4.15 (Optional) GUI and Graphics Case Study: Event Handling; Drawing Lines

An appealing feature of Java is its graphics and multimedia support, which enables you to visually enhance your applications. Chapter 22, JavaFX Graphics and Multimedia, discusses these capabilities in detail.

In this and the next several GUI and Graphics Case Study sections, we introduce a few of JavaFX's graphics capabilities. In this section, you'll build the `DrawLines` app that draws lines on a **Canvas** control—a rectangular area in which you can draw. This app also introduces event handling—when the user clicks a **Button**, a method that we'll designate in Scene Builder will be called by JavaFX to draw the lines on the `Canvas`. Like Section 3.6, this GUI and Graphics Case Study section is a bit longer than the subsequent ones as we walk you through the steps to configure the GUI.

4.15.1 Test-Driving the Completed Draw Lines App

First, let's test-drive the completed app, so you can see it in

action:

1. In a command window, change directories to the `GUIGraphicsCaseStudy04` directory in this chapter's `ch04` examples folder.
2. Compile the app with the following command, which compiles both of this app's Java source-code files:

```
javac *.java
```

3. Run the app with the following command:

```
java DrawLines
```

When the app's window appears on the screen (Fig. 4.17(a)), click the **Draw Lines** `Button`. Figure 4.17(b) shows the app's GUI after the user clicks the **Draw Lines** `Button`.

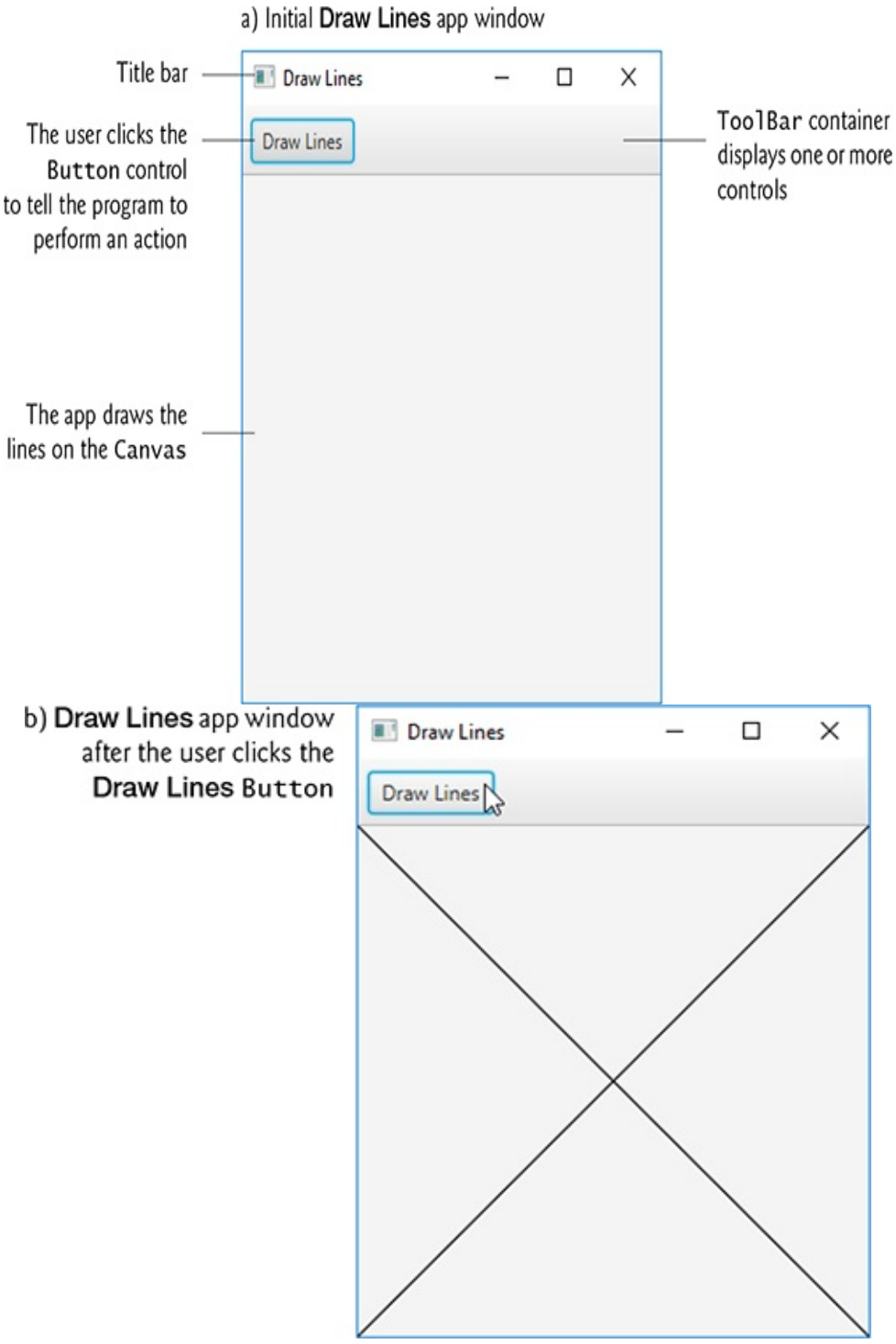
a) Initial Draw Lines app window
Title bar
Draw Lines
The user clicks the Button control to tell the program to perform an action
Draw Lines
ToolBar container displays one or more controls
The app draws the lines on the Canvas
b) Draw Lines app window after the user clicks the Draw Lines Button
Draw Lines
Draw Lines

Fig. 4.17

Draw Lines app in action.

Description

4.15.2 Building the App's GUI

You'll build the app's GUI in `DrawLines.fxml`, using the same techniques you learned in Section 3.6. The GUI contains the following controls:

- a `Button` that you press to draw the lines
- a `Canvas` that displays an image

You'll also use `BorderPane` and `ToolBar` layouts (discussed shortly), to help you arrange the `Button` and `Canvas` in the user interface.

Opening Scene Builder and Creating the File `DrawLines.fxml`

As you did in Section 3.6.4, open Scene Builder, then select **File** > **Save** to display the **Save As** dialog. Select a location in which to store the FXML file, name the file `DrawLines.fxml` and click the **Save** button to create the file.

Creating a `BorderPane` Layout Container

In Section 3.6, you arranged controls in a `VBox` layout container. For the remaining GUI and Graphics Case Study apps, you'll use a **BorderPane layout container**, which arranges controls into one or more of the five regions shown in Fig. 4.18.

Look-and-Feel Observation 4.1

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.*

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. To begin designing the GUI, drag-and-drop a **BorderPane** from the **Library** window's **Containers** section onto Scene Builder's content panel. (You

also can double-click **BorderPane** to add one to the content panel.)

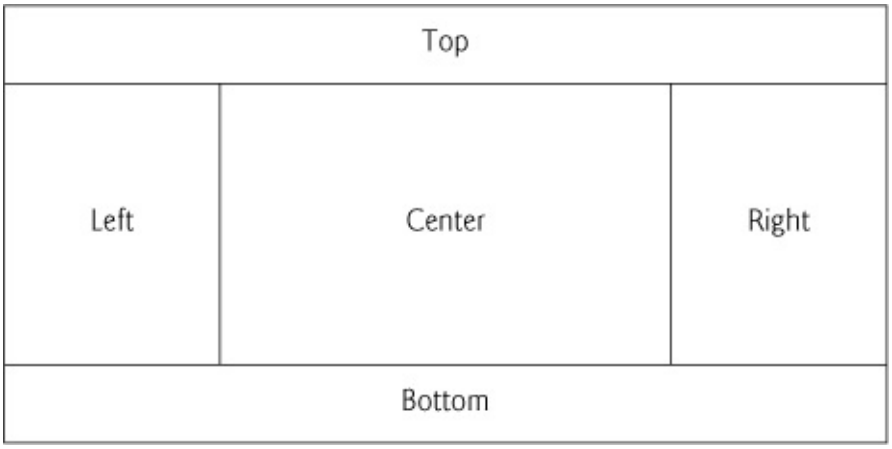

Fig. 4.18

`BorderPane`'s five areas.

Adding a `ToolBar` and Configuring Its `Button`

Next, you'll add a **ToolBar layout container** to the `BorderPane`'s top region. A `ToolBar` arranges controls horizontally by default. In this app, the `ToolBar` will contain only the app's **Draw Lines** `Button`—you'll see a `ToolBar` with two `Button`s in Section 5.11, and later apps in the book will place other controls in a `ToolBar`.

Look-and-Feel Observation 4.2

`ToolBar`*s typically organize multiple controls at a layout's edges, such as in a* `BorderPane`*'s top, right, bottom or left areas.*

Drag a **ToolBar** from the Scene Builder **Library** 's **Containers** section onto the `Border-Pane`'s top area as shown in Fig. 4.19. As you drag over the `BorderPane`, Scene Builder shows you the five regions so you can determine where the `ToolBar` will appear when you drop it. By default, the `ToolBar` you drag onto your layout has one `Button` (Fig. 4.20).

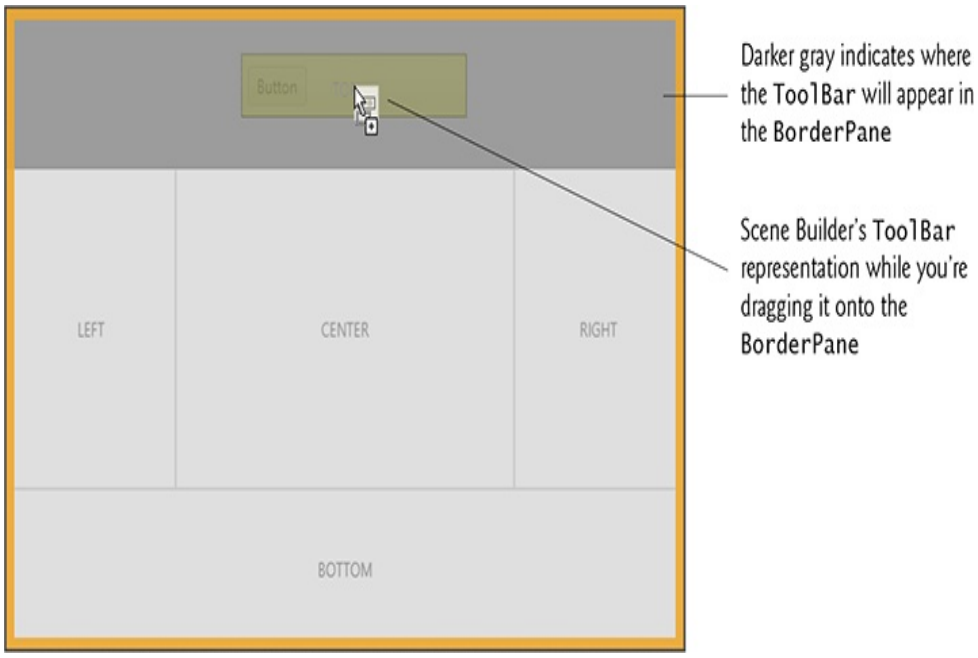

Fig. 4.19

Dragging a `ToolBar` onto a `BorderPane`.

Fig. 4.20

Initial `ToolBar` containing a `Button`—by default the `ToolBar` is the full width of the `BorderPane`'s top area.

Configuring the `Button`'s

Text

Next, double-click the `Button` to edit its text. Type `Draw Lines` and press *Enter*. The updated `Button` is shown in Fig. 4.21.

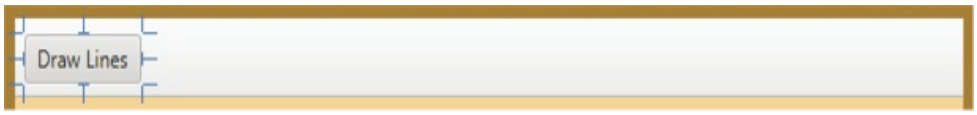

Fig. 4.21

`ToolBar` containing the `Button` with its updated text.

Adding and Configuring a `Canvas`

Next, you'll create the `Canvas` in which the app draws lines. Drag-and-drop a **Canvas** from the Scene Builder **Library** 's **Miscellaneous** section to the `BorderPane`'s center area, as shown in Fig. 4.22. Scene Builder automatically creates a 200-by-200 pixel `Canvas` and centers it horizontally and vertically in the `BorderPane`'s center area.

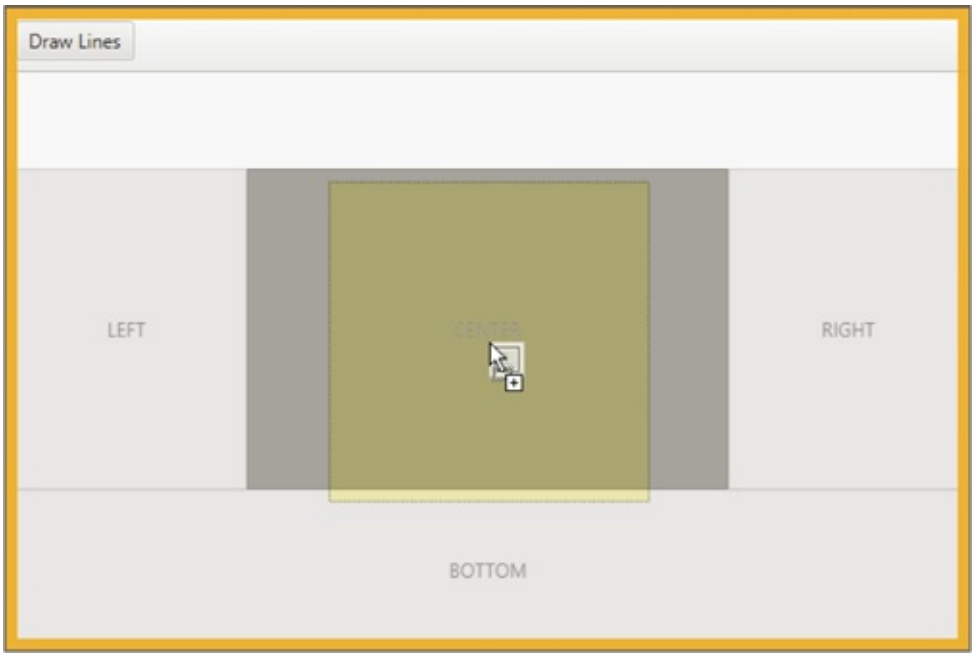

Fig. 4.22

Dragging and dropping the `Canvas` onto the `BorderPane`'s center area.

Setting the `Canvas`'s Width and Height

For this app, we increased the size of the `Canvas` to 300-by-300, which is the size we'll use in many of the subsequent GUI and Graphics Case Study apps. To do so:

1. Click the `Canvas` once to select it.
2. Expand the **Inspector** 's **Layout** section by clicking the right arrow ( ) next to **Layout**.

3. Type `300` for the **Width** property's value and press *Enter.*
4. Repeat *Step 3* for the **Height** property's value.

Configuring the BorderPane Layout Container's Size

Finally, you'll specify that the `BorderPane` should size itself, based on its contents. Recall from Section 3.6.7 that the preferred width and height of a GUI's primary layout (in this case, the `BorderPane`) determines the default size of the JavaFX app's window. To set the `BorderPane`'s preferred size:

1. Select the `BorderPane` by clicking it in the content panel or by clicking it in the Scene Builder **Document** window's **Hierarchy** section (located in Scene Builder's bottom-left corner).
2. In the **Inspector** 's **Layout** section, click the **Pref Width** property's down arrow and select `USE_COMPUTED_SIZE` (Fig. 4.23). This tells the `BorderPane` to calculate its preferred width, based on the width of its contents. In this app, the widest item is the `Canvas`, so the `BorderPane` initially will be 300 pixels wide.
3. Repeat *Step 2* for the **Pref Height** property. This tells the `BorderPane` to calculate its preferred height, based on the height of its contents. In this case, the `Border-Pane`'s height will be the sum of the `ToolBar`'s and `Canvas`'s heights.

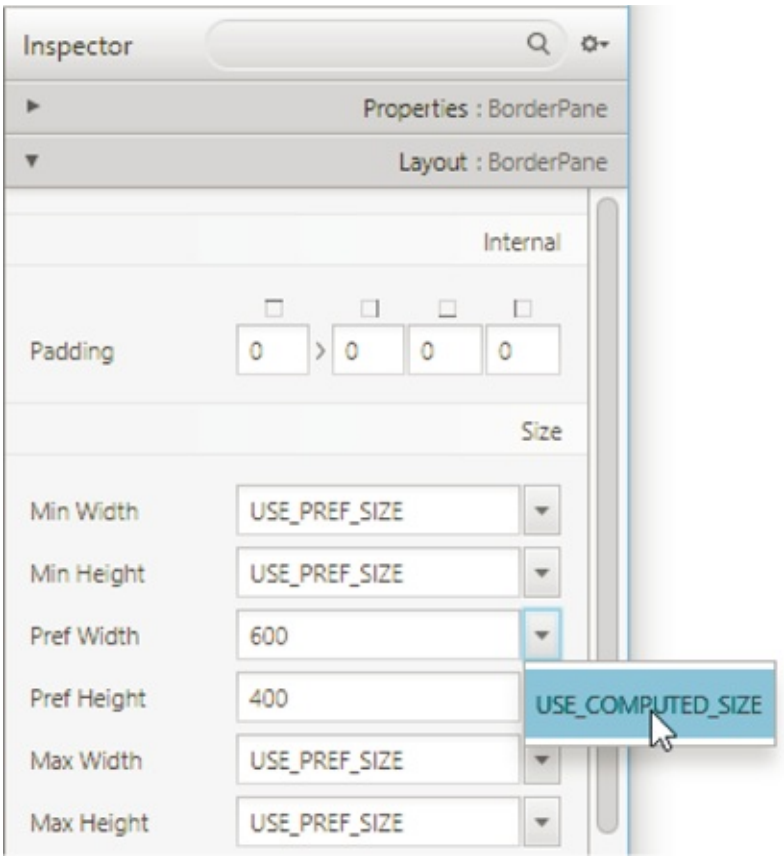

Fig. 4.23

Selecting `USE_COMPUTED_SIZE` for the `BorderPane`'s **Pref Width** property.

Description

Saving the Design

You've now completed the GUI. The design should appear as shown in Fig. 4.24. Save the FXML file by selecting **File > Save**.

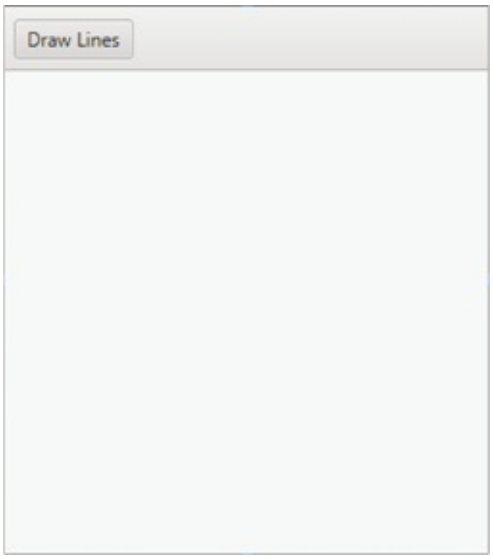

Fig. 4.24

Completed GUI in Scene Builder’s content panel.

4.15.3 Preparing to Interact with the GUI Programmatically

You’ve created objects (like `Scanner`, `Account` and `Student`), assigned them to variables and interacted with the objects programmatically through those variables. JavaFX GUI controls also are objects that you can interact with programmatically. For GUIs created in Scene Builder, *JavaFX creates the controls for you* when the app begins executing.

As you saw in Section 4.15.1, when you press the app’s **Draw**

Lines `Button`, the app responds to that event by drawing lines on the `Canvas`. In this section, you'll learn how to use Scene Builder to specify the names of variables and methods that will enable you to interact with the GUI's controls. We'll then show that Scene Builder can generate a class containing those variables and methods.

Specifying the App's Controller Class

Of course, instance variables and methods must be declared in a class. For an FXML GUI, these are declared in a class known as the **controller class**, because it controls what the app does in response to the user's GUI interactions. You'll see this app's controller class in Section 4.15.4. When you launch a JavaFX app with an FXML GUI, the JavaFX runtime:

- creates the GUI,
- creates an object of the controller class that you specified in Scene Builder,
- initializes any of the controller object's instance variables that you specified in Scene Builder, so they refer to the corresponding control objects, and
- configures any of the controller object's event handlers that you specified in Scene Builder, so that they'll be called when the user interacts with the corresponding control(s).

To specify the controller class's name:

1. Expand the Scene Builder **Document** window's **Controller** section

(located in Scene Builder's bottom-left corner).

2. For the **Controller Class**, type `DrawLinesController` and press *Enter*. By convention, the controller class has the same base name as the FXML file (`DrawLines` in `DrawLines.fxml`) followed by `Controller`.

Specifying the `Canvas`'s Instance Variable Name

To draw on the `Canvas` object, the program needs a variable that refers to it—this will be an instance variable in the controller class. A control's **fx:id property** specifies the instance variable's name—when the app begins executing, JavaFX initializes the instance variable with the corresponding control object. To specify the `Canvas`'s **fx:id** :

1. Select the `Canvas`, then expand the Scene Builder **Inspector** 's **Code** section by clicking the right arrow () next to **Code**.
2. For the **fx:id** property, type `canvas` and press *Enter*.

Specifying the `Button`'s Event-Handler Method

An **event handler** is a method that's called in response to a user interaction, such as when the user clicks this app's **Draw Lines** `Button`. The method is called by the JavaFX runtime on an object of the controller class. To specify the `Button`

event handler’s name:

1. Click the `Button` to select it.
2. For the **On Action** property in the **Inspector** ’s **Code** section, type the method name `drawLinesButtonPressed` and press *Enter*.
3. Save the FXML file by selecting **File > Save**.

Generating the Initial Controller Class

Scene Builder can generate a `DrawLinesController` class—which it calls the *controller skeleton*—containing the instance variables and event handlers you specified. Select **View > Show Sample Controller Skeleton** to display the class (Fig. 4.25), which has the controller-class name you specified, an instance variable with the `Canvas`’s **fx:id** and the empty event-handler method `drawLinesButtonPressed`. We’ll discuss the `@FXML` annotations shortly. To use this code in your controller class, click the **Copy** button, then create a Java source-code file named `DrawLinesController.java` in the same folder as `DrawLines.fxml`, paste the code into that file and save the file. You can add the program’s logic into the file.

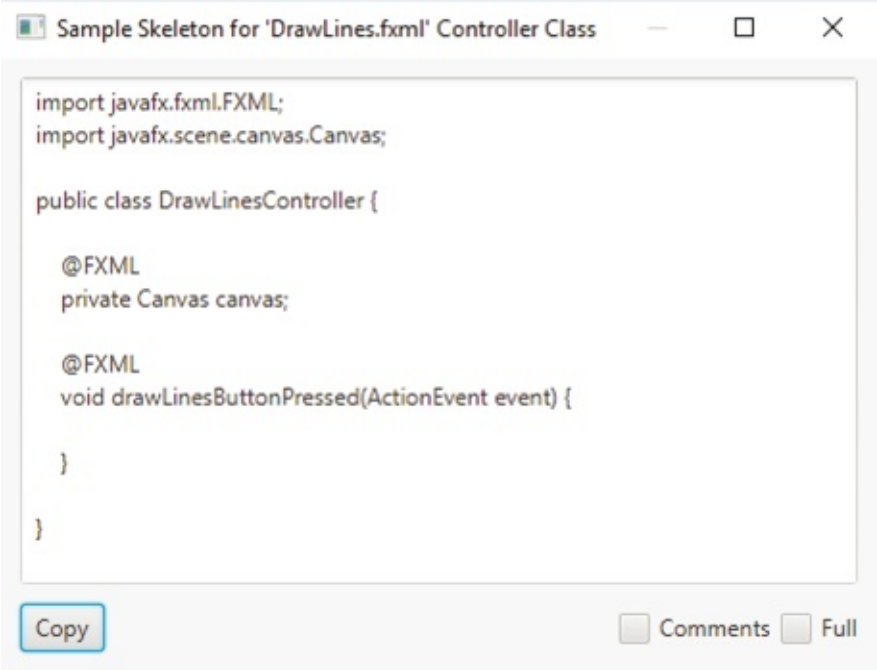

Fig. 4.25

Skeleton controller-class code generated by Scene Builder.

Description

4.15.4 Class `DrawLinesController`

Each JavaFX app you'll see in this book has at least two Java source-code files:

- A controller class that defines the GUI's event handlers and logic. In this app, `DrawLinesController.java` (Fig. 4.26) contains the controller class.

- A main-application class that loads the app's FXML file, creates the GUI then creates and configures the corresponding controller-class object. In this app, `Draw-Lines.java` (Fig. 4.28) contains the main-application class. By convention, this class has the same base name as the FXML file (`DrawLines` in `DrawLines.fxml`).

Class `DrawLinesController` (Fig. 4.26) performs the drawing. The `import` statements in lines 5–6 allow us to use two classes from the package `javafx.scene.canvas`:

- `Canvas` provides the drawing area.
- **GraphicsContext** provides various methods for drawing onto a `Canvas`—every `Canvas` has a `GraphicsContext` object.

```
   1  // Fig. 4.26: DrawLinesController.java
2  // Using strokeLine to connect the corners of a c
      3  import javafx.event.ActionEvent;
          4  import javafx.fxml.FXML;
    5  import javafx.scene.canvas.Canvas;
 6  import javafx.scene.canvas.GraphicsContext;
                      7
     8  public class DrawLinesController {
 9     @FXML private Canvas canvas; // used to get th
                     10
11     // when user presses Draw Lines button, draw t
                 12     @FXML
13     void drawLinesButtonPressed(ActionEvent event)
14        // get the GraphicsContext, which is used t
15        GraphicsContext gc = canvas.getGraphicsCont
                     16
17        // draw line from upper-left to lower-right
18        gc.strokeLine(0, 0, canvas.getWidth(), canv
                     19
20       // draw line from upper-right to lower-left
21       gc.strokeLine(canvas.getWidth(), 0, 0, canva
                  22    }
                    23  }
```

Fig. 4.26

Using `strokeLine` to connect the corners of a canvas.

@FXML Annotation

Recall from Section 4.15.3 that each control we manipulate programmatically needs an **fx:id**. Line 9 in Fig. 4.26 declares the controller class's corresponding instance variable `canvas`. The **@FXML annotation** preceding the declaration indicates that the variable name can be used in the GUI's FXML file. The variable name you specify in the controller class must precisely match the **fx:id** you specified when building the GUI. Note that we did *not* initialize this variable. When the `DrawLines` app executes, it initializes the `canvas` variable with the `Canvas` object from `DrawLines.fxml`.

The `@FXML` annotation preceding the method (line 12) indicates that this method can be used in the FXML file to specify a control's event handler. When the **Draw Lines** app executes, it configures `drawLinesButtonPressed` (lines 12–22) to execute when the user presses the **Draw Lines** `Button`. A `Button` press generates an `ActionEvent` object that contains information about the event that occurred. The corresponding event-handling method must return `void`

and receive one `ActionEvent` parameter (line 13). (We don't use the information in `ActionEvent` parameter in this example.)

Getting a `GraphicsContext` to Draw on a `Canvas`

Every `Canvas` has a `GraphicsContext` object for drawing on the `Canvas`. You obtain the `GraphicsContext` by calling the `Canvas`'s **getGraphicsContext2D** method (line 15):

```
GraphicsContext gc = canvas.getGraphicsContext2D();
```

By convention, the variable name `gc` is used for this object.

`Canvas`'s Coordinate System

A **coordinate system** (Fig. 4.27) is a scheme for identifying points. In `Canvas`'s coordinate system, the upper-left corner has the coordinates (0, 0). An *x*–*y* coordinate pair is composed of an ***x*-coordinate** (the **horizontal coordinate**) and a ***y*-**

coordinate (the **vertical coordinate**). The *x*-coordinate is the horizontal location moving from *left to right*. The *y*-coordinate is the vertical location moving from *top to bottom*. The ***x*-axis** describes every horizontal coordinate, and the ***y*-axis** describes every vertical coordinate. Coordinates indicate where graphics should be displayed. Coordinate units are measured in **pixels**. The term pixel stands for “picture element.” A pixel is a display monitor’s smallest addressable element.

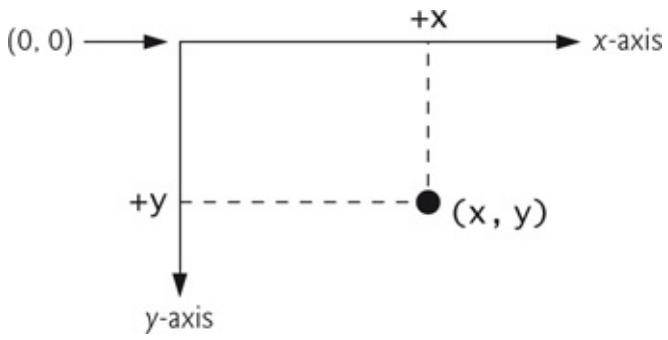

Fig. 4.27

Java coordinate system. Units are measured in pixels.

Drawing Lines

Lines 18 and 21 in Fig. 4.26 use `GraphicsContext` variable `gc` to draw lines that diagonally connect the `Canvas`’s corners. Class `GraphicsContext`’s **strokeLine** method requires four arguments to draw a line—the first two are the start point’s *x*-*y* coordinates and the last two are the end point’s *x*-*y* coordinates. In line 18, the first two arguments

represent the `Canvas`'s upper-left corner and the last two represent the `Canvas`'s bottom-right corner. `Canvas` methods **getWidth** and **getHeight** return the `Canvas`'s width and height in pixels, respectively. In line 21, the first two arguments represent the `Canvas`'s upper-right corner and the last two represent the `Canvas`'s bottom-left corner.

4.15.5 Class `DrawLines`—The Main Application Class

Figure 4.28 shows class `DrawLines`—the app's main-application class that begins the program's execution, creates the GUI from the FXML file, creates and configures the controller-class object and displays the GUI in a window. Every JavaFX app we present in this book uses a similar main-application class in which we make only three code changes (separate from the comments)—the class name (line 9), the FXML filename (line 14) and the text that's displayed in the application window's title bar (line 17). This class uses various concepts that we discuss in later chapters. For now, use class `DrawLines` as a template for the main-application class in each GUI and Graphics Case Study example and exercise. Simply copy it, rename it, then edit lines 9, 14 and 17. We discuss the details of a JavaFX app's main-application class in Section 12.5.4.

```
1  // Fig. 4.28: DrawLines.java
2  // Main application class that loads and displays
```

```
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 DrawLines extends Application {
10     @Override
11     public void start(Stage stage) throws Exceptio
12        // loads DrawLines.fxml and configures the
13        Parent root =
14           FXMLLoader.load(getClass().getResource("
15
16        Scene scene = new Scene(root); // attach sc
17        stage.setTitle("Draw Lines"); // displayed
18        stage.setScene(scene); // attach scene to s
19        stage.show(); // display the stage
20     }
21
22     // application execution starts here
23     public static void main(String[] args) {
24        launch(args); // create a DrawLines object
25     }
26  }
```

Fig. 4.28

Main application class that loads and displays the **Draw Lines** GUI.

GUI and Graphics Case

Study Exercises

1. **4.1** Using loops and control statements to draw lines can lead to many interesting designs.

    1. Create the design in Fig. 4.29 . This design draws lines from the top-left corner, fanning them out until they cover the upper-left half of the `Canvas`. One approach is to divide the width and height into an equal number of steps (we found 20 steps works well). The first endpoint of each line will be the top-left corner (0, 0). The second endpoint can be found by starting at the bottom-left corner and moving up one vertical step and right one horizontal step. Draw a line between the two endpoints. Continue moving up and to the right one step to find each successive endpoint.

Fig. 4.29

Lines fanning from a corner.

    2. Modify part (a) to have lines fan out from all four corners, as shown in Fig. 4.30 . Lines from opposite corners should intersect

along the middle.

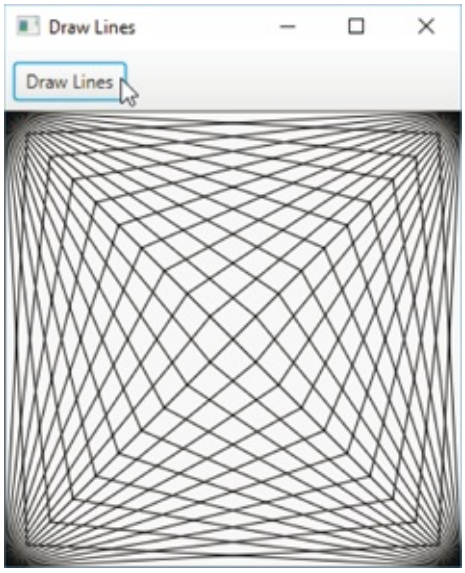

Fig. 4.30

Lines fanning from all four corners.

2. **4.2** Figures 4.31 –4.32 display two additional designs created using loops and `strokeLine`.

    1. Create the design in Fig. 4.31 . Begin by dividing each edge into an equal number of increments (we chose 20 again). The first line starts in the top-left corner and ends one step right on the bottom edge. For each successive line, move down one increment on the left edge and right one increment on the bottom edge. Continue drawing lines until you reach the bottom-right corner.

    2. Modify part (a) to mirror the design in all four corners, as shown in Fig. 4.32 .

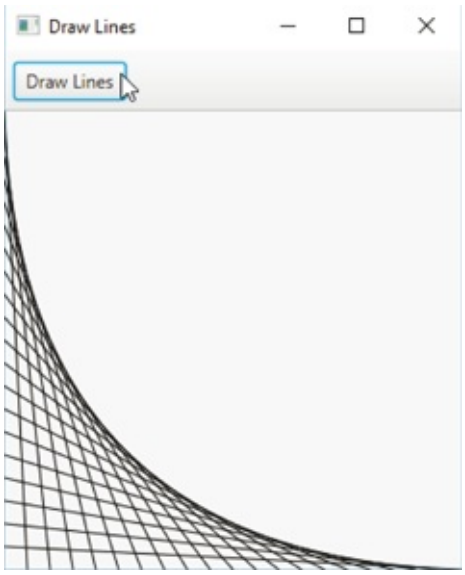

Fig. 4.31

Line art with loops and `strokeLine`.

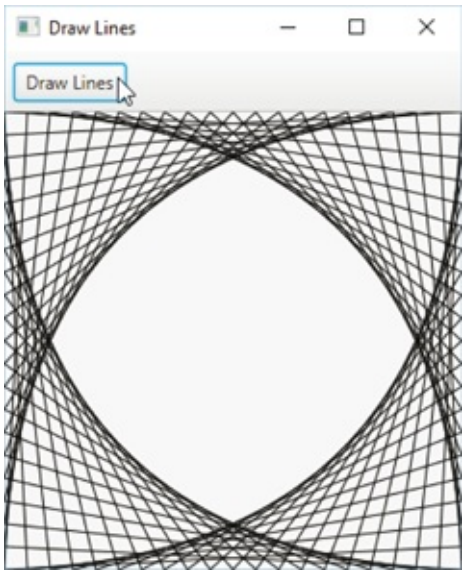

Fig. 4.32

Line art with loops and `strokeLine`—repeating from all four corners.