22.10 Drawing on a `Canvas`

So far, you've displayed and manipulated JavaFX two-dimensional shape objects that reside in the scene graph. In this section, we demonstrate similar drawing capabilities using the `javafx.scene.canvas` package, which contains two classes:

- Class `Canvas` is a subclass of `Node` in which you can draw graphics.
- Class `GraphicsContext` performs the drawing operations on a `Canvas`.

As you'll see, a `GraphicsContext` object enables you to specify the same drawing characteristics that you've previously used on `Shape` objects. However, with a `GraphicsContext`, you must set these characteristics and draw the shapes programmatically. To demonstrate various `Canvas` capabilities, Fig. 22.15 re-implements Section 22.3's `BasicShapes` example. Here, you'll see various JavaFX classes and `enum`s (from packages `javafx.scene.image`, `javafx.scene.paint` and `javafx.scene.shape`) that JavaFX's CSS capabilities use behind the scenes to style `Shape`s.

Performance Tip 22.1

A `Canvas` *typically is preferred for performance-oriented graphics, such as those in games with moving elements.*

```
1  // Fig. 22.15: CanvasShapesController.java
2  // Drawing on a Canvas.
3  import javafx.fxml.FXML;
4  import javafx.scene.canvas.Canvas;
5  import javafx.scene.canvas.GraphicsContext;
6  import javafx.scene.image.Image;
7  import javafx.scene.paint.Color;
8  import javafx.scene.paint.CycleMethod;
9  import javafx.scene.paint.ImagePattern;
10 import javafx.scene.paint.LinearGradient;
11 import javafx.scene.paint.RadialGradient;
12 import javafx.scene.paint.Stop;
13 import javafx.scene.shape.ArcType;
14 import javafx.scene.shape.StrokeLineCap;
15
16 public class CanvasShapesController {
17    // instance variables that refer to GUI compo
18    @FXML private Canvas drawingCanvas;
19
20    // draw on the Canvas
21    public void initialize() {
22       GraphicsContext gc = drawingCanvas.getGrap
23       gc.setLineWidth(10); // set all stroke wid
24
25       // draw red line
26       gc.setStroke(Color.RED);
27       gc.strokeLine(10, 10, 100, 100);
28
29       // draw green line
30       gc.setGlobalAlpha(0.5); // half transparen
31       gc.setLineCap(StrokeLineCap.ROUND);
32       gc.setStroke(Color.GREEN);
33       gc.strokeLine(100, 10, 10, 100);
34
35       gc.setGlobalAlpha(1.0); // reset alpha tra
36
```

```
        // draw rounded rect with red border and y
            gc.setStroke(Color.RED);
            gc.setFill(Color.YELLOW);
        gc.fillRoundRect(120, 10, 90, 90, 50, 50);
        gc.strokeRoundRect(120, 10, 90, 90, 50, 50
                    
        // draw circle with blue border and red/wh
            gc.setStroke(Color.BLUE);
               Stop[] stopsRadial =
            {new Stop(0, Color.RED), new Stop(1, Co
        RadialGradient radialGradient = new Radial
            0.6, true, CycleMethod.NO_CYCLE, stopsR
            gc.setFill(radialGradient);
            gc.fillOval(230, 10, 90, 90);
            gc.strokeOval(230, 10, 90, 90);
                    
        // draw ellipse with green border and imag
            gc.setStroke(Color.GREEN);
        gc.setFill(new ImagePattern(new Image("yel
            gc.fillOval(340, 10, 200, 90);
        gc.strokeOval(340, 10, 200, 90);
                    
        // draw arc with purple border and cyan/wh
            gc.setStroke(Color.PURPLE);
               Stop[] stopsLinear =
            {new Stop(0, Color.CYAN), new Stop(1, C
        LinearGradient linearGradient = new Linear
            true, CycleMethod.NO_CYCLE, stopsLinear
            gc.setFill(linearGradient);
        gc.fillArc(560, 10, 90, 90, 45, 270, ArcTy
        gc.strokeArc(560, 10, 90, 90, 45, 270, Arc
                }
            }
```

Fig. 22.15

Drawing on a `Canvas`.

Description

Obtaining the `GraphicsContext`

To draw on a `Canvas`, you first obtain its `GraphicsContext` by calling `Canvas` method `getGraphicsContext2D` (line 22).

Setting the Line Width for All the Shapes

When you set a `GraphicsContext`'s drawing characteristics, they're applied (as appropriate) to all subsequent shapes you draw. For example, line 23 calls

`setLineWidth` to specify the `GraphicsContext`'s line thickness (`10`). All subsequent `GraphicsContext` method calls that draw lines or shape borders will use this setting. This is similar to the `-fx-strokewidth` CSS attribute we specified for all shapes in Fig. 22.4.

Drawing Lines

Lines 26–33 draw the red and green lines:

- `GraphicsContext`'s `setStroke` method (lines 26 and 32) specifies the `Paint` object (package `javafx.scene.paint`) used to draw the line. The `Paint` can be any of the subclasses `Color`, `ImagePattern`, `LinearGradient` or `RadialGradient` (all from package `javafx.scene.paint`). We demonstrate each of these in this example —`Color` for the lines and `Color`, `ImagePattern`, `LinearGradient` or `RadialGradient` as the fills for other shapes.
- `GraphicsContext`'s `strokeLine` method (lines 27 and 33) draws a line using the current `Paint` object that's set as the stroke. The four arguments are the *x*-*y* coordinates of the start and end points, respectively.
- `GraphicsContext`'s `setLineCap` method (line 31) sets line cap, like the CSS property `-fx-stroke-line-cap` in Fig. 22.4. The argument to this method must be constant from the `enum` `StrokeLineCap` (package `javafx.scene.shape`). Here we round the line ends.
- `GraphicsContext`'s `setGlobalAlpha` method (line 30) sets the alpha transparency of all subsequent shapes you draw. For the green line we used `0.5`, which is 50% transparent. After drawing the green line, we reset this to the default `1.0` (line 35), so that subsequent shapes are fully opaque.

Drawing a Rounded Rectangle

Lines 38–41 draw a rounded rectangle with a red border:

- Line 38 sets the border color to `Color.RED`.
- `GraphicsContext`'s `setFill` method (lines 39, 49, 55 and 65) specifies the `Paint` object that fills a shape. Here we fill the rectangle with `Color.YELLOW`.
- `GraphicsContext`'s `fillRoundRect` method draws a *filled* rectangle with rounded corners using the current `Paint` object set as the fill. The method's first four arguments represent the rectangle's upper-left *x*-coordinate, upper-left *y*-coordinate, width and height, respectively. The last two arguments represent the arc width and arc height that are used to round the corners. These work identically to the CSS properties `-fx-arc-width` and `-fx-arc-height` properties in Fig. 22.4. `GraphicsContext` also provides a `fillRect` method that draws a rectangle without rounded corners.
- `GraphicsContext`'s `strokeRoundRect` method has the same arguments as `fillRoundRect`, but draws a hollow rectangle with rounded corners. `GraphicsContext` also provides a `strokeRect` method that draws a rectangle without rounded corners.

Drawing a Circle with a `RadialGradient` Fill

Lines 44–51 draw a circle with a blue border and a red-white, radial-gradient fill. Line 44 sets the border color to `Color.BLUE`. Lines 45–48 configure the `RadialGradient`—these lines perform the same tasks as

the CSS function `radial-gradient` in Fig. 22.4.

First, lines 45–46 create an array of `Stop` objects (package `javafx.scene.paint`) representing the color stops. Each `Stop` has an offset from 0.0 to 1.0 representing the offset (as a percentage) from the gradient's start point and a `Color`. Here the `Stop`s indicate that the radial gradient will transition from red at the gradient's start point to white at its end point.

The `RadialGradient` constructor (lines 47–48) receives as arguments:

- the focus angle, which specifies the direction of the radial gradient's focal point from the gradient's center,
- the distance of the focal point as a percentage (0.0–1.0),
- the center point's *x* and *y* location as percentages (0.0–1.0) of the width and height for the shape being filled,
- a `boolean` indicating whether the gradient should scale to fill its shape,
- a constant from the `CycleMethod` `enum` (package `javafx.scene.paint`) indicating how the color stops are applied, and
- an array of `Stop` objects—this can also be a comma-separated list of `Stop`s or a `List<Stop>` object.

This creates a red-white radial gradient that starts with solid red at the center of the shape and—at 60% of the radial gradient's radius—transitions to white. Line 49 sets the fill to the new `radialGradient`, then lines 50–51 call `GraphicsContext`'s `fillOval` and `strokeOval` methods to draw a filled oval and hollow oval, respectively. Each method receives as arguments the upper-left *x*-

coordinate, upper-left *y*-coordinate, width and height of the rectangular area (that is, the bounding box) in which the oval should be drawn. Because the width and height are the same, these calls draw circles.

Drawing an Oval with an `ImagePattern` Fill

Lines 54–57 draw an oval with a green border and containing an image:

- Line 54 sets the border color to `Color.GREEN`.
- Line 55 sets the fill to an `ImagePattern`—a subclass of `Paint` that loads an `Image`, either from the local system or from a URL specified as a `String`. `ImagePattern` is the class used by the CSS function `image-pattern` in Fig. 22.4.
- Lines 56–57 draw a filled oval and a hollow oval, respectively.

Drawing an Arc with a `LinearGradient` Fill

Lines 60–67 draw an arc with a purple border and filled with a cyan-white linear gradient:

- Line 60 sets the border color to `Color.PURPLE`.
- Lines 61–64 configure the `LinearGradient`, which is the class used by CSS function `linear-gradient` in Fig. 22.4. The constructor's first four arguments are the endpoint coordinates that represent the

direction and angle of the gradient—if the *x*-coordinates are the same, the gradient is vertical; if the *y*-coordinates are the same, the gradient is horizontal and all other linear gradients follow a diagonal line. When these values are specified in the range 0.0 to 1.0 and the constructor's fifth argument is `true`, the gradient is scaled to fill the shape. The next argument is the `CycleMethod`. The last argument is an array of `Stop` objects—again, this can be a comma-separated list of `Stop`s or a `List<Stop>` object.

Lines 66–67 call `GraphicsContext`'s `fillArc` and `strokeArc` methods to draw a filled arc and hollow arc, respectively. Each method receives as arguments

- the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height of the rectangular area (that is, the bounding box) in which the oval should be drawn,
- the start angle and sweep of the arc in degrees, and
- a constant from the `ArcType` `enum` (package `javafx.scene.shape`)

Additional `GraphicsContext` Features

There are many additional `GraphicsContext` features, which you can explore at

```
https://docs.oracle.com/javase/8/javafx/api/javafx/sc
```

Some of the capabilities that we did not discuss here include:

- Drawing and filling text—similar to the font features in Section 22.2.
- Drawing and filling polylines, polygons and paths—similar to the corresponding `Shape` subclasses in Section 22.4.
- Applying effects and transforms—similar to the transforms in Section 22.5.
- Drawing images.
- Manipulating the individual pixels of a drawing in a `Canvas` via a `PixelWriter`.
- Saving and restoring graphics characteristics via the `save` and `restore` methods.