13.4 Color Chooser App: Property Bindings and Property Listeners

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

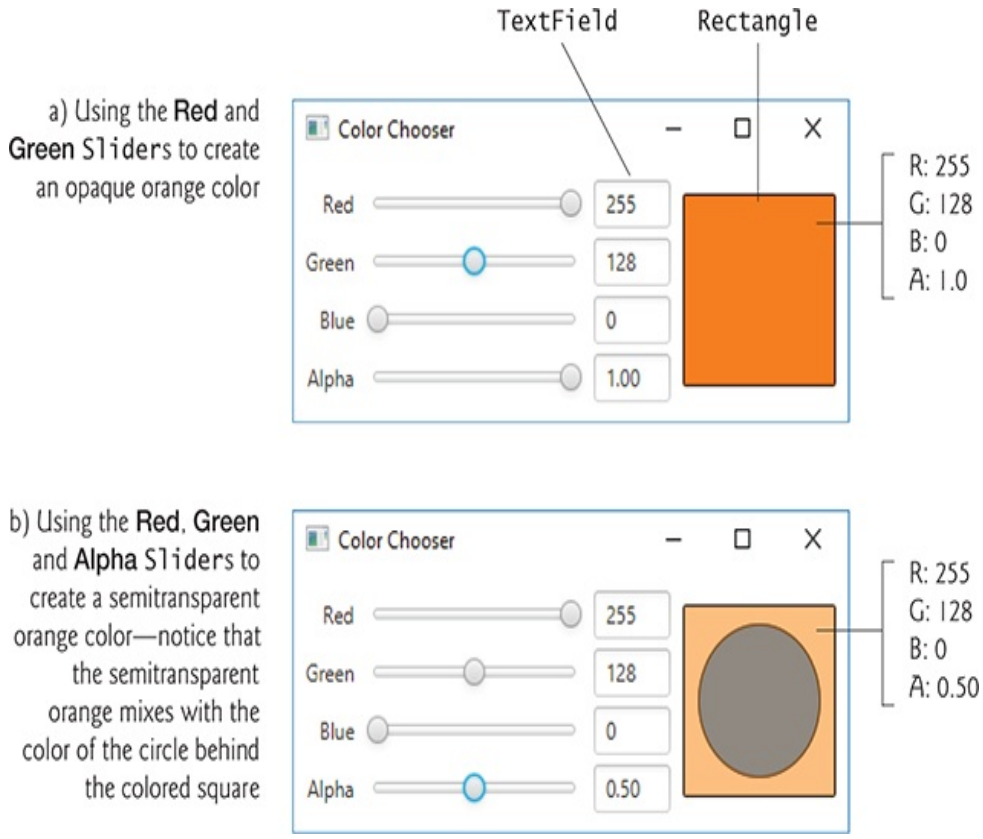

Fig. 13.8

Color Chooser app with opaque and semitransparent orange colors.

Description

13.4.1 Technologies Overview

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

RGBA Colors

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

Properties of a Class

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

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

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

Software Engineering Observation 13.2

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

Property Bindings

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

Property Listeners

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

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

13.4.2 Building the GUI

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

fx:id Property Values for This App's Controls

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

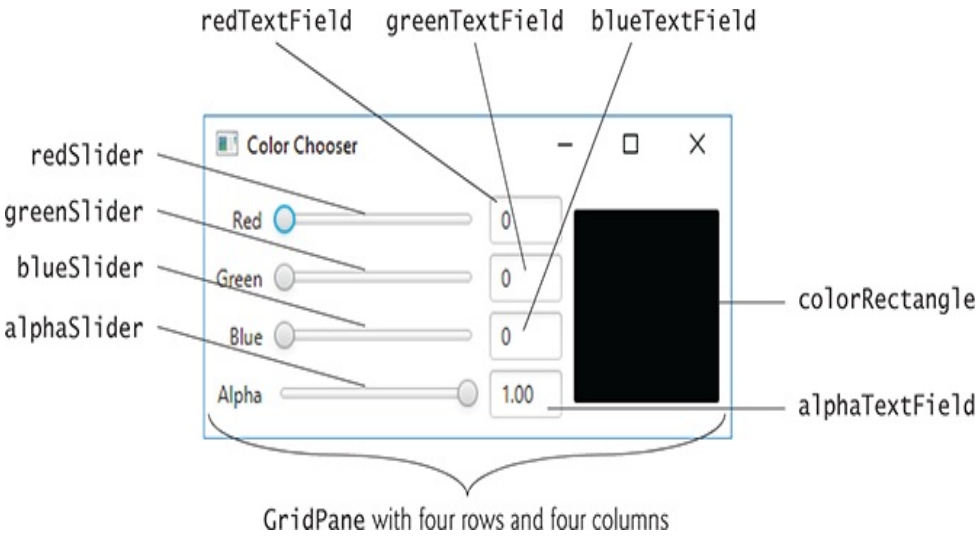

Fig. 13.9

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

Description

Step 1: Adding a `GridPane`

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

Step 2: Configuring the `GridPane`

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

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

Step 3: Adding the Controls

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

Step 4: Configuring the `Sliders`

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

Step 5: Configuring the `TextFields`

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

Step 6: Configuring the `Rectangle`

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

Step 7: Configuring the `Circle`

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

Step 8: Configuring the Rows

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

Step 9: Configuring the Columns

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

Step 10: Configuring the `GridPane`

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

Step 11: Specifying the Controller Class's Name

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

Step 12: Generating a Sample Controller Class

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

13.4.3 `ColorChooser` Subclass of `Application`

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

```
   1  // Fig. 13.8: ColorChooser.java
2  // Main application class that loads and display
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 Excepti
12        Parent root =
13           FXMLLoader.load(getClass().getResource(
14
```

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

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

Fig. 13.10

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

13.4.4 `ColorChooserController` Class

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

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

```
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
11  public class ColorChooserController {
12      // instance variables for interacting with GU
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
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
31          redTextField.textProperty().bind(
32              redSlider.valueProperty().asString("%.0
33          greenTextField.textProperty().bind(
34              greenSlider.valueProperty().asString("%
35          blueTextField.textProperty().bind(
36              blueSlider.valueProperty().asString("%.
37          alphaTextField.textProperty().bind(
38              alphaSlider.valueProperty().asString("%
39
40          // listeners that set Rectangle's fill bas
41          redSlider.valueProperty().addListener(
42              new ChangeListener<Number>() {
43                  @Override
44                  public void changed(ObservableValue<
```

```
45                  Number oldValue, Number newValue)
 46                   red = newValue.intValue();
47                  colorRectangle.setFill(Color.rgb(
               48                }
                49             }
                 50          );
51          greenSlider.valueProperty().addListener(
  52              new ChangeListener<Number>() {
           53                @Override
54              public void changed(ObservableValue<
55                  Number oldValue, Number newValue)
 56                  green = newValue.intValue();
57                  colorRectangle.setFill(Color.rgb(
               58                }
                59             }
                 60          );
61          blueSlider.valueProperty().addListener(
  62              new ChangeListener<Number>() {
           63                @Override
64              public void changed(ObservableValue<
65                  Number oldValue, Number newValue)
 66                   blue = newValue.intValue();
67                  colorRectangle.setFill(Color.rgb(
               68                }
                69             }
                 70          );
71          alphaSlider.valueProperty().addListener(
  72              new ChangeListener<Number>() {
           73                @Override
74              public void changed(ObservableValue<
75                  Number oldValue, Number newValue)
76                  alpha = newValue.doubleValue();
77                  colorRectangle.setFill(Color.rgb(
               78                }
                79             }
                 80          );
                   81       }
                    82    }
```

Fig. 13.11

Controller for the `ColorChooser` app.

Instance Variables

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

Method `initialize`

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

Property-to-Property Bindings

Lines 31–38 set up property bindings between a `Slider`'s value and the corresponding `TextField`'s text so that

changing a `Slider` updates the corresponding `TextField`. Consider lines 31–32, which bind the `redSlider`'s `valueProperty` to the `redTextField`'s `textProperty`:

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

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

Property Listeners

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