13.3 Painter App: `RadioButton`s, Mouse Events and Shapes

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

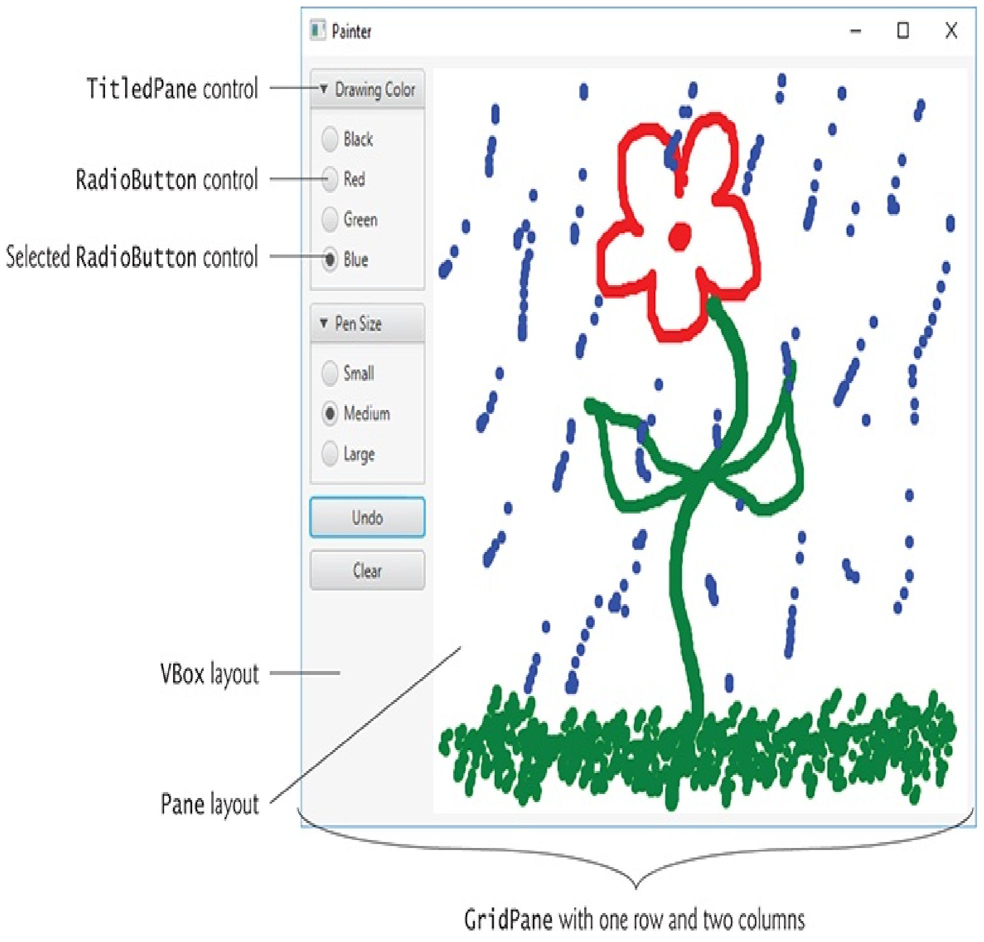

Fig. 13.2

Painter app.

Description

13.3.1 Technologies Overview

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

Painter app.

`RadioButtons` and `ToggleGroups`

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

`BorderPane` Layout Container

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

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

Look-and-Feel Observation 13.1

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

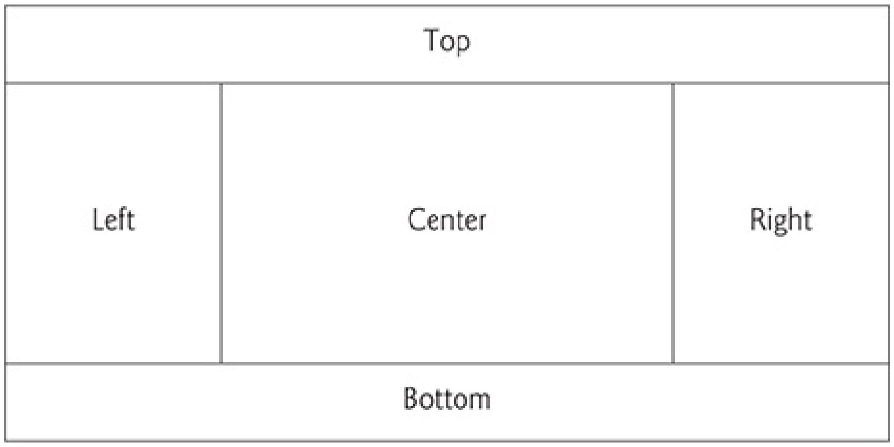

Fig. 13.3

`BorderPane`’s five areas.

`TitledPane` Layout Container

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

JavaFX Shapes

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

Pane Layout Container

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

Mouse Event Handling

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

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

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

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

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

Fig. 13.4

Mouse events.

Setting a Control’s User Data

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

13.3.2 Creating the

`Painter.fxml` File

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

13.3.3 Building the GUI

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

Software Engineering Observation 13.1

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

fx:id Property Values for This App's Controls

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

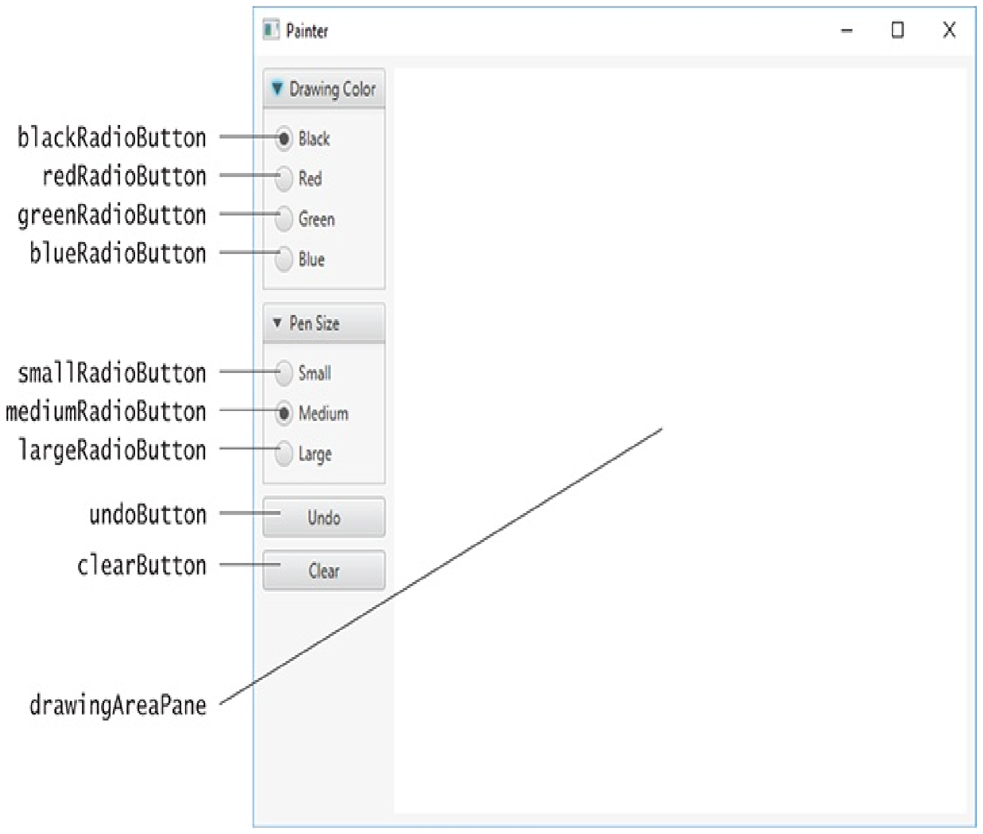

Fig. 13.5

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

Description

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

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

Step 2: Configuring the `BorderPane`

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

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

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

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

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

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

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

Step 5: Customizing the `TitledPanes`

Each `TitledPane` in the completed GUI contains multiple `RadioButton`s. 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 `VBox`es will be sized based on their contents.

Step 6: Adding the `RadioButtons` to the `VBox`

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

Step 7: Specifying the `ToggleGroups` for the

`RadioButtons`

Select all four `RadioButton`s 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 `RadioButton`s will be associated with it to ensure that only one is selected at a time. Repeat this step for the three `RadioButton`s in the second `TitledPane`’s `VBox`, but set the **Toggle Group** property to `sizeToggleGroup`.

Step 8: Changing the `TitledPanes`’ Preferred Width and Height

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

Step 9: Adding the `Buttons`

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

Step 10: Setting the Width the VBox

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

Step 11: Specifying the Controller Class's Name

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

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

Step 12: Specifying the Event-Handler Method Names

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

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

Step 13: Generating a Sample Controller Class

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

13.3.4 `Painter` Subclass of `Application`

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

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

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

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

```
    }

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

Fig. 13.6

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

13.3.5 `PainterController` Class

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

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

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

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

      private final int radius;

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

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

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

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

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

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

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

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

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

      // if there are any shapes remove the last
      if (count > 0) {
         drawingAreaPane.getChildren().remove(co
      }
   }

   // handles Clear Button's ActionEvents
   @FXML
   private void clearButtonPressed(ActionEvent e
      drawingAreaPane.getChildren().clear(); //
   }
}
```

Fig. 13.7

Controller for the **Painter** app.

`PenSize` `enum`

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

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

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

Instance Variables

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

Method `initialize`

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

`drawingAreaMouseDragged` Event Handler

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

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

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

`colorRadioButtonSelected` Event Handler

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

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

`sizeRadioButtonSelected` Event Handler

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

`undoButtonPressed` Event Handler

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

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

`clearButtonPressed` Event Handler

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