5.11 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals

In this section, we'll discuss the `Draw Shapes` app, which demonstrates drawing rectangles and ovals using the `GraphicsContext` methods `strokeRect` and `strokeOval`, respectively.

Creating the Draw Shapes App's GUI

The app's GUI is defined in `DrawShapes.fxml` using the techniques you learned in Sections 3.6 and 4.15, and is nearly identical to the GUI in Fig. 4.17. To build the GUI, we copied `DrawLines.fxml` into a new directory and renamed it `DrawShapes.fxml`, then opened `DrawShapes.fxml` in Scene Builder and made the following changes:

- We specified `DrawShapesController` as the app's **Controller class** in the Scene Builder **Document** window's **Controller** section.
- We set the `Button`'s **Text** property to `Draw Rectangles` (in the **Inspector**'s **Properties** section) and set the `Button`'s **On Action** event

handler to `drawRectanglesButtonPressed` (**Inspector**'s **Code** section).

- We dragged another `Button` from the Scene Builder **Library**'s **Controls** section onto the `ToolBar`, then set that `Button`'s **Text** property to `Draw Ovals` and set its **On Action** event handler to `drawOvalsButtonPressed`.

If necessary, review the steps presented in Sections 4.15.2–4.15.3 to make these changes.

Class `DrawLinesController`

As in the prior Java FX example, this app has two Java source-code files:

- `DrawShapes.java` contains the JavaFX app's `main` application class that loads `DrawShapes.fxml` and configures the `DrawShapesController`.
- `DrawShapesController.java` contains the controller class that draws either rectangles or ovals, based on which `Button` the user presses.

To create `DrawShapes.java`, we copied `DrawLines.java`, renamed it and made the code edits described in Section 4.15.5. We do not show `DrawShapes.java`, because the only changes are the class name (`DrawShapes`), the name of the FXML file to load (`Draw-Shapes.fxml`) and the text displayed in the stage's title bar (`Draw Shapes`).

Class `DrawLinesController` (Fig. 5.27) performs the drawing when the user clicks either of the app's `Button`s. When the `DrawShapes` app executes, it will

- configure the `DrawShapesController`'s `canvas` variable (line 9) to refer to the `Canvas` from `DrawShapes.fxml`,
- configure the method `drawRectanglesButtonPressed` (lines 12–15) to execute when the user presses the **Draw Rectangles** `Button`, and
- configure the method `drawOvalsButtonPressed` (lines 18–21) to execute when the user presses the **Draw Ovals** `Button`.

Methods `drawRectanglesButtonPressed` and `drawOvalsButtonPressed` each call method `draw` (lines 24–47)—passing the arguments `"rectangles"` or `"ovals"`, respectively, to indicate which shapes `draw` should display.

```
   1   // Fig. 5.27: DrawShapesController.java
 2   // Using strokeRect and strokeOval to draw recta
       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 DrawShapesController {
        9      @FXML private Canvas canvas;
                       10
 11      // when user presses Draw Rectangles button,
                  12      @FXML
 13      void strokeRectanglesButtonPressed(ActionEven
          14         draw("rectangles");
                    15      }
                       16
 17      // when user presses Draw Ovals button, call
                  18      @FXML
```

```
19      void strokeOvalsButtonPressed(ActionEvent eve
            20          draw("ovals");
                    21      }
                        22
23      // draws rectangles or ovals based on which B
    24      public void draw(String choice) {
25          // get the GraphicsContext, which is used
26          GraphicsContext gc = canvas.getGraphicsCon
                        27
28          // clear the canvas for next set of shapes
29          gc.clearRect(0, 0, canvas.getWidth(), canv
                        30
             31         int step = 10;
                        32
     33          // draw 10 overlapping shapes
    34          for (int i = 0; i < 10; i++) {
35              // pick the shape based on the user's c
         36             switch (choice) {
37                  case "rectangles": // draw rectangle
38                      gc.strokeRect(10 + i * step, 10 +
39                          90 + i * step, 90 + i * step);
            40                      break;
   41                  case "ovals": // draw ovals
42                      gc.strokeOval(10 + i * step, 10 +
43                          90 + i * step, 90 + i * step);
              44                    break
                   45             }
                    46          }
                      47      }
                        48   }
```

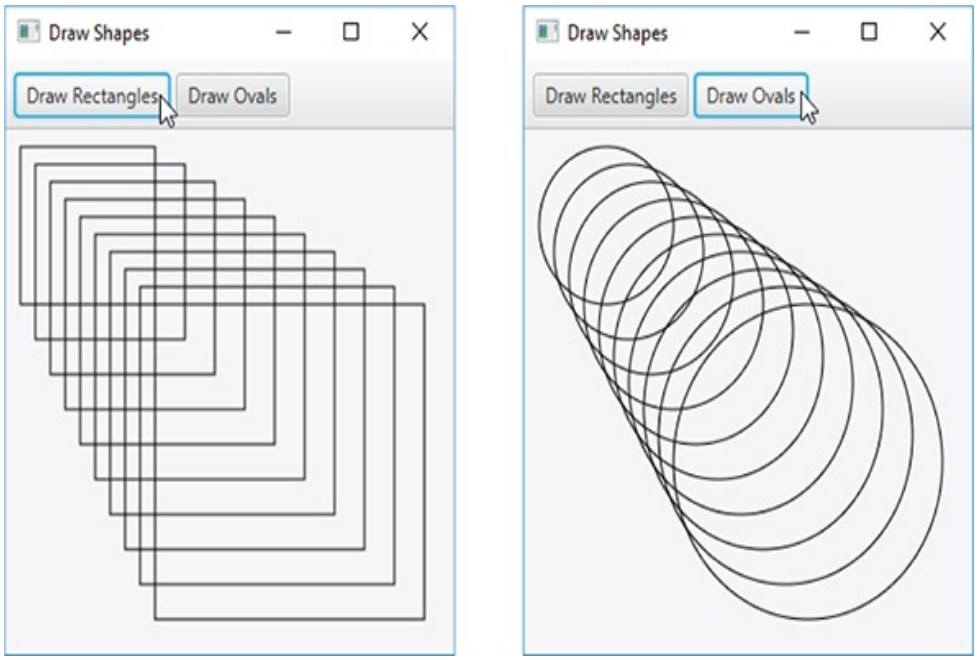

Fig. 5.27

Using `strokeRect` and `strokeOval` to draw rectangles and ovals.

Description

Method `draw` performs the actual drawing. Each time the method is called, it uses `GraphicsContext` method `clearRect` to clear any prior drawing (line 29). This method clears a rectangular area by setting it to the `Canvas`'s background color. The method's four arguments are the *x*- and *y*-coordinates of the rectangle's upper-left corner and the rectangle's width and height. In this case, we clear the entire `Canvas` by starting from its upper-left corner (0, 0) and using its `getWidth` and `getHeight` methods to get the `Canvas`'s width and height, respectively.

Lines 34–46 loop 10 times to draw 10 shapes. The *nested* `switch` statement (lines 36–45) chooses between drawing rectangles and drawing ovals. If method `draw`'s `choice` parameter contains `"rectangles"`, then the method draws rectangles. Lines 38–39 call `GraphicsContext` method `strokeRect`, which requires four arguments. The first two represent the *x*- and *y*-coordinates of the upper-left corner of the rectangle; the next two represent the rectangle's width and height. In this example, we start at a position 10 pixels down and 10 pixels right of the top-left corner, and every iteration of the loop moves the upper-left corner another 10 pixels down and to the right. The width and the height of the rectangle start at 90 pixels and increase by 10 pixels in each iteration.

If method `draw`'s `choice` parameter contains `"ovals"`, the program draws ovals. It creates an imaginary rectangle called a **bounding rectangle** and places inside it an oval that touches the midpoints of all four sides. Method `strokeOval` (lines 42–43) requires the same four arguments as method `strokeRect`. The arguments specify the position and size of the *bounding rectangle* for the oval. The values passed to `strokeOval` in this example are the same as those passed to `strokeRect` in lines 38–39. Since the width and height of the bounding rectangle are identical in this example, lines 42–43 draw a *circle*.

GUI and Graphics Case Study Exercises

5.1 Draw 12 concentric circles in the center of a `Canvas` (Fig. 5.28). The innermost circle should have a radius of 10 pixels, and each successive circle should have a radius 10 pixels larger than the previous one. Begin by finding the `Canvas`'s center. To determine the upper-left corner location of a given circle's bounding rectangle, move up one radius (that is, `-10` pixels) and to the left one radius (that is, `-10` pixels) from the center. The bounding rectangle's width and height are both the same as the circle's diameter (that is, twice the radius).

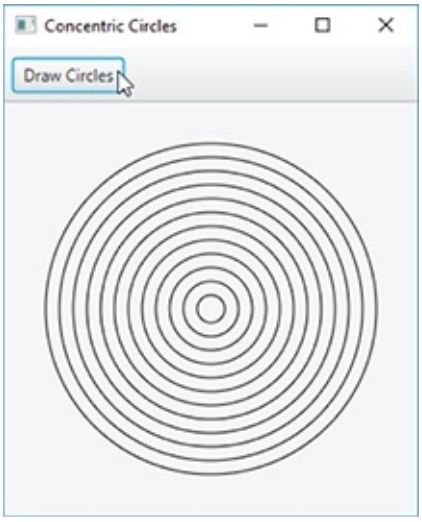

Fig. 5.28

Drawing concentric circles.

5.2 Modify Exercise 5.1 so that it also displays each circle's bounding square (Fig. 5.29).

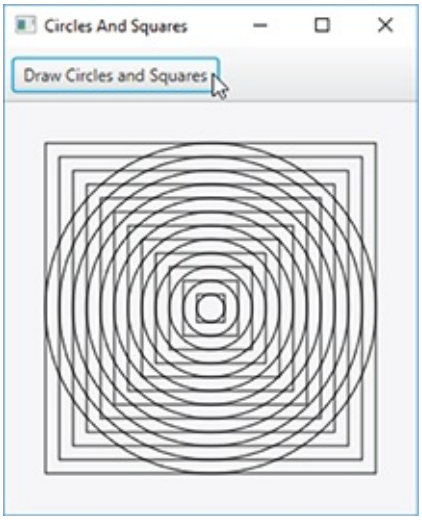

Fig. 5.29

Drawing concentric circles and their bounding squares.

Description