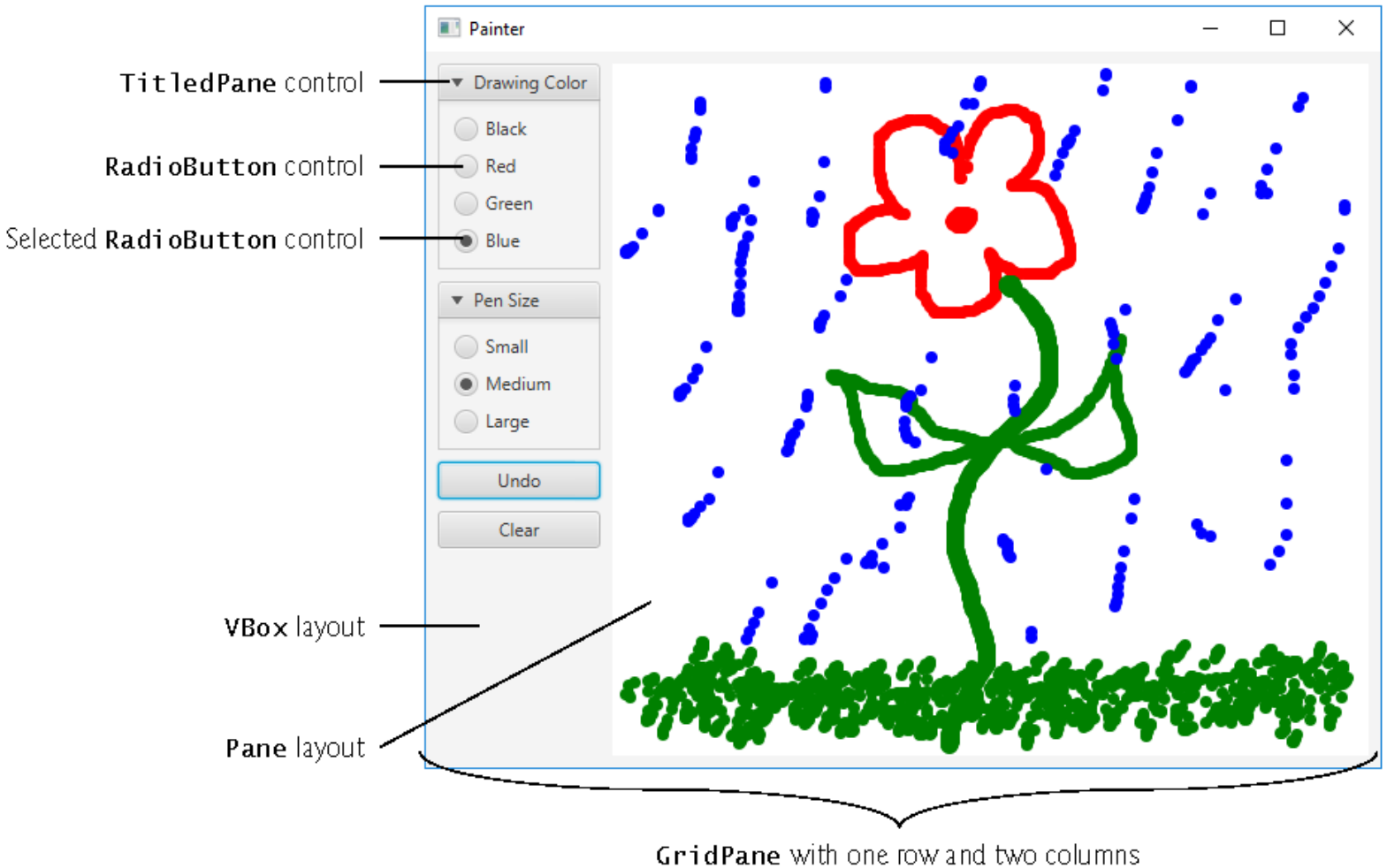


Fig. 13.2 | **Painter** app.

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.

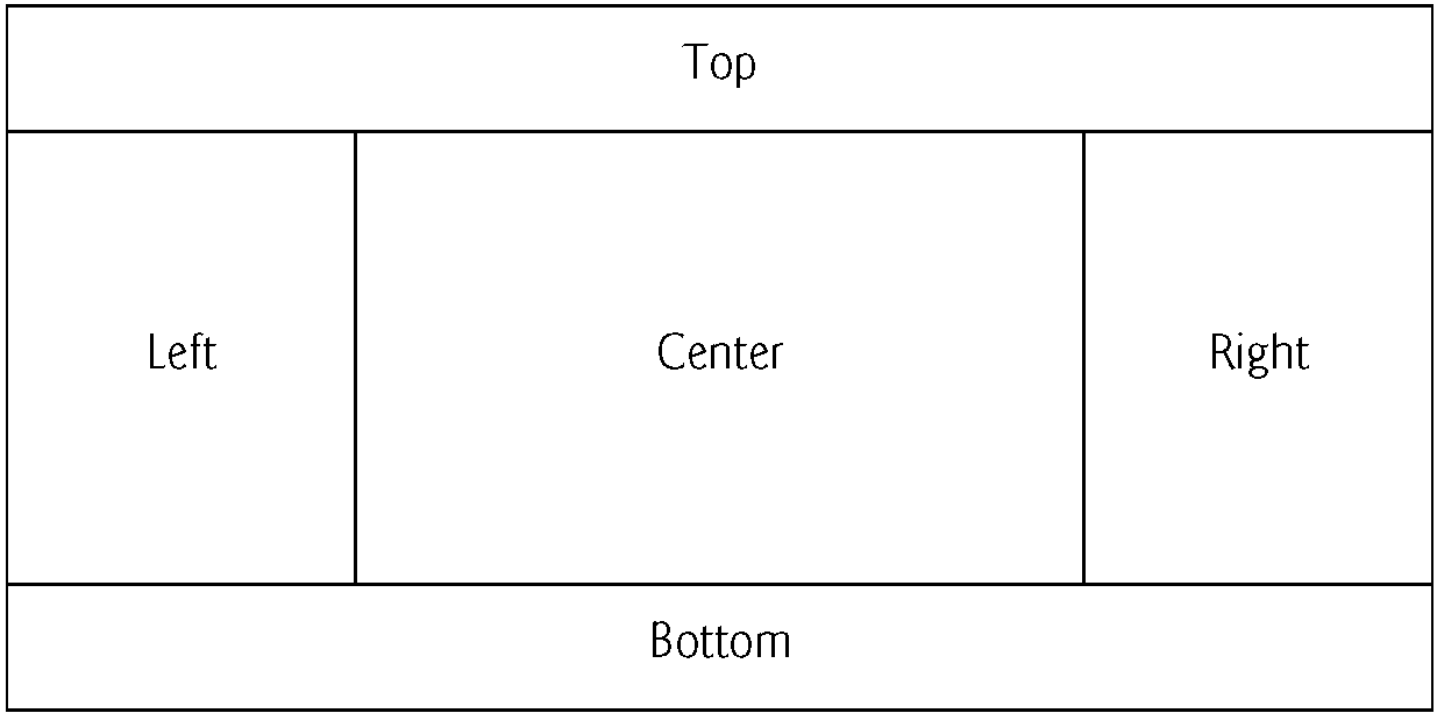


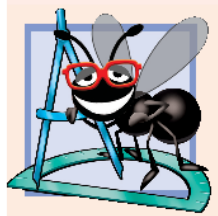
Fig. 13.3 | BorderPane's five areas.

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.

Fig. 13.4 | Mouse events. (Part 1 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.)



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.

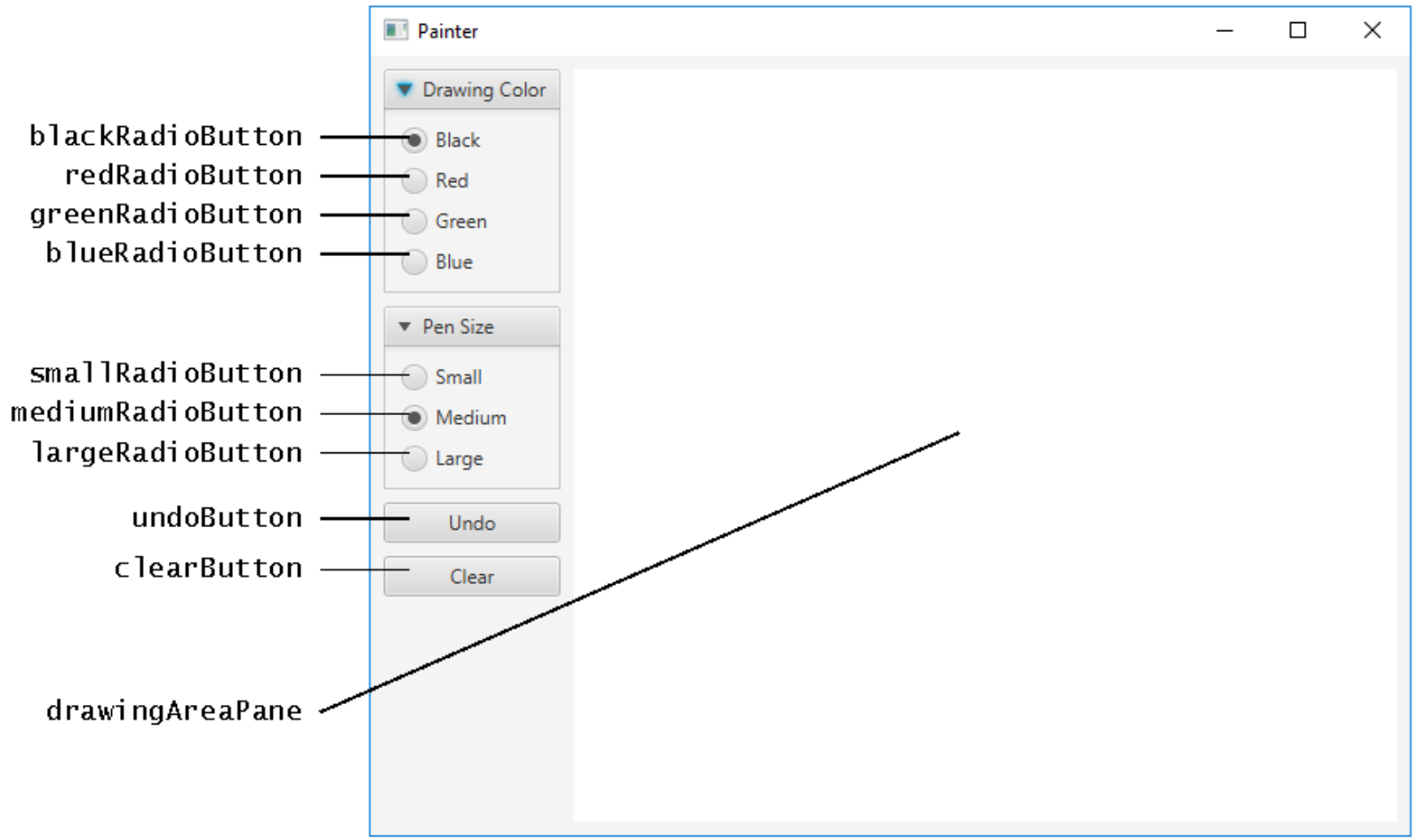


Fig. 13.5 | **Painter** GUI labeled with **fx:id**s for the programmatically manipulated controls.

```
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.

```
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.)

```
13  public class PainterController {
14     // enum representing pen sizes
15     private enum PenSize {
16        SMALL(2),
17        MEDIUM(4),
18        LARGE(6);
19
20        private final int radius;
21
22        PenSize(int radius) {this.radius = radius;} // constructor
23
24        public int getRadius() {return radius;}
25     };
26
```

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.)

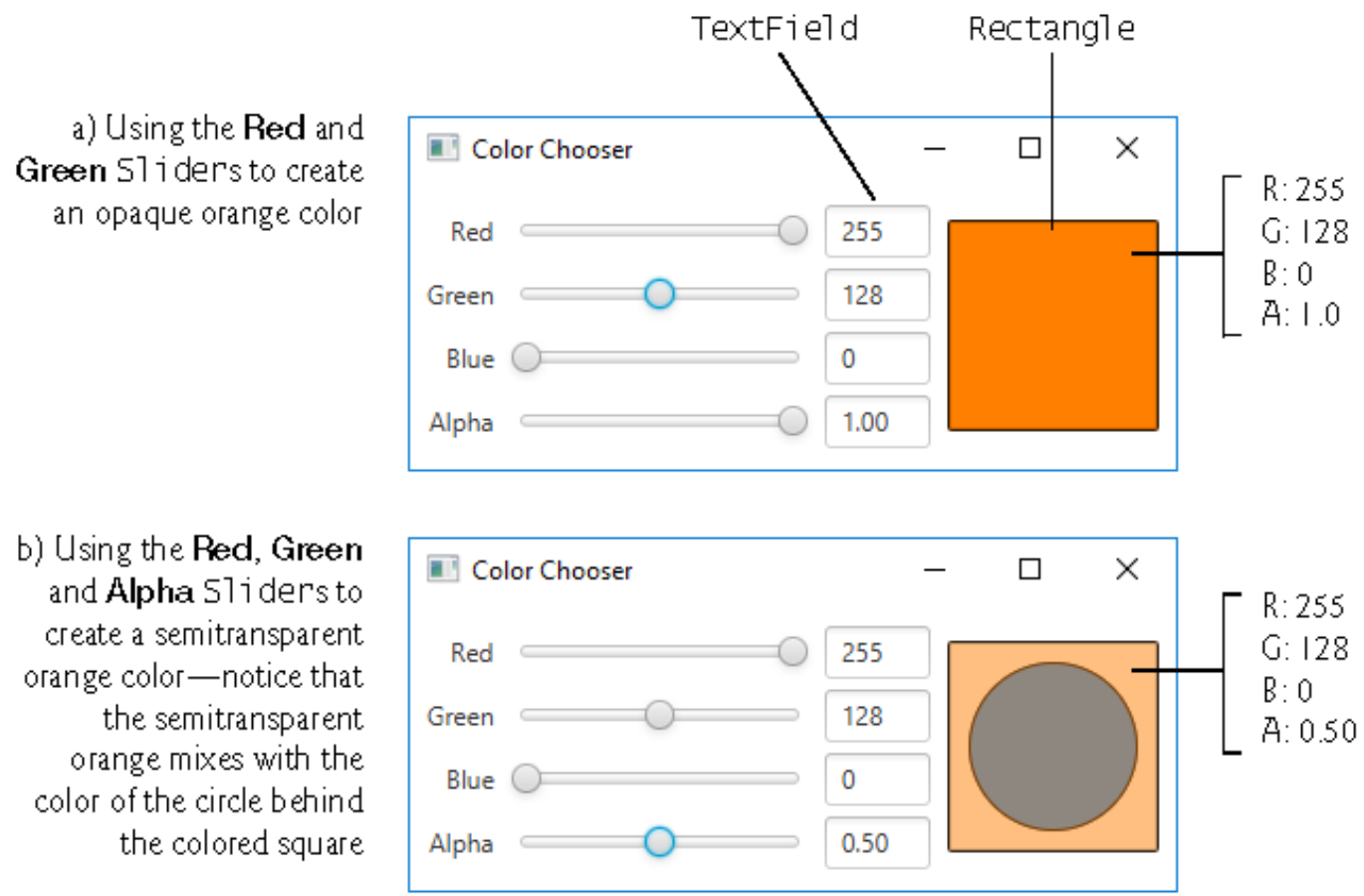
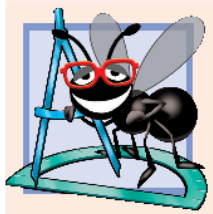


Fig. 13.8 | **Color Chooser** app with opaque and semitransparent orange colors.



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.

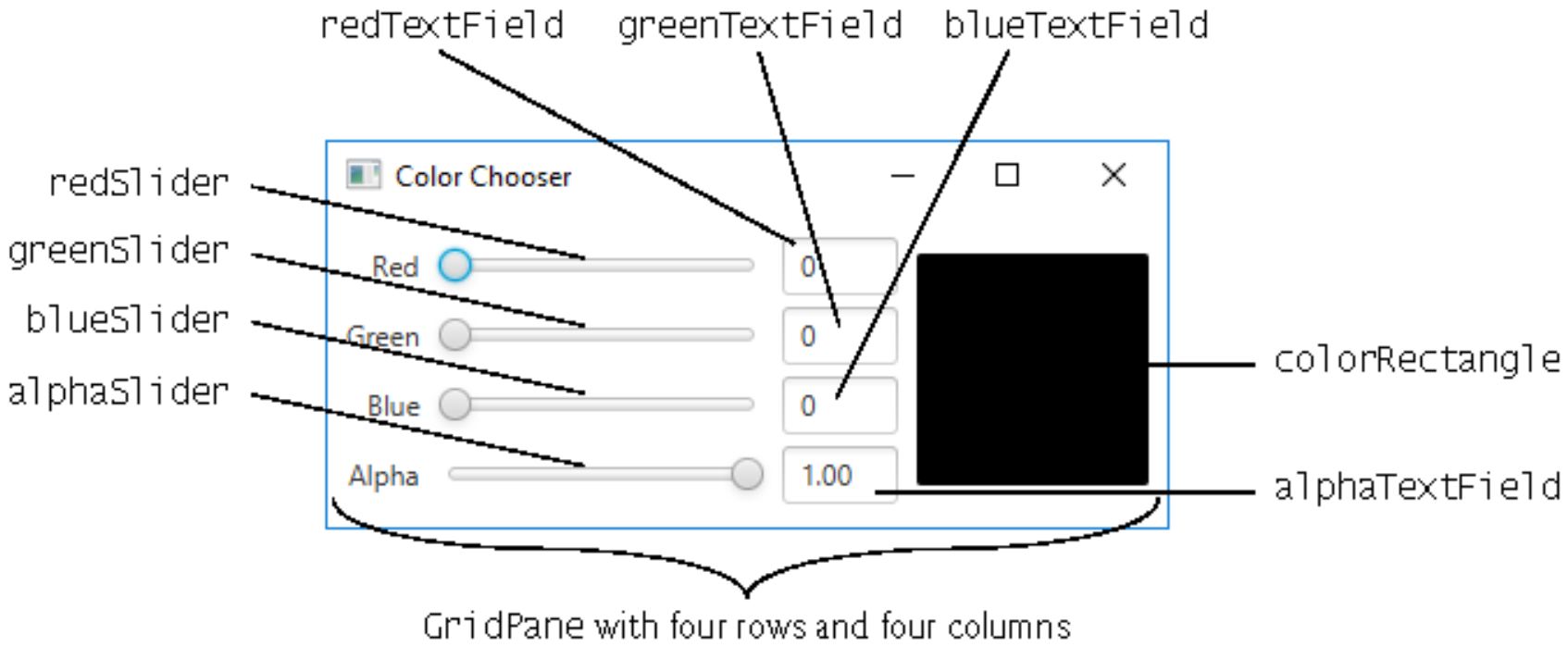


Fig. 13.9 | **Color Chooser** app's programmatically manipulated controls labeled with their **fx:id**s.

```
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 1 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.)

```
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 1 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.)

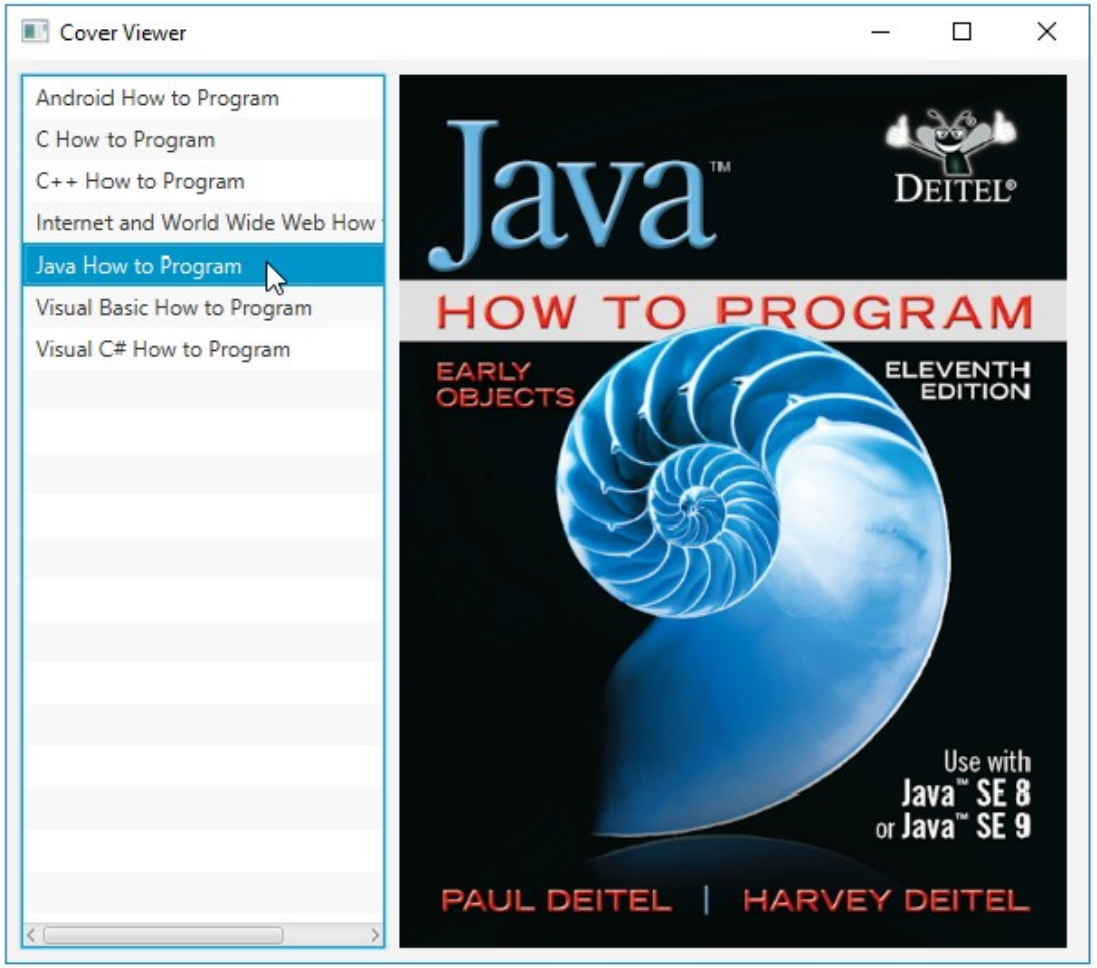


Fig. 13.12 | **Cover Viewer** with *Java How to Program* selected.

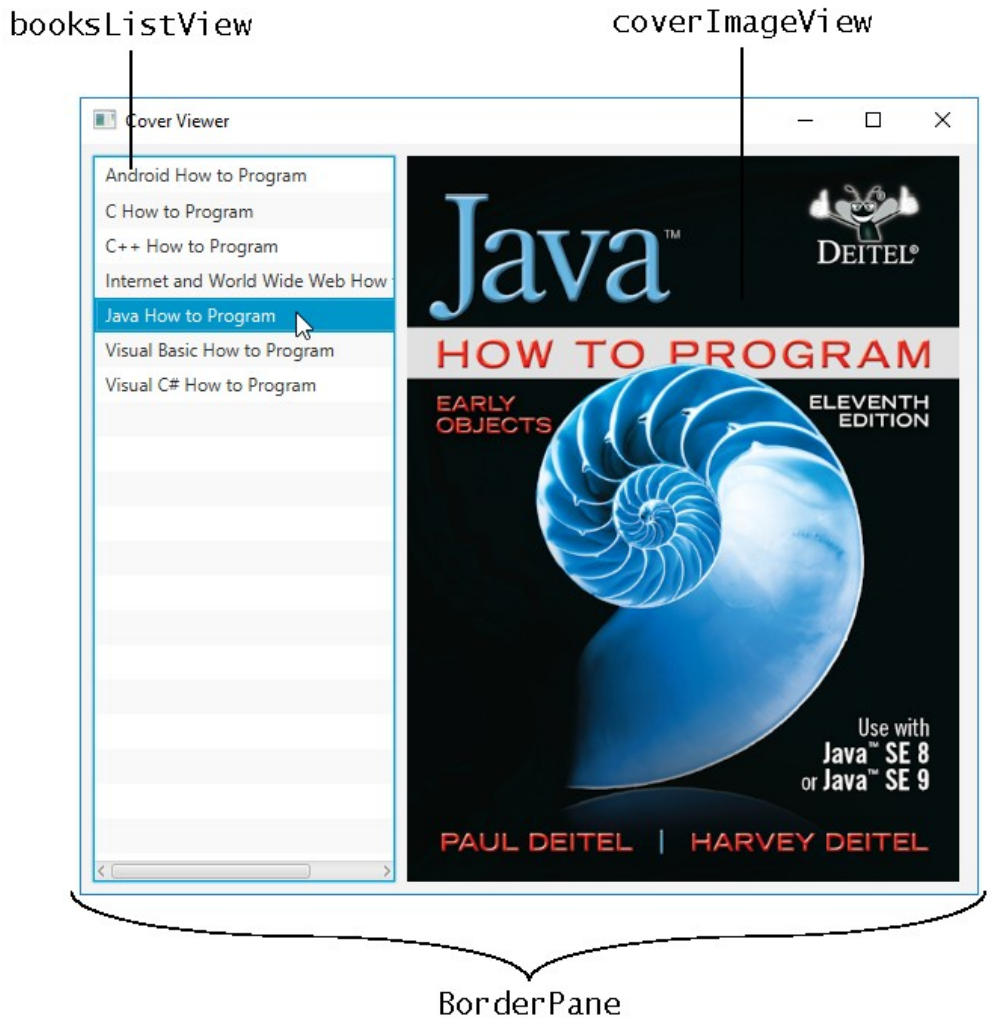


Fig. 13.13 | **Cover Viewer** app's programmatically manipulated controls labeled with their **fx:id**s.

```
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.

```
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 1 of 4.)

```
12  public class CoverViewerController {
13     // instance variables for interacting with GUI
14     @FXML private ListView<Book> booksListView;
15     @FXML private ImageView coverImageView;
16
17     // stores the list of Book Objects
18     private final ObservableList<Book> books =
19        FXCollections.observableArrayList();
20
```

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/androidhtp.jpg", "/images/large/androidhtp.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/cpphtp.jpg", "/images/large/cpphtp.jpg"));
30         books.add(new Book("Internet and World Wide Web How to Program",
31            "/images/small/iw3htp.jpg", "/images/large/iw3htp.jpg"));
32         books.add(new Book("Java How to Program",
33            "/images/small/jhtp.jpg", "/images/large/jhtp.jpg"));
34         books.add(new Book("Visual Basic How to Program",
35            "/images/small/vbhtp.jpg", "/images/large/vbhtp.jpg"));
36         books.add(new Book("Visual C# How to Program",
37            "/images/small/vcshtp.jpg", "/images/large/vcshtp.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.)

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

```
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
```

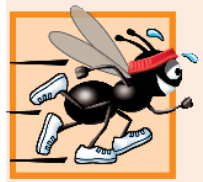
Fig. 13.17 | Custom `ListView` cell factory that displays an image and text.

```
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.setTextAlignment(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.



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`.