Chapter 22
JavaFX Graphics and Multimedia

Java How to Program, 11/e

Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Use JavaFX graphics and multimedia capabilities to make your apps "come alive" with graphics, animations, audio and video.
- Use external Cascading Style Sheets to customize the look of **Node**s while maintaining their functionality.

OBJECTIVES (cont.)

- Customize fonts attributes such as font family, size and style.
- Display two-dimensional shape nodes of types `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Path`, `Polyline` and `Polygon`.
- Customize the stroke and fill of shapes with solid colors, images and gradients.
- Use `Transforms` to reposition and reorient nodes.

OBJECTIVES (cont.)

- Display and control video playback with `Media`, `MediaPlayer` and `MediaView`.
- Animate `Node` properties with `Transition` and `Timeline` animations.
- Use an `AnimationTimer` to create frame-by-frame animations.
- Draw graphics on a `Canvas` node.
- Display 3D shapes.

22.1 Introduction

22.2 Controlling Fonts with Cascading Style Sheets (CSS)

22.2.1 CSS That Styles the GUI
22.2.2 FXML That Defines the GUI—Introduction to XML Markup
22.2.3 Referencing the CSS File from FXML
22.2.4 Specifying the **VBox**'s Style Class
22.2.5 Programmatically Loading CSS

22.3 Displaying Two-Dimensional Shapes

22.3.1 Defining Two-Dimensional Shapes with FXML
22.3.2 CSS That Styles the Two-Dimensional Shapes

22.4 `Polylines`, `Polygons` and `Paths`

22.4.1 GUI and CSS
22.4.2 `PolyShapesController` Class

22.5 Transforms

22.6 Playing Video with `Media`, `MediaPlayer` and `MediaViewer`

- 22.6.1 `VideoPlayer` GUI
- 22.6.2 `VideoPlayerController` Class

22.7 `Transition` Animations

- 22.7.1 `TransitionAnimations.fxml`
- 22.7.2 `TransitionAnimations-Controller` Class

22.8 `Timeline` Animations
22.9 Frame-by-Frame Animation with `AnimationTimer`
22.10 Drawing on a `Canvas`
22.11 Three-Dimensional Shapes
22.12 Wrap-Up

22.1 Introduction

- Use external Cascading Style Sheets (CSS) to customize the appearance of JavaFX nodes.
- Customize fonts and font attributes used to display text.
- Display two-dimensional shapes, including lines, rectangles, circles, ellipses, arcs, polylines, polygons and custom paths
- Apply transforms to `Node`s, such as rotating a `Node` around a particular point, scaling, translating (moving) and more.

22.1 Introduction

- Display video and control its playback (e.g., play, pause, stop, and skip to specific time).
- Animate JavaFX `Nodes` with `Transition` and `Timeline` animations that change `Node` property values over time. Create frame-by-frame animations with an `AnimationTimer`.
- Draw two-dimensional graphics on a `Canvas Node`.
- Display three-dimensional shapes, including boxes, cylinders and spheres.

22.2 Controlling Fonts with Cascading Style Sheets (CSS)

- In this chapter, we format JavaFX objects using a technology called Cascading Style Sheets (CSS) that's typically used to style the elements in web pages
- CSS allows you to specify *presentation* (e.g., fonts, spacing, sizes, colors, positioning) separately from the GUI's *structure* and *content* (layout containers, shapes, text, GUI components, etc.)
- If a JavaFX GUI's presentation is determined entirely by CSS rules, you can simply swap in a new style sheet to change the GUI's appearance

22.2 Controlling Fonts with Cascading Style Sheets (CSS) (cont.)

- In this section, you'll use CSS to specify the font properties of several `Labels` and the spacing and padding properties for the `VBox` layout that contains the `Labels`
- You'll place CSS rules that specify the font properties, spacing and padding in a separate file that ends with the `.css` filename extension, then reference that file from the FXML

22.2 Controlling Fonts with Cascading Style Sheets (CSS) (cont.)

- As you'll see,
  - before referencing the CSS file from the FXML, Scene Builder displays the GUI without styling, and
  - after referencing the CSS file from the FXML, Scene Builder renders the GUI with the CSS rules applied to the appropriate objects.
- For a complete JavaFX CSS reference visit:
  - `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html`

22.2.1 CSS That Styles the GUI

- Figure 22.1 presents this app's CSS rules that specify the `VBox`'s and each `Label`'s style
- This file is located in the same folder as the rest of the example's files.

```
1  /* Fig. 22.1: FontsCSS.css */
2  /* CSS rules that style the VBox and Labels */
3
4  .vbox {
5     -fx-spacing: 10;
6     -fx-padding: 10;
7  }
8
9  #label1 {
10    -fx-font: bold 14pt Arial;
11 }
12
13 #label2 {
14    -fx-font: 16pt "Times New Roman";
15 }
16
17 #label3 {
18    -fx-font: bold italic 16pt "Courier New";
19 }
```

Fig. 22.1 | CSS rules that style the VBox and Labels. (Part 1 of 2.)

```
20
21   #label4 {
22      -fx-font-size: 14pt;
23      -fx-underline: true;
24   }
25
26   #label5 {
27      -fx-font-size: 14pt;
28   }
29
30   #label5 .text {
31      -fx-strikethrough: true;
32   }
```

Fig. 22.1 | CSS rules that style the VBox and Labels. (Part 2 of 2.)

22.2.1 CSS That Styles the GUI (cont.)

`#label1` CSS Rule—ID Selectors

- Lines 9–11 define the `#label1` CSS rule. Selectors that begin with # are known as ID selectors-—they are applied to objects with the specified ID
- In this case, the `#label1` selector- matches the object with the `fx:id label1`-—that is, the `Label` object in line 12 of Fig. 22.2
- The `#label1` CSS rule specifies the CSS property
  - `-fx-font: bold 14pt Arial;`
- This rule sets an object's `font` property
- The object to which this rule applies displays its text in a bold, 14-point, Arial font

22.2.1 CSS That Styles the GUI (cont.)

.vbox CSS Rule—Style Class Selectors

- Lines 4–7 define the `.vbox` CSS rule that will be applied to this app's `VBox` object (lines 8–18 of Fig. 22.2)
- Each CSS rule begins with a CSS selector which specifies the JavaFX objects that will be styled according to the rule
- In the `.vbox` CSS rule, `.vbox` is a style class selector that's applied to any JavaFX object that has a `styleClass` property with the value `"vbox"`
- In CSS, a style class selector begins with a dot (`.`) and is followed by its class name (not to be confused with a Java class)
- By convention, selector names typically have all lowercase letters, and multi-word names separate each word from the next with a dash (`-`).
- Each CSS rule's body is delimited by a set of required braces (`{}`) containing the CSS properties that are applied to objects matching the CSS selector

22.2.1 CSS That Styles the GUI (cont.)

- Each JavaFX CSS property name begins with `-fx-\` followed by the name of the corresponding JavaFX object's property in all lowercase letters
- So, `-fx-spacing` in line 5 of Fig. 22.1 defines the value for a JavaFX object's `spacing` property, and `-fx-padding` in line 6 defines the value for a JavaFX object's `padding` property
- The value of each property is specified to the right of the required colon (`:`)
- In this case, we set `-fx-spacing` to `10` to place 10 pixels of vertical space between objects in the `VBox`, and `-fx-padding` to `10` to separate the `VBox`'s contents from the `VBox`'s edges by 10 pixels at the top, right, bottom and left edges
- You also can specify the `-fx-padding` with four values separated by spaces

22.2.1 CSS That Styles the GUI (cont.)

- The `-fx-font` property can specify all aspects of a font, including its style, weight, size and font family—the size and font family are required
- There are also properties for setting each font component: `-fx-font-style`, `-fx-font-weight`, `-fx-font-size` and `-fx-font-family`
- These are applied to a JavaFX object's similarly named properties
- For more information on specifying CSS font attributes, see
  - `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html#typefont`
- For a complete list of CSS selector types and how you can combine them, see
  - `https://www.w3.org/TR/css3-selectors/`

22.2.1 CSS That Styles the GUI (cont.)

*`#label2` **CSS Rule***

- Lines 13–15 define the `#label2` CSS rule that will be applied to the `Label` with the `fx:id` `label2`
- The CSS property
  - `-fx-font: 16pt "Times New Roman";`
- specifies only the required font size (`16pt`) and font family (`"Times New Roman"`) components—font family names with multiple words must be enclosed in double quotes.

22.2.1 CSS That Styles the GUI (cont.)

#label3 CSS Rule

- Lines 17–19 define the `#label3` CSS rule that will be applied to the `Label` with the `fx:id` `label3`
- `-fx-font: bold italic 16pt "Courier New";`
  - specifies all the font components—weight (`bold`), style (`italic`), size (`16pt`) and font family (`"Courier New"`).

22.2.1 CSS That Styles the GUI (cont.)

- ***#label4 CSS Rule***
- Lines 21–24 define the `#label4` CSS rule that will be applied to the `Label` with the `fx:id label4`
- `-fx-font-size: 14pt;`
  - specifies the font size `14pt`—all other aspects of this `Label`'s font are inherited from the `Label`'s parent container
- `-fx-underline: true;`
  - indicates that the text in the `Label` should be *underlined*—the default value for this property is `false`.

22.2.1 CSS That Styles the GUI (cont.)

`#label5` CSS Rule

- Lines 26–28 define the `#label5` CSS rule that will be applied to the `Label` with the `fx:id` `label5`
- `-fx-font-size: 14pt;`
  - specifies the font size `14pt`.

22.2.1 CSS That Styles the GUI (cont.)

*`#label5 .text` **CSS Rule***

- Lines 30–32 define the `#label5 .text` CSS rule that will be applied to the `Text` object within the `Label` that has the `fx:id` value `"label5"`
- The selector in this case is a combination of an ID selector and a style class selector
- Each `Label` contains a `Text` object with the CSS class `.text`
- When applying this CSS rule, JavaFX first locates the object with the ID `label5`, then within that object looks for a nested object that specifies the class `text`.
- `-fx-strikethrough: true;`
  - indicates that the text in the `Label` should be displayed with a line through it—the default value for this property is `false`

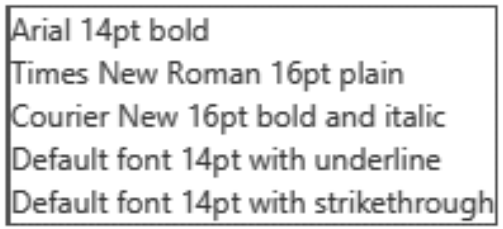
22.2.2 FXML That Defines the GUI—Introduction to XML Markup

- Figure 22.2 shows the contents of `FontCSS.fxml`—the `FontCSS` app's FXML GUI, which consists of a `VBox` layout element (lines 8–18) containing five `Label` elements (lines 12–16)
- When you first drag five `Label`s onto the `VBox` and configure their text (Fig. 22.2(a)), all the `Label`s initially have the same appearance in Scene Builder
- Also, initially there's no spacing between and around the `Label`s in the `VBox`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 22.2: FontCSS.fxml -->
3  <!-- FontCSS GUI that is styled via external CSS -->
4
5  <?import javafx.scene.control.Label?>
6  <?import javafx.scene.layout.VBox?>
7
8  <VBox styleClass="vbox" stylesheets="@FontCSS.css"
9     xmlns="http://javafx.com/javafx/8.0.60"
10    xmlns:fx="http://javafx.com/fxml/1">
11    <children>
12       <Label fx:id="label1" text="Arial 14pt bold" />
13       <Label fx:id="label2" text="Times New Roman 16pt plain" />
14       <Label fx:id="label3" text="Courier New 16pt bold and italic" />
15       <Label fx:id="label4" text="Default font 14pt with underline" />
16       <Label fx:id="label5" text="Default font 14pt with strikethrough" />
17    </children>
18 </VBox>
```

Fig. 22.2 | FontCSS GUI that is styled via external CSS. (Part 1 of 3.)

a) GUI as it appears in Scene Builder *before* referencing the completed CSS file



b) GUI as it appears in Scene Builder *after* referencing the `FontCSS.css` file containing the rules that style the `VBox` and the `Label`s

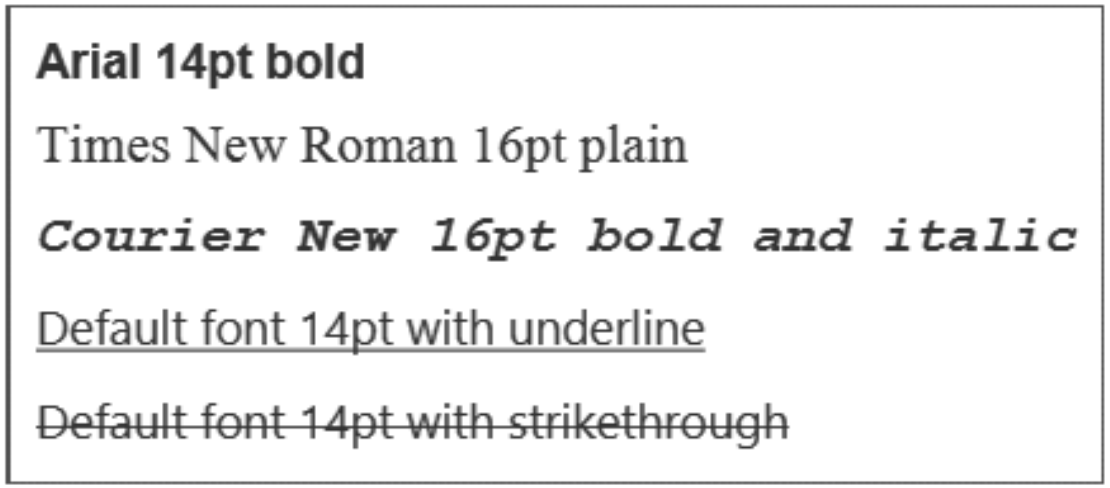


Fig. 22.2 | **FontCSS** GUI that is styled via external CSS. (Part 2 of 3.)

c) GUI as it appears in the *running* application

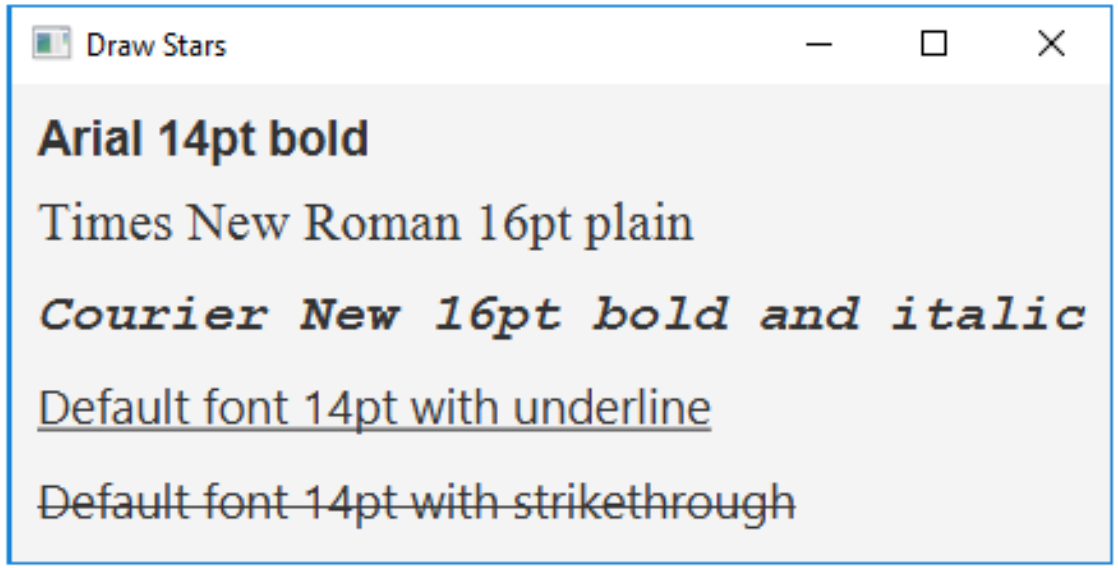


Fig. 22.2 | **FontCSS** GUI that is styled via external CSS. (Part 3 of 3.)

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

XML Declaration

- Each FXML document begins with an XML declaration (line 1), which must be the first line in the file and indicates that the document contains XML markup
- For FXML documents, line 1 must appear as shown in Fig. 22.2
- The XML declaration's `version` attribute specifies the XML syntax version (`1.0`) used in the document
- The `encoding-` attribute specifies the format of the document's character—XML documents typically contain Unicode characters in `UTF-8` format (`https://en.wikipedia.org/wiki/UTF-8`)

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

Attributes

- Each XML attribute has the format
  - *name*="*value*"
- The *name* and *value* are separated by = and the *value* placed in quotation marks (" ")
- Multiple *name*=*value* pairs are separated by whitespace

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

Comments

- Lines 2–3 are XML comments, which begin with <!-- and end with -->, and can be placed almost anywhere in an XML document
- XML comments can span to multiple lines

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

FXML `import` Declarations

- Lines 5–6 are FXML `import` declarations that specify the fully qualified names of the JavaFX- types used in the document
- Such declarations are delimited by `<?import` and `?>`

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

Elements

- XML documents contain elements that specify the document's structure
- Most elements are delimited by a start tag and an end tag:
  - A start tag consists of angle brackets (< and >) containing the element's name followed by zero or more attributes. For example, the `VBox` element's start tag (lines 8–10) contains four attributes.
  - An end tag consists of the element name preceded by a forward slash (/) in angle brackets—for example, `</VBox>` in line 18.
- An element's start and end tags enclose the element's contents
- In this case, lines 11–17 declare other elements that describe the `VBox`'s contents

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

- Every XML document must have exactly one root element that contains all the other elements
- In Fig. 22.2, `VBox` is the root.
- A layout element always contains a `children` element (lines 11–17) containing the child `Nodes` that are arranged by that layout
- For a `VBox`, the `children` element contains the child `Nodes` in the order they're displayed on the screen from top to bottom

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

- The elements in lines 12–16 represent the `VBox`'s five `Labels`
- These are empty elements that use the shorthand start-tag-only notation:
  - `<`*ElementName attributes* `/>`
- in which the empty element's start tag ends with `/>` rather than `>`
- The empty element:
  - `<Label fx:id="label1" text="Arial 14pt bold" />`
- is equivalent to
  - `<Label fx:id="label1" text="Arial 14pt bold">`
    `</Label>`
- which does not have content between the start and end tags
- Empty elements often have attributes (such as `fx:id` and `text` for each `Label` element)

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

XML Namespaces

- In lines 9–10, the VBox attributes
  - `xmlns="http://javafx.com/javafx/8.0.60"`
  - `xmlns:fx="http://javafx.com/fxml/1"`
- specify the XML namespaces used in FXML markup
- An XML namespace specifies a collection of element and attribute names that you can use in the document
- The attribute
  - `xmlns="http://javafx.com/javafx/8.0.60"`
- specifies the default namespace

22.2.2 FXML That Defines the GUI—Introduction to XML Markup (cont.)

- FXML `import` declarations (like those in lines 5–6) add names to this namespace for use in the document
- The attribute
  - `xmlns:fx="http://javafx.com/fxml/1"`
- specifies JavaFX's `fx` namespace
- Elements and attributes from this namespace (such as the `fx:id` attribute) are used internally by the `FXMLLoader` class
- The `fx:` in `fx:id` is a namespace prefix that specifies the namespace (`fx`) that defines the attribute (`id`)
- Every element or attribute name in Fig. 22.2 that does not begin with `fx:` is part of the default namespace.

22.2.3 Referencing the CSS File from FXML

- For the `Labels` to appear with the fonts shown in Fig. 22.2(b), we must reference the `FontCSS.css` file from the FXML
  - This enables Scene Builder to apply the CSS rules to the GUI
- To reference the CSS file:
  - Select the `VBox` in the Scene Builder.
  - In the Properties inspector, click the + button under the Stylesheets heading.
  - In the dialog that appears, select the `FontCSS.css` file and click Open.
- Adds the `stylesheets` attribute (line 8) to the `VBox`'s opening tag
  - `stylesheets="@FontCSS.css"`
- The @ symbol—called the local resolution operator in FXML—indicates that the file `FontCSS.css` is located relative to the FXML file on disk
- No path information is specified here, so the CSS file and the FXML file must be in the same folder

22.2.4 Specifying the VBox's Style Class

- The preceding steps apply the font styles to the `Label`s, based on their ID selectors, but do not apply the spacing and padding to the `VBox`
- Recall that for the `VBox` we defined a CSS rule using a *style class selector* with the name `.vbox`
- To apply the CSS rule to the `VBox`:
  - Select the `VBox` in the Scene Builder.
  - In the Properties inspector, under the Style Class heading, specify the value `vbox` *without* the dot, then press *Enter* to complete the setting.
- This adds the `styleClass` attribute to the `VBox`'s opening tag (line 8)
  - `styleClass="vbox"`
- At this point the GUI appears as in Fig. 22.2(b)
- You can now run the app to see the output in Fig. 22.2(c)

22.2.5 Programmatically Loading CSS

- In the `FontCSS` app, the FXML referenced the CSS style sheet directly (line 8)
- It's also possible to load CSS files dynamically and add them to a `Scene`'s collection of style sheets
- You might do this, for example, in an app that enables users to choose their preferred look-and-feel, such as a light background with dark text vs. a dark background with light text.
- To load a stylesheet dynamically, add the following statement to the `Application` subclass's `start` method:
  - ```
    scene.getStylesheets().add(
       getClass().getResource("FontCSS.css").toExternalForm());
    ```
    ```

22.2.5 Programmatically Loading CSS (cont.)

- In the preceding statement:
  - Inherited `Object` method `getClass` obtains a `Class` object representing the app's `Application` subclass.
  - `Class` method `getResource` returns a `URL` representing the location of the file `FontCSS.css`. Method `getResource` looks for the file in the same location from which the `Application` subclass was loaded.
  - `URL` method `toExternalForm` returns the `URL`'s `String` representation. This is passed to the `add` method of the `Scene`'s collection of style sheets—this adds the style sheet to the scene.

22.3 Displaying Two-Dimensional Shapes

- JavaFX has two ways to draw shapes:
  - You can define `Shape` and `Shape3D` (package `javafx.scene.shape`) subclass objects, add them to a container in the JavaFX stage and manipulate them like other JavaFX `Node`s.
  - You can add a `Canvas` object (package `javafx.scene.canvas`) to a container in the JavaFX stage, then draw on it using various `GraphicsContext` methods.
- The `BasicShapes` example presented in this section shows you how to display two-dimensional `Shape`s of types `Line`, `Rectangle`, `Circle`, `Ellipse` and `Arc`
- Like other `Node` types, you can drag shapes from the Scene Builder Library's Shapes category onto the design area, then configure them via the Inspector's Properties, Layout and Code sections—of course, you also may create objects of any JavaFX `Node` type programmatically

22.3.1 Defining Two-Dimensional Shapes with FXML

- Figure 22.3 shows the completed FXML for the `BasicShapes` app, which references the `BasicShapes.css` file (line 13) that we present in Section 22.3.2
- For this app we dragged two `Lines`, a `Rectangle`, a `Circle`, an `Ellipse` and an `Arc` onto a `Pane` layout and configured their dimensions and positions in Scene Builder

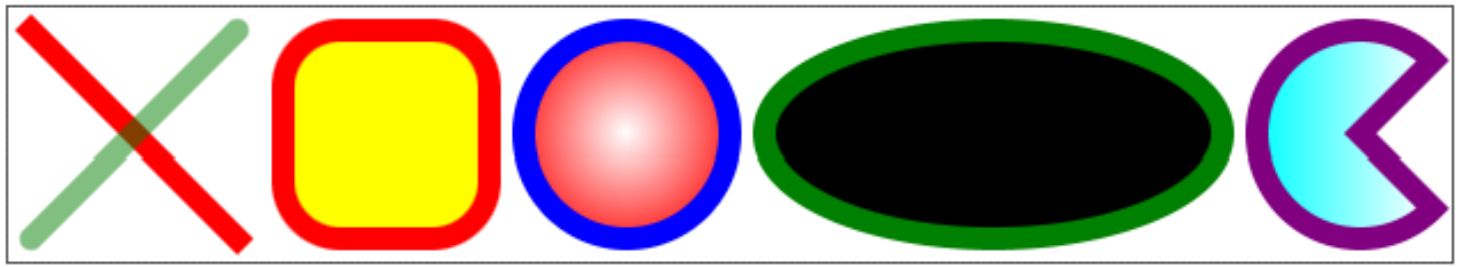
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 22.3: BasicShapes.fxml -->
3  <!-- Defining Shape objects and styling via CSS -->
4
5  <?import javafx.scene.layout.Pane?>
6  <?import javafx.scene.shape.Arc?>
7  <?import javafx.scene.shape.Circle?>
8  <?import javafx.scene.shape.Ellipse?>
9  <?import javafx.scene.shape.Line?>
10 <?import javafx.scene.shape.Rectangle?>
11
12 <Pane id="Pane" prefHeight="110.0" prefWidth="630.0"
13    stylesheets="@BasicShapes.css" xmlns="http://javafx.com/javafx/8.0.60"
14    xmlns:fx="http://javafx.com/fxml/1">
```

Fig. 22.3 | Defining Shape objects and styling via CSS. (Part 1 of 3.)

```
15      <children>
16         <Line fx:id="line1" endX="100.0" endY="100.0"
17            startX="10.0" startY="10.0" />
18         <Line fx:id="line2" endX="10.0" endY="100.0"
19            startX="100.0" startY="10.0" />
20         <Rectangle fx:id="rectangle" height="90.0" layoutX="120.0"
21            layoutY="10.0" width="90.0" />
22         <Circle fx:id="circle" centerX="270.0" centerY="55.0"
23            radius="45.0" />
24         <Ellipse fx:id="ellipse" centerX="430.0" centerY="55.0"
25            radiusX="100.0" radiusY="45.0" />
26         <Arc fx:id="arc" centerX="590.0" centerY="55.0" length="270.0"
27            radiusX="45.0" radiusY="45.0" startAngle="45.0" type="ROUND" />
28      </children>
29   </Pane>
```

Fig. 22.3 | Defining Shape objects and styling via CSS. (Part 2 of 3.)

a) GUI in Scene Builder with CSS applied—Ellipse's image fill does not show.



b) GUI in running app—Ellipse's image fill displays correctly.

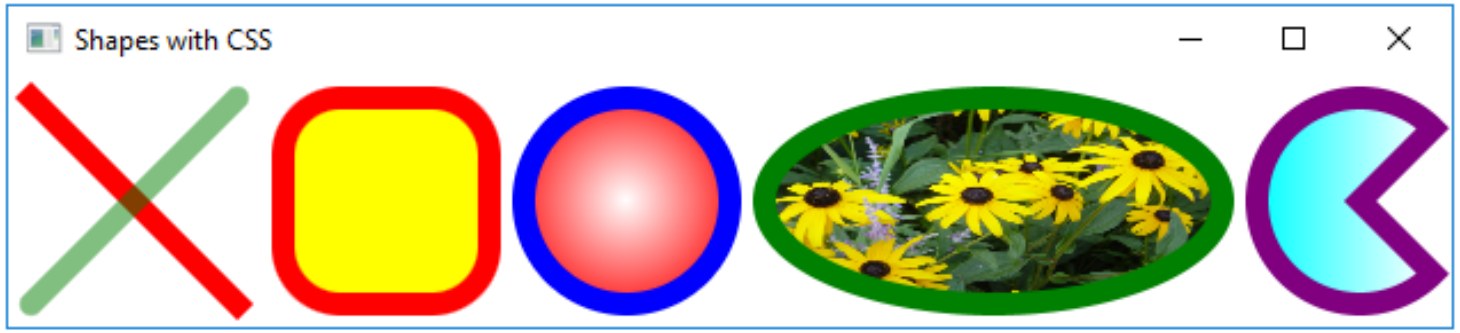


Fig. 22.3 | Defining **Shape** objects and styling via CSS. (Part 3 of 3.)

22.3.1 Defining Two-Dimensional Shapes with FXML (cont.)

- For each property you can set in Scene Builder, there is a corresponding attribute in FXML
  - For example, the `Pane` object's `Pref Height` property in Scene Builder corresponds to the `prefHeight` attribute (line 12) in FXML
- When you build this GUI in Scene Builder, use the FXML attribute values shown in Fig. 22.3
- Note that as you drag each shape onto your design, Scene Builder automatically configures certain properties, such as the `Fill` and `Stroke` colors for the `Rectangle`, `Circle`, `Ellipse` and `Arc`
- For each such property that does not have a corresponding attribute shown in Fig. 22.3, you can remove the attribute either by setting the property to its default value in Scene Builder or by manually editing the FXML

22.3.1 Defining Two-Dimensional Shapes with FXML (cont.)

- Lines 6–10 import the shape classes used in the FXML
- We specified `fx:id` values (lines 16 and 18) for the two `Lines`—used in CSS rules with ID selectors to define separate styles for each `Line`
- We removed the shapes' `fill`, `stroke` and `strokeType` properties that Scene Builder autogenerated
- The default `fill` for a shape is black
- The default stroke is a one-pixel black line
- The default `strokeType` is centered—based on the stroke's thickness, half the thickness appears inside the shape's bounds and half outside
- You also may display a shape's stroke completely inside or outside the shape's bounds
- We specify the strokes and fills with the styles in Section 22.3.2

22.3.1 Defining Two-Dimensional Shapes with FXML (cont.)

`Line` *Objects*

- Lines 16–17 and 18–19 define two `Lines`
- Each connects two endpoints specified by the properties `startX`, `startY`, `endX` and `endY`
- The *x*- and *y*-coordinates are measured from the top-left corner of the `Pane`, with *x*-coordinates- increasing left to right and *y*-coordinates increasing top to bottom
- If you specify a `Line`'s `layoutX` and `layoutY` properties, then the `startX`, `startY`, `endX` and `endY` properties are measured from that point

22.3.1 Defining Two-Dimensional Shapes with FXML (cont.)

`Rectangle` Object

- Lines 20–21 define a `Rectangle` object
- A `Rectangle` is displayed based on its `layoutX`, `layoutY`, `width` and `height` properties:
  - A `Rectangle`'s upper-left corner is positioned at the coordinates specified by the `layoutX` and `layoutY` properties, which are inherited from class `Node`.
  - A `Rectangle`'s dimensions are specified by the `width` and `height` properties—in this case they have the same value, so the `Rectangle` defines a square.

22.3.1 Defining Two-Dimensional Shapes with FXML (cont.)

`Circle` *Object*

- Lines 22–23 define a `Circle` object with its center at the point specified by the `centerX` and `centerY` properties
- The `radius` property determines the `Circle`'s size (two times the `radius`) around its center point

22.3.1 Defining Two-Dimensional Shapes with FXML (cont.)

`Ellipse` Object

- Lines 24–25 define an `Ellipse` object
- Like a `Circle`, an `Ellipse`'s center is specified by the `centerX` and `centerY` properties
- You also specify `radiusX` and `radiusY` properties that help determine the `Ellipse`'s width (left and right of the center point) and height (above and below the center point).

22.3.1 Defining Two-Dimensional Shapes with FXML (cont.)

`Arc` Object

- Lines 26–27 define an `Arc` object
- Like an `Ellipse`, an `Arc`'s center is specified by the `centerX` and `centerY` properties, and the `radiusX` and `radiusY` properties determine the `Arc`'s width and height
- For an `Arc`, you also specify:
  - `length`—The arc's length in degrees (0–360). Positive values sweep counter-clockwise.
  - `startAngle`—The angle in degrees at which the arc should begin.
  - `type`—How the arc should be closed. `ROUND` indicates that the starting and ending points of the arc should be connected to the center point by straight lines. You also may choose `OPEN`, which does not connect the start and end points, or `CHORD`, which connects the start and end points with a straight line.

22.3.2 CSS That Styles the Two-Dimensional Shapes

- Figure 22.4 shows the CSS for the `BasicShapes` app
- In this CSS file, we define two CSS rules with ID selectors (`#line1` and `#line2`) to style the app's two `Line` objects
- The remaining rules use type selectors, which apply to all objects of a given type
- You specify a type selector by using the JavaFX class name

```
/* Fig. 22.4: BasicShapes.css */
/* CSS that styles various two-dimensional shapes */

Line, Rectangle, Circle, Ellipse, Arc {
    -fx-stroke-width: 10;
}

#line1 {
    -fx-stroke: red;
}

#line2 {
    -fx-stroke: rgba(0%, 50%, 0%, 0.5);
    -fx-stroke-line-cap: round;
}

```

Fig. 22.4 | CSS that styles various two-dimensional shapes. (Part 1 of 2.)

```
17  Rectangle {
18     -fx-stroke: red;
19     -fx-arc-width: 50;
20     -fx-arc-height: 50;
21     -fx-fill: yellow;
22  }
23
24  Circle {
25     -fx-stroke: blue;
26     -fx-fill: radial-gradient(center 50% 50%, radius 60%, white, red);
27  }
28
29  Ellipse {
30     -fx-stroke: green;
31     -fx-fill: image-pattern("yellowflowers.png");
32  }
33
34  Arc {
35     -fx-stroke: purple;
36     -fx-fill: linear-gradient(to right, cyan, white);
37  }
```

Fig. 22.4 | CSS that styles various two-dimensional shapes. (Part 2 of 2.)

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

Specifying Common Attributes for Various Objects

- The CSS rule in lines 4–6 defines the `-fx-stroke-width` CSS property for all the shapes in the app—this property specifies the thickness of the `Lines` and the border thickness of all the other shapes
- To apply this rule to multiple shapes we use CSS type selectors in a comma-separated list
- So, line 4 indicates that the rule in lines 4–6 should be applied to all objects of types `Line`, `Rectangle`, `Circle`, `Ellipse` and `Arc` in the GUI.

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

Styling the `Lines`

- The CSS rule in lines 8–10 sets the `-fx-stroke` to the solid color `red`
- This rule applies to the `Line` with the `fx:id "line1"`
- This rule is in addition to the rule at lines 4–6, which sets the stroke width for all `Lines` (and all the other shapes)
- When JavaFX renders an object, it combines all the CSS rules that apply to the object to determine its appearance
- This rule applies to the `Line` with the `fx:id "line1"`
- For details on all the ways to specify color in CSS, see
  - `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html#typecolor`

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

- The CSS rule in lines 12–15 applies to the `Line` with the `fx:id` `"line2"`
- For this rule, we specified the `-fx-stroke` property's color using the CSS function `rgba`, which defines a color based on its red, green, blue and alpha (transparency) components
- Here we used the version of `rgba` that receives percentages from 0% to 100% specifying the amount of red, green and blue in the color, and a value from 0.0 (transparent) to 1.0 (opaque) for the alpha component
- Line 13 produces a semitransparent green line
- You can see the interaction between the two `Lines`' colors at the intersection point in Fig. 22.3's output windows
- The `-fx-stroke-line-cap` CSS property (line 14) indicates that the ends of the `Line` should be *rounded*—the rounding effect becomes more noticeable with thicker strokes.

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

Styling the *`Rectangle`*

- For `Rectangles`, `Circles`, `Ellipses` and `Arcs` you can specify both the `-fx-stroke` for the shapes' borders and the `-fx-fill`, which specifies the color or pattern that appears inside the shape
- The rule in lines 17–22 uses a CSS type selector to indicate that all `Rectangles` should have `red` borders (line 18) and `yellow` fill (line 21)
- Lines 19–20 define the `Rectangle`'s `-fx-arc-width` and `-fx-arc-height` properties, which specify the width and height of an ellipse that's divided in half horizontally and vertically, then used to round the `Rectangle`'s corners
- Because these properties have the same value (`50`) in this app, the four corners are each one quarter of a circle with a diameter of 50

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

Styling the `Circle`

- The CSS rule at lines 24–27 applies to all `Circle` objects
- Line 25 sets the `Circle`'s stroke to `blue`
- Line 26 sets the `Circle`'s fill with a gradient—colors that transition gradually from one color to the next
- You can transition between as many colors as you like and specify the points at which to change colors, called color stops
- You can use gradients for any property that specifies a color
- In this case, we use the CSS function `radial-gradient` in which the color changes gradually from a center point outward

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

- `-fx-fill: radial-gradient(center 50% 50%, radius 60%, white, red);`
  - indicates that the gradient should begin from a `center` point at `50% 50%`—the middle of the shape horizontally and the middle of the shape vertically
- The `radius` specifies the distance from the center at which an even mixture of the two colors appears
- This radial gradient begins with the color `white` in the center and ends with `red` at the outer edge of the `Circle`'s fill
- We'll discuss a linear gradient momentarily

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

Styling the `Ellipse`

- The CSS rule at lines 29–32 applies to all `Ellipse` objects
- Line 30 specifies that an `Ellipse-` should have a `green` stroke
- Line 31 specifies that the `Ellipse`'s fill should be the image in the file `yellowflowers.png`, which is located in this app's folder
- This image is provided in the `images` folder with the chapter's examples—if you're building this app from scratch, copy the video into the app's folder on your system
- To specify an image as fill, you use the CSS function `image-pattern`
  - [*Note:* At the time of this writing, Scene Builder does not display a shape's fill correctly if it's specified with a CSS `image-pattern`. You must run the example to see the fill, as shown in Fig. 22.3(b).]

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

Styling the `Arc`

- The CSS rule at lines 34–37 applies to all `Arc` objects
- Line 35 specifies that an `Arc` should have a `purple` stroke
- `-fx-fill: linear-gradient(to right, cyan, white);`
  - specifies that the `Arc` should be filled with a linear gradient—such gradients gradually transition from one color to the next horizontally, vertically or diagonally
- You can transition between as many colors as you like and specify the points at which to change colors

22.3.2 CSS That Styles the Two-Dimensional Shapes (cont.)

- To create a linear gradient, you use the CSS function `linear-gradient`
- In this case, `to right` indicates that the gradient should start from the shape's left edge and transition through colors to the shape's right edge
- We specified only two colors here—`cyan` at the left edge of the gradient and `white` at the right edge—but two or more colors can be specified in the comma-separated list
- For more information on all the options for configuring radial gradients, linear gradients and image patterns, see
  - `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html#typepaint`

22.4 Polylines, Polygons and Paths

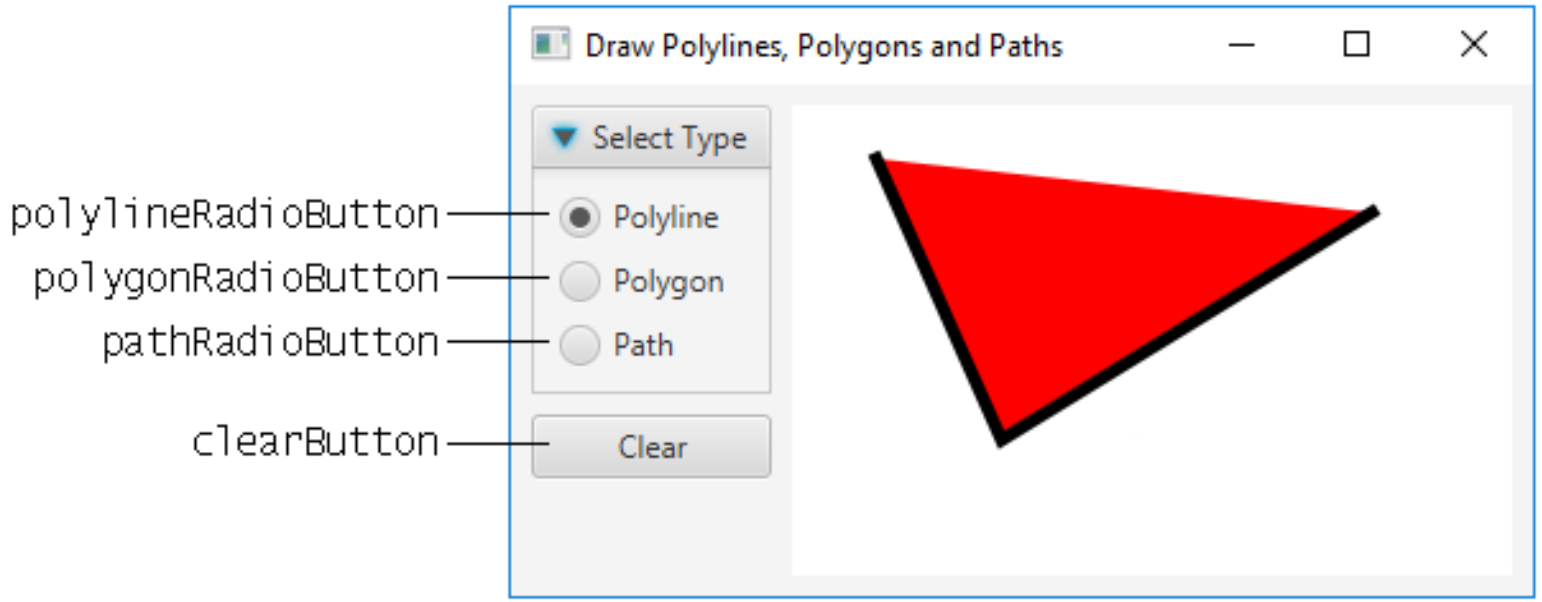
- There are several kinds of JavaFX shapes that enable you to create custom shapes:
  - `Polyline`—draws a series of connected lines defined by a set of points.
  - `Polygon`—draws a series of connected lines defined by a set of points and connects the last point to the first point.
  - `Path`—draws a series of connected `PathElements` by moving to a given point, then drawing lines, arcs and curves.
- In the `PolyShapes` app, you select which shape you want to display by selecting one of the `RadioButtons` in the left column
- You specify a shape's points by clicking throughout the `AnchoredPane` in which the shapes are displayed.
- For this example, we do not show the `PolyShapes` subclass of `Application` (located in the example's `PolyShapes.java` file), because it loads the FXML and displays the GUI, as demonstrated in Chapters 12 and 13.

22.4.1 GUI and CSS

- This app's GUI (Fig. 22.5) is similar to that of the `Painter` app in Section 13.3
  - The three `RadioButtons` are part of a `ToggleGroup` with the `fx:id` `"toggleGroup"`. The Polyline `RadioButton` should be Selected by default. We also set each `Radio-Button`'s On Action event handler to `shapeRadioButtonSelected`.
  - We dragged a `Polyline`, a `Polygon` and a `Path` from the Scene Builder Library's Shapes section onto the `Pane` that displays the shapes, and we set their `fx:id`s to `polyline`, `polygon` and `path`, respectively. We set each shape's `visible` property to `false` by selecting the shape in Scene Builder, then unchecking the Visible checkbox in the Properties inspector. We display only the shape with the selected `RadioButton` at runtime.

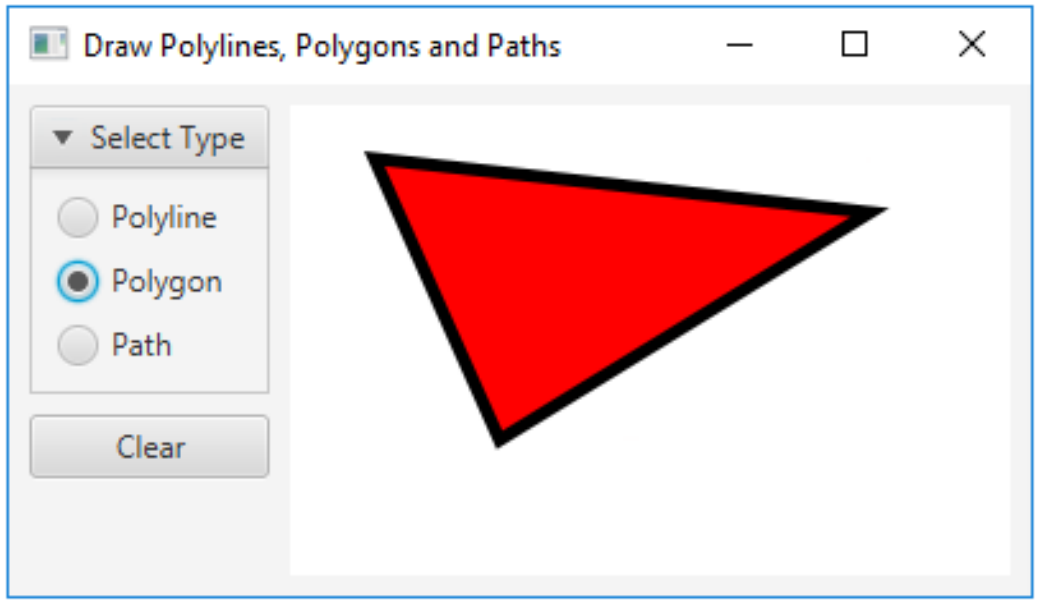
22.4.1 GUI and CSS

- We set the `Pane`'s On Mouse Clicked event handler to `drawingAreaMouseClicked`.
- We set the Clear `Button`'s On Action event handler to `clearButtonPressed`.
- We set the controller class to `PolyShapesController`.
- Finally, we edited the FXML to remove the `Path` object's `<elements>` and the `Polyline` and `Polygon` objects' `<points>`, as we'll set these programmatically in response to the user's mouse-click events.



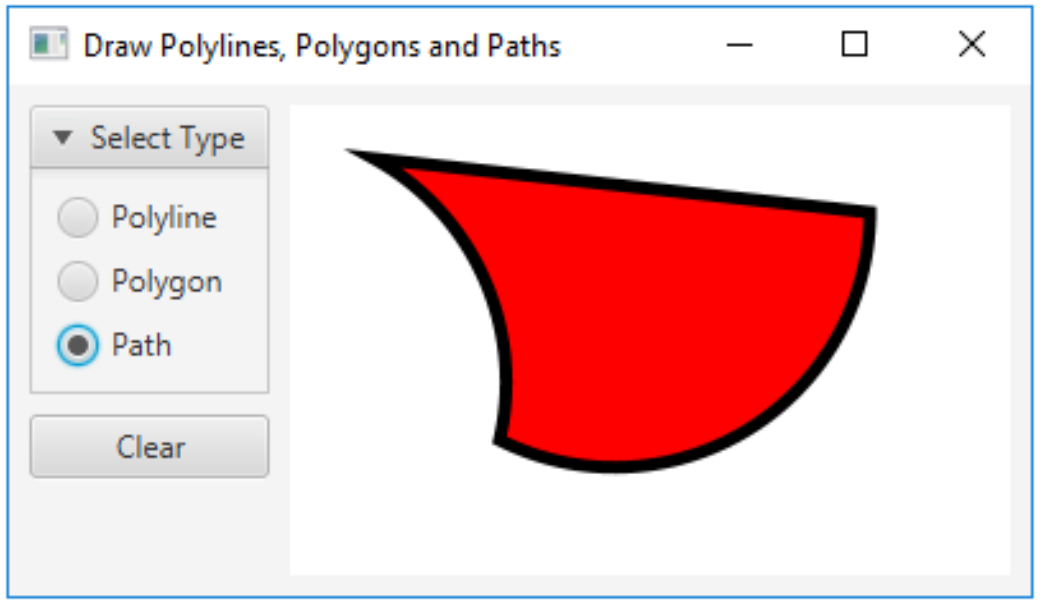
a) `Polyline` with three points—the starting and ending points are not connected

Fig. 22.5 | **`Polyline`s, `Polygon`s and `Path`s.** (Part 1 of 3.)



b) `Polygon` with three points—the starting and ending points are connected automatically

Fig. 22.5 | `Polyline`s, `Polygon`s and `Path`s. (Part 2 of 3.)



c) `Path` with two arc segments and a line connecting the first and last points

Fig. 22.5 | `Polylines`, `Polygons` and `Paths`. (Part 3 of 3.)

22.4.1 GUI and CSS (cont.)

- The `PolyShapes.css` file defines the properties `-fx-stroke`, `-fx-stroke-width` and `-fx-fill` that are applied to all three shapes in this example
- As you can see in Fig. 22.5, the stroke is a thick black line (5 pixels wide) and the fill is red
  - ```
    Polyline, Polygon, Path {
        -fx-stroke: black;
        -fx-stroke-width: 5;
        -fx-fill: red;
    }
    ```
    ```

22.4.2 `PolyShapesController` Class

- Figure 22.6 shows this app's `PolyShapesController` class, which responds to the user's interactions
- The `enum` `ShapeType` (line 17) defines three constants that we use to determine which shape to display
- Lines 20–26 declare the variables that correspond to the GUI components and shapes with `fx:id`s in the FXML
- The `shapeType` variable (line 29) stores whichever shape type is currently selected in the GUI's `RadioButtons`—by default, the `Polyline` will be displayed
- As you'll soon see, the `sweepFlag` variable is used to determine whether an arc in a `Path` is drawn with a negative or positive sweep angle

```
1  // Fig. 22.6: PolyShapesController.java
2  // Drawing Polylines, Polygons and Paths.
3  import javafx.event.ActionEvent;
4  import javafx.fxml.FXML;
5  import javafx.scene.control.RadioButton;
6  import javafx.scene.control.ToggleGroup;
7  import javafx.scene.input.MouseEvent;
8  import javafx.scene.shape.ArcTo;
9  import javafx.scene.shape.ClosePath;
10 import javafx.scene.shape.MoveTo;
11 import javafx.scene.shape.Path;
12 import javafx.scene.shape.Polygon;
13 import javafx.scene.shape.Polyline;
14
```

Fig. 22.6 | Drawing `Polylines`, `Polygons` and `Paths`. (Part 1 of 5.)

```
15  public class PolyShapesController {
16     // enum representing shape types
17     private enum ShapeType {POLYLINE, POLYGON, PATH};
18
19     // instance variables that refer to GUI components
20     @FXML private RadioButton polylineRadioButton;
21     @FXML private RadioButton polygonRadioButton;
22     @FXML private RadioButton pathRadioButton;
23     @FXML private ToggleGroup toggleGroup;
24     @FXML private Polyline polyline;
25     @FXML private Polygon polygon;
26     @FXML private Path path;
27
28     // instance variables for managing state
29     private ShapeType shapeType = ShapeType.POLYLINE;
30     private boolean sweepFlag = true; // used with arcs in a Path
31
```

Fig. 22.6 | Drawing `Polylines`, `Polygons` and `Paths`. (Part 2 of 5.)

```
32      // set user data for the RadioButtons and display polyline object
33      public void initialize() {
34         // user data on a control can be any Object
35         polylineRadioButton.setUserData(ShapeType.POLYLINE);
36         polygonRadioButton.setUserData(ShapeType.POLYGON);
37         pathRadioButton.setUserData(ShapeType.PATH);
38
39         displayShape(); // sets polyline's visibility to true when app loads
40      }
41
```

Fig. 22.6 | Drawing `Polylines`, `Polygons` and `Paths`. (Part 3 of 5.)

```
42      // handles drawingArea's onMouseClicked event
43      @FXML
44      private void drawingAreaMouseClicked(MouseEvent e) {
45         polyline.getPoints().addAll(e.getX(), e.getY());
46         polygon.getPoints().addAll(e.getX(), e.getY());
47
48         // if path is empty, move to first click position and close path
49         if (path.getElements().isEmpty()) {
50            path.getElements().add(new MoveTo(e.getX(), e.getY()));
51            path.getElements().add(new ClosePath());
52         }
53         else { // insert a new path segment before the ClosePath element
54            // create an arc segment and insert it in the path
55            ArcTo arcTo = new ArcTo();
56            arcTo.setX(e.getX());
57            arcTo.setY(e.getY());
58            arcTo.setRadiusX(100.0);
59            arcTo.setRadiusY(100.0);
60            arcTo.setSweepFlag(sweepFlag);
61            sweepFlag = !sweepFlag;
62            path.getElements().add(path.getElements().size() - 1, arcTo);
63         }
64      }
65
```

Fig. 22.6 | Drawing `Polylines`, `Polygons` and `Paths`. (Part 4 of 5.)

```
66      // handles color RadioButton's ActionEvents
67      @FXML
68      private void shapeRadioButtonSelected(ActionEvent e) {
69         // user data for each color RadioButton is a ShapeType constant
70         shapeType =
71            (ShapeType) toggleGroup.getSelectedToggle().getUserData();
72         displayShape(); // display the currently selected shape
73      }
74
75      // displays currently selected shape
76      private void displayShape() {
77         polyline.setVisible(shapeType == ShapeType.POLYLINE);
78         polygon.setVisible(shapeType == ShapeType.POLYGON);
79         path.setVisible(shapeType == ShapeType.PATH);
80      }
81
82      // resets each shape
83      @FXML
84      private void clearButtonPressed(ActionEvent event) {
85         polyline.getPoints().clear();
86         polygon.getPoints().clear();
87         path.getElements().clear();
88      }
89   }
```

Fig. 22.6 | Drawing `Polylines`, `Polygons` and `Paths`. (Part 5 of 5.)

22.4.2 PolyShapesController Class (cont.)

***Method* `initialize`**

- Recall from Section 13.3.1 that you can associate any `Object` with each JavaFX control via its `setUserData` method
- For the shape `RadioButtons` in this app, we store the specific `ShapeType` that the `RadioButton` represents (lines 35–37)
- We use these values when handling the `RadioButton` events to set the `shapeType` instance variable
- Line 39 then calls method `displayShape` to display the currently selected shape (the `Polyline` by default)
- Initially, the shape is not visible because it does not yet have any points.

22.4.2 PolyShapesController Class (cont.)

***Method* `drawingAreaMouseClicked`**

- When the user clicks the app's `Pane`, method `drawingAreaMouseClicked` (lines 43–64) modifies all three shapes to incorporate the new point at which the user clicked
- `Polylines` and `Polygons` store their points as a collection of `Double` values in which the first two values represent the first point's location, the next two values represent the second point's location, etc
- Line 45 gets the `polyline` object's collection of points, then adds the new click point to the collection by calling its `addAll` method and passing the `Mouse-Event`'s *x*- and *y*-coordinate values
- This adds the new point's information to the end of the collection

22.4.2 PolyShapesController Class (cont.)

- Line 46 performs the same task for the `polygon` object.
- Lines 49–63 manipulate the `path` object
- A `Path` is represented by a collection of `PathElements`
- The subclasses of `PathElement` used in this example are:
  - `MoveTo`—Moves to a specific position without drawing anything.
  - `ArcTo`—Draws an arc from the previous `PathElement`'s endpoint to the specified location. We'll discuss this in more detail momentarily.
  - `ClosePath`—Closes the path by drawing a straight line from the end point of the last `PathElement` to the start point of the first `PathElement`.
- Other `PathElements` not covered here include `LineTo`, `HLineTo`, `VLineTo`, `CubicCurveTo` and `QuadCurveTo`.

22.4.2 PolyShapesController Class (cont.)

- When the user clicks the `Pane`, line 49 checks whether the `Path` contains elements
- If not, line 50 moves the starting point of the `path` to the mouse-click location by adding a `MoveTo` element to the path's `PathElements` collection
- Then line 51 adds a new `ClosePath` element to complete the path
- For each subsequent mouse-click event, lines 55–60 create an `ArcTo` element and line 62 inserts it before the `ClosePath` element by calling the `PathElements` collection's `add` method that receives an index as its first argument.

22.4.2 PolyShapesController Class (cont.)

- Lines 56–57 set the `ArcTo` element's end point to the `MouseEvent`'s coordinates
- The arc is drawn as a piece of an ellipse for which you specify the horizontal radius and vertical radius (lines 58–59)
- Line 60 sets the `ArcTo`'s `sweepFlag`, which determines whether the arc sweeps in the positive angle direction (`true`; counter clockwise) or the negative angle direction (`false`; clockwise)
- By default an `ArcTo` element is drawn as the shortest arc between the last `PathElement`'s end point and the point specified by the `ArcTo` element
- To sweep the long way around the ellipse, set the `ArcTo`'s `largeArcFlag` to `true`
- For each mouse click, line 61 reverses the value of our controller class's `sweepFlag` instance variable so that the `ArcTo` elements toggle between positive and negative angles for variety

22.4.2 PolyShapesController Class (cont.)

Method *shapeRadioButtonSelected*

- When the user clicks a shape `RadioButton`, lines 70–71 set the controller's `shapeType` instance variable, then line 72 calls method `displayShape` to display the selected shape
- Try creating a `Polyline` of several points, then changing to the `Polygon` and `Path` to see how the points are used in each shape.

22.4.2 PolyShapesController Class (cont.)

***Method* `displayShape`**

- Lines 77–79 simply set the visibility of the three shapes, based on the current `shapeType`
- The currently selected shape's visibility is set to `true` to display the shape, and the other shapes' visibility is set to `false` to hide those shapes.

***Method* `clearButtonPressed`**

- When the user clicks the Clear `Button`, lines 85–86 clear the `polyline`'s and `polygon`'s collections of points, and line 87 clears the `path`'s collection of `PathElements`

22.5 Transforms

- A transform can be applied to any UI element to *reposition* or *reorient* the element
- The built-in JavaFX transforms are subclasses of `Transform`
- Some of these subclasses include:
  - `Translate`—*moves* an object to a new location.
  - `Rotate`—*rotates* an object around a point and by a specified rotation angle.
  - `Scale`—*scales* an object's size by the specified amounts.
- The next example draws stars using the `Polygon` control and uses `Rotate` transforms to create a circle of randomly colored stars
- The FXML for this app consists of an empty 300-by-300 `Pane` layout with the `fx:id "pane"`
- We also set the controller class to `DrawStars-Controller`

22.5 Transforms (cont.)

- Figure 22.7 shows the app's controller and a sample output
- Method initialize (lines 14–37) defines the stars, applies the transforms and attaches the stars to the app's `pane`
- Lines 16–18 define the points of a star as an array of type `Double`—the collection of points stored in a `Polygon` is implemented with a generic collection, so you must use type `Double` rather than `double` (recall that primitive types cannot be used in Java generics)
- Each pair of values in the array represents the *x*- and *y*-coordinates of one point in the `Polygon`
- We defined ten points in the array

```
1  // Fig. 22.7: DrawStarsController.java
2  // Create a circle of stars using Polygons and Rotate transforms
3  import java.security.SecureRandom;
4  import javafx.fxml.FXML;
5  import javafx.scene.layout.Pane;
6  import javafx.scene.paint.Color;
7  import javafx.scene.shape.Polygon;
8  import javafx.scene.transform.Transform;
9
10 public class DrawStarsController {
11    @FXML private Pane pane;
12    private static final SecureRandom random = new SecureRandom();
13
14    public void initialize() {
15       // points that define a five-pointed star shape
16       Double[] points = {205.0,150.0, 217.0,186.0, 259.0,186.0,
17          223.0,204.0, 233.0,246.0, 205.0,222.0, 177.0,246.0, 187.0,204.0,
18          151.0,186.0, 193.0,186.0};
19
```

Fig. 22.7 | Create a circle of stars using `Polygons` and `Rotate` transforms. (Part 1 of 3.)

```
20          // create 18 stars
21          for (int count = 0; count < 18; ++count) {
22             // create a new Polygon and copy existing points into it
23             Polygon newStar = new Polygon();
24             newStar.getPoints().addAll(points);
25
26             // create random Color and set as newStar's fill
27             newStar.setStroke(Color.GREY);
28             newStar.setFill(Color.rgb(random.nextInt(255),
29                random.nextInt(255), random.nextInt(255),
30                random.nextDouble()));
31
32             // apply a rotation to the shape
33             newStar.getTransforms().add(
34                Transform.rotate(count * 20, 150, 150));
35             pane.getChildren().add(newStar);
36          }
37       }
38    }
```

Fig. 22.7 | Create a circle of stars using `Polygons` and `Rotate` transforms. (Part 2 of 3.)

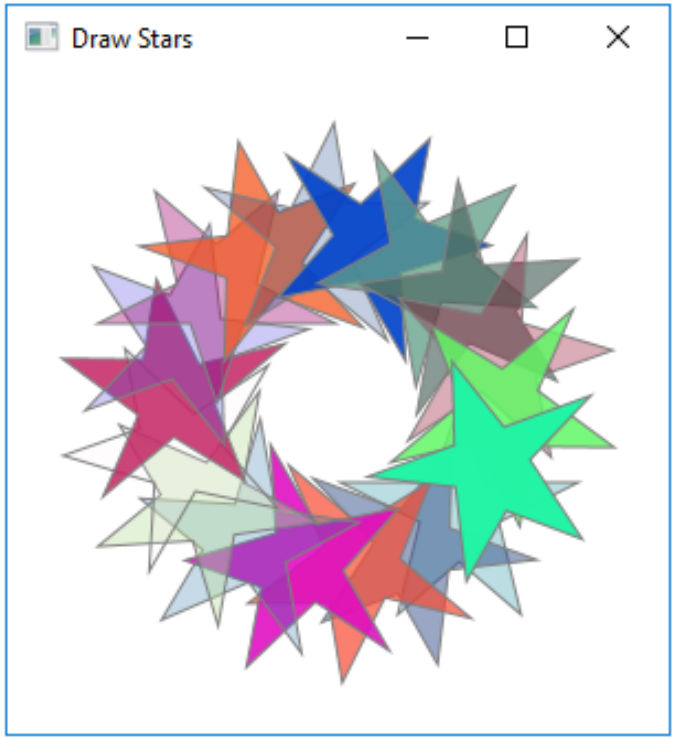


Fig. 22.7 | Create a circle of stars using `Polygon`s and `Rotate` transforms. (Part 3 of 3.)

22.5 Transforms (cont.)

- During each iteration of the loop, lines 23–34 create a `Polygon` using the points in the `points` array and apply a different `Rotate` transform
- This results in the circle of `Polygons` in the screen capture
- To generate the random colors for each star, we use a `SecureRandom` object to create three random values from 0–255 for the red, green and blue components of the color, and one random value from 0.0–1.0 for the color's alpha transparency value
- We pass those values to class `Color`'s `static rgb` method to create a `Color`
- To apply a rotation to the new `Polygon`, we add a `Rotate` transform to the `Polygon`'s collection of `Transforms` (lines 33–34)

22.5 Transforms (cont.)

- To create the `Rotate` transform object, we invoke class `Transform`'s `static` method `rotate` (line 34), which returns a `Rotate` object
- The method's first argument is the rotation angle
- Each iteration of the loop assigns a new rotation-angle value by using the control variable multiplied by 20 as the `rotate` method's first argument
- The method's next two arguments are the *x*- and *y*-coordinates of the point of rotation around which the `Polygon` rotates
- The center of the circle of stars is the point *(150, 150)*, because we rotated all 18 stars around that point
- Adding each `Polygon` as a new child element of the `pane` object allows the `Polygon` to be rendered on screen.

22.6 Playing Video with `Media`, `MediaPlayer` and `MediaViewer`

- Many of today's most popular apps are multimedia intensive
- JavaFX provides audio and video multimedia capabilities via the classes of package `javafx.scene.media`:
  - For simple audio playback you can use class `AudioClip`
  - For audio playback with more playback controls and for video playback you can use classes `Media`, `MediaPlayer` and `MediaView`.
- In this section, you'll build a basic video player
- We'll explain classes `Media`-, `MediaPlayer` and `MediaView` as we encounter them in the project's controller class (Section 22.6.2)

22.6 TPlaying Video with `Media`, `MediaPlayer` and `MediaViewer` (cont.)

- The video used in this example is from NASA's multimedia library
  - `http://www.nasa.gov/centers/kennedy/multimedia/HD-index.html`
- The video file `sts117.mp4` is provided in the `video` folder with this chapter's examples
- When building the app from scratch, copy the video onto the app's folder.

Media Formats

- JavaFX supports MPEG-4 (also called MP4) and Flash Video formats
- We downloaded a Windows WMV version of the video file used in this example, then converted it to MP4 via a free online video converter.
  - There are many free online and downloadable video-format conversion toolsWe used the one at https://convertio.co/video-converter/.

22.6 Playing Video with `Media`, `MediaPlayer` and `MediaViewer` (cont.)

ControlsFX Library's `ExceptionDialog`

- `ExceptionDialog` is one of many additional JavaFX controls available through the open-source project ControlsFX at
  - `http://controlsfx.org`
- We use an `ExceptionDialog` in this app to display a message to the user if an error occurs during media playback.
- Place the extracted ControlsFX JAR file (named `controlsfx-8.40.12.jar` at the time of this writing) in your project's folder—a JAR file is a compressed archive like a ZIP file, but contains Java class files and their corresponding resources
- We included a copy of the JAR file with the final example

22.6 TPlaying Video with `Media`, `MediaPlayer` and `MediaViewer` (cont.)

Compiling and Running the App with ControlsFX

- To compile this app, you must specify the JAR file as part of the app's classpath
- To do so, use the `javac` command's `-classpath` option, as in:
  - `javac -classpath .;controlsfx-8.40.12.jar *.java`
- Similary, to run the app, use the `java` command's `-cp` option, as in
  - `java -cp .;controlsfx-8.40.12.jar VideoPlayer`
- In the preceding commands, Linux and macOS users should use a colon (`:`) rather than a semicolon(`;`)
- The classpath in each command specifies the current folder containing the app's files—this is represented by the dot (`.`)—and the name of the JAR file containing the ControlsFX classes (including `ExceptionDialog`)

22.6.1 VideoPlayer GUI

- Figure 22.8 shows the completed `VideoPlayer.fxml` file and two sample screen captures of the final running `VideoPlayer` app
- The GUI's layout is a `BorderPane` consisting of
  - a `MediaView` (located in the Scene Builder Library's Controls section) with the `fx:id` `mediaView` and
  - a `ToolBar` (located in the Scene Builder Library's Containers section) containing one `Button` with the `fx:id` `playPauseButton` and the text `"Play"`
  - The controller method `playPauseButtonPressed` responds when the `Button` is pressed

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 22.8: VideoPlayer.fxml -->
3  <!-- VideoPlayer GUI with a MediaView and a Button -->
4
5  <?import javafx.scene.control.Button?>
6  <?import javafx.scene.control.ToolBar?>
7  <?import javafx.scene.layout.BorderPane?>
8  <?import javafx.scene.media.MediaView?>
9
10 <BorderPane prefHeight="400.0" prefWidth="600.0"
11    style="-fx-background-color: black;"
12    xmlns="http://javafx.com/javafx/8.0.60"
13    xmlns:fx="http://javafx.com/fxml/1"
14    fx:controller="VideoPlayerController">
```

Fig. 22.8 | `VideoPlayer` GUI with a `MediaView` and a `Button`. Video courtesy of NASA—see `http://www.nasa.gov/multimedia/guidelines/` for usage guidelines. (Part 1 of 4.)

```
15      <bottom>
16         <ToolBar prefHeight="40.0" prefWidth="200.0"
17            BorderPane.alignment="CENTER">
18            <items>
19               <Button fx:id="playPauseButton"
20                  onAction="#playPauseButtonPressed" prefHeight="25.0"
21                  prefWidth="60.0" text="Play" />
22            </items>
23         </ToolBar>
24      </bottom>
25      <center>
26         <MediaView fx:id="mediaView" BorderPane.alignment="CENTER" />
27      </center>
28   </BorderPane>
```

Fig. 22.8 | `VideoPlayer` GUI with a `MediaView` and a `Button`. Video courtesy of NASA—see `http://www.nasa.gov/multimedia/guidelines/` for usage guidelines. (Part 2 of 4.)

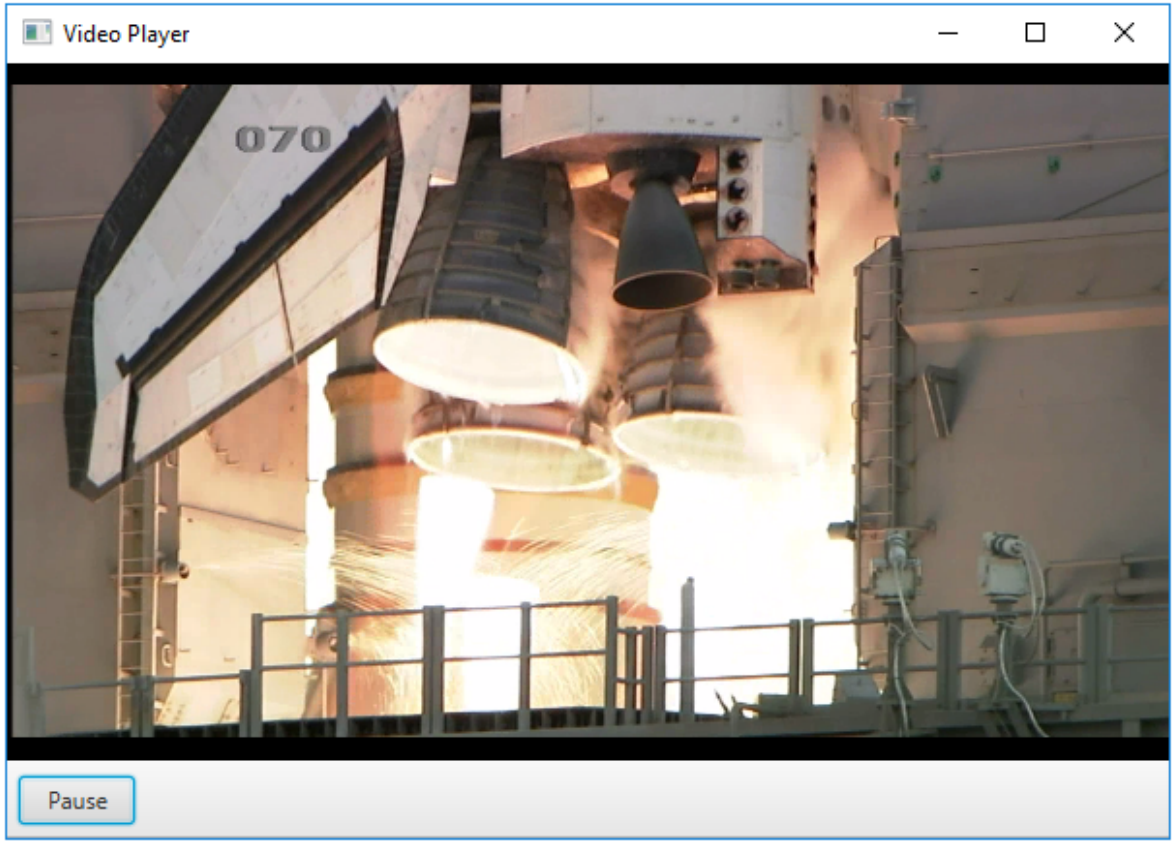


Fig. 22.8 | `VideoPlayer` GUI with a `MediaView` and a `Button`. Video courtesy of NASA—see `http://www.nasa.gov/multimedia/guidelines/` for usage guidelines. (Part 3 of 4.)

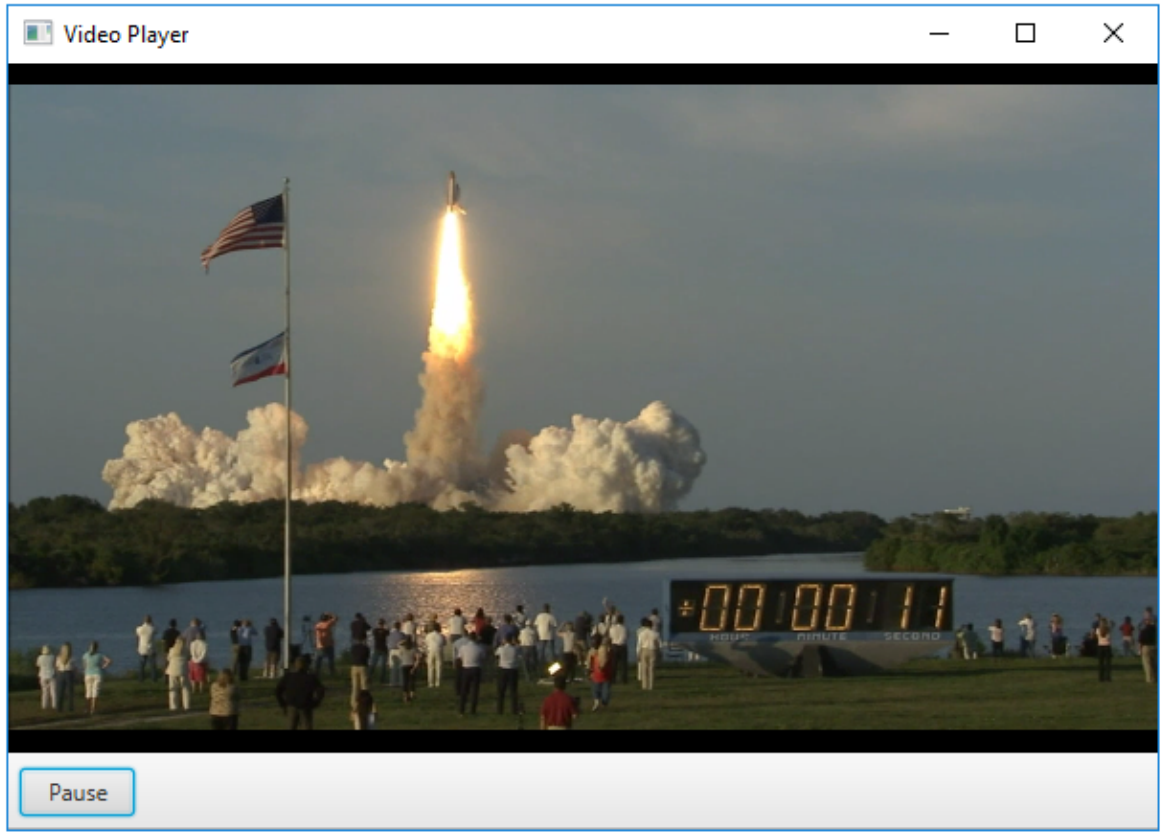


Fig. 22.8 | `VideoPlayer` GUI with a `MediaView` and a `Button`. Video courtesy of NASA—see `http://www.nasa.gov/multimedia/guidelines/` for usage guidelines. (Part 4 of 4.)

22.6.1 VideoPlayer GUI (cont.)

- We placed the `MediaView` in the `BorderPane`'s center region (lines 25–27) so that it occupies all available space in the `BorderPane`, and we placed the `ToolBar` in the `BorderPane`'s bottom region (lines 15–24)
- By default, Scene Builder adds one `Button` to the `ToolBar` when you drag the `ToolBar` onto your layout
- You can then add other controls to the `ToolBar` as necessary
- We set the controller class to `VideoPlayerController`

22.6.2 VideoPlayerController Class

- Figure 22.9 shows the completed `VideoPlayerController` class, which configures video playback and responds to state changes from the `MediaPlayer` and the events when the user presses the `playPauseButton`
- The controller uses classes `Media-`, `MediaPlayer` and `MediaView` as follows:
  - A `Media` object specifies the location of the media to play and provides access to various information about the media, such as its duration, dimensions and more.
  - A `MediaPlayer` object loads a `Media` object and controls playback. In addition, a `MediaPlayer` transitions through its various states (*ready*, *playing*, *paused*, etc.) during media loading and playback. As you'll see, you can provide `Runnables` that execute in response to these state transitions.
  - A `MediaView` object displays the `Media-` being played by a given `MediaPlayer` object-.

```
1  // Fig. 22.9: VideoPlayerController.java
2  // Using Media, MediaPlayer and MediaView to play a video.
3  import java.net.URL;
4  import javafx.beans.binding.Bindings;
5  import javafx.beans.property.DoubleProperty;
6  import javafx.event.ActionEvent;
7  import javafx.fxml.FXML;
8  import javafx.scene.control.Button;
9  import javafx.scene.media.Media;
10 import javafx.scene.media.MediaPlayer;
11 import javafx.scene.media.MediaView;
12 import javafx.util.Duration;
13 import org.controlsfx.dialog.ExceptionDialog;
14
```

Fig. 22.9 | Using `Media`, `MediaPlayer` and `MediaView` to play a video. (Part 1 of 6.)

```
15  public class VideoPlayerController {
16     @FXML private MediaView mediaView;
17     @FXML private Button playPauseButton;
18     private MediaPlayer mediaPlayer;
19     private boolean playing = false;
20
21     public void initialize() {
22        // get URL of the video file
23        URL url = VideoPlayerController.class.getResource("sts117.mp4");
24
25        // create a Media object for the specified URL
26        Media media = new Media(url.toExternalForm());
27
28        // create a MediaPlayer to control Media playback
29        mediaPlayer = new MediaPlayer(media);
30
31        // specify which MediaPlayer to display in the MediaView
32        mediaView.setMediaPlayer(mediaPlayer);
```

Fig. 22.9 | Using `Media`, `MediaPlayer` and `MediaView` to play a video. (Part 2 of 6.)

```
33
34          // set handler to be called when the video completes playing
35          mediaPlayer.setOnEndOfMedia(
36             new Runnable() {
37                public void run() {
38                   playing = false;
39                   playPauseButton.setText("Play");
40                   mediaPlayer.seek(Duration.ZERO);
41                   mediaPlayer.pause();
42                }
43             }
44          );
45
```

Fig. 22.9 | Using `Media`, `MediaPlayer` and `MediaView` to play a video. (Part 3 of 6.)

```
46          // set handler that displays an ExceptionDialog if an error occurs
47          mediaPlayer.setOnError(
48             new Runnable() {
49                public void run() {
50                   ExceptionDialog dialog =
51                      new ExceptionDialog(mediaPlayer.getError());
52                   dialog.showAndWait();
53                }
54             }
55          );
56
```

Fig. 22.9 | Using `Media`, `MediaPlayer` and `MediaView` to play a video. (Part 4 of 6.)

```
57          // set handler that resizes window to video size once ready to play
58          mediaPlayer.setOnReady(
59             new Runnable() {
60                public void run() {
61                   DoubleProperty width = mediaView.fitWidthProperty();
62                   DoubleProperty height = mediaView.fitHeightProperty();
63                   width.bind(Bindings.selectDouble(
64                      mediaView.sceneProperty(), "width"));
65                   height.bind(Bindings.selectDouble(
66                      mediaView.sceneProperty(), "height"));
67                }
68             }
69          );
70       }
71
```

Fig. 22.9 | Using `Media`, `MediaPlayer` and `MediaView` to play a video. (Part 5 of 6.)

```
72        // toggle media playback and the text on the playPauseButton
73        @FXML
74        private void playPauseButtonPressed(ActionEvent e) {
75           playing = !playing;
76
77           if (playing) {
78              playPauseButton.setText("Pause");
79              mediaPlayer.play();
80           }
81           else {
82              playPauseButton.setText("Play");
83              mediaPlayer.pause();
84           }
85        }
86     }
```

Fig. 22.9 | Using `Media`, `MediaPlayer` and `MediaView` to play a video. (Part 6 of 6.)

22.6.2 `VideoPlayerController` Class (cont.)

Instance Variables

- Lines 16–19 declare the controller's instance variables
- When the app loads, the `mediaView` variable (line 16) is assigned a reference to the `MediaView` object declared in the app's FXML
- The `mediaPlayer` variable (line 18) is configured in method `initialize` to load the video specified by a `Media` object and used by method `playPauseButtonPressed` (lines 73–85) to play and pause the video.

22.6.2 `VideoPlayerController` Class (cont.)

Creating a `Media` Object Representing the Video to Play

- Method `initialize` configures media playback and registers event handlers for `Media-Player` events
- Line 23 gets a `URL` representing the location of the `sts117.mp4` video file
- `VideoPlayerController.class` creates a `Class` object representing the `VideoPlayerController` class
- Next line 26 creates a `Media` object representing the video
- The argument to the `Media` constructor is a `String` representing the video's location-, which we obtain with `URL` method `toExternalForm`
- The URL `String` can represent a local file on your computer or can be a location on the web
- The `Media` constructor throws various exceptions, including `MediaExceptions` if the media cannot be found or is not of a supported media format.

22.6.2 `VideoPlayerController` Class (cont.)

Creating a `MediaPlayer` Object to Load the Video and Control Playback

- To load the video and prepare it for playback, you must associate it with a `MediaPlayer` object (line 29)
- Playing multiple videos requires a separate `MediaPlayer` for each `Media` object
- However, a given `Media` object can be associated with multiple `MediaPlayers`
- The `MediaPlayer` constructor throws a `NullPointerException` if the `Media` is `null` or a `Media-Exception` if a problem occurs during construction of the `MediaPlayer` object.

22.6.2 `VideoPlayerController` Class (cont.)

Attaching the `MediaPlayer` Object to the `MediaView` to Display the Video

- A `MediaPlayer` does not provide a view in which to display video
- For this purpose, you must associate a `MediaPlayer` with a `MediaView`
- When the `MediaView` already exists you call the `MediaView`'s `setMediaPlayer` method (line 32) to perform this task
- When creating a `Media-View` object programmatically, you can pass the `MediaPlayer` to the `MediaView`'s constructor
- A `MediaView` is like any other `Node` in the scene graph, so you can apply CSS styles, transforms and animations (Sections 22.7–22.9) to it as well

22.6.2 `VideoPlayerController` Class (cont.)

Configuring Event Handlers for `MediaPlayer` Events

- A `MediaPlayer` transitions through various states
- Some common states include *ready*, *playing* and *paused*
- For these and other states, you can execute a task as the `MediaPlayer` enters the corresponding state
- In addition, you can specify tasks that execute when the end of media playback is reached or when an error occurs during playback
- To perform a task for a given state, you specify an object that implements the `Runnable` interface (package `java.lang`)
- This interface contains a no-parameter `run` method that returns `void`

22.6.2 `VideoPlayerController` Class (cont.)

- Lines 35–44 call the `MediaPlayer`'s `setOnEndOfMedia` method, passing an object of an anonymous inner class that implements interface `Runnable` to execute when video playback completes
- Line 38 sets the `boolean` instance variable `playing` to `false` and line 39 changes the text on the `playPauseButton` to `"Play"` to indicate that the user can click the `Button` to play the video again
- Line 40 calls `MediaPlayer` method `seek` to move to the beginning of the video and line 41 pauses the video.
- Lines 47–55 call the `MediaPlayer`'s `setOnError` method to specify a task to perform if the `MediaPlayer` enters the *error* state, indicating that an error occurred during playback

22.6.2 `VideoPlayerController` Class (cont.)

Binding the `MediaViewer`'s Size to the Scene's Size

- Lines 58–69 call the `MediaPlayer`'s `setOnReady` method to specify a task to perform if the `MediaPlayer` enters the *ready* state
- We use property bindings to bind the `MediaView`'s `width` and `height` properties to the scene's `width` and `height` properties so that the `MediaView` resizes with app's window
- A `Node`'s `sceneProperty` returns a `ReadOnlyObjectProperty<Scene>` that you can use to access to the `Scene` in which the `Node` is displayed
- The `ReadOnlyObjectProperty<Scene>` represents an object that has many properties

22.6.2 `VideoPlayerController` Class (cont.)

- To bind to a specific properties of that object, you can use the methods of class `Bindings` (package `javafx.beans.binding`) to select the corresponding properties
- The `Scene`'s `width` and `height` are each `DoubleProperty` objects
- `Bindings` method `selectDouble` gets a reference to a `DoubleProperty`
- The method's first argument is the object that contains the property and the second argument is the name of the property to which you'd like to bind.

22.6.2 `VideoPlayerController` Class (cont.)

***Method* `playPauseButtonPressed`**

- The event handler `playPauseButtonPressed` (lines 73–85) toggles video playback
- When `playing` is `true`, line 78 sets the `playPauseButton`'s text to `"Pause"` and line 79 calls the `MediaPlayer`'s `play` method; otherwise, line 82 sets the `playPauseButton`'s text to `"Play"` and line 83 calls the `MediaPlayer`'s `pause` method.

***Using Java SE 8 Lambdas to Implement the* `Runnable`s**

- Each of the anonymous inner classes in this controller's `initialize` method can be implemented more concisely using lambdas as shown in Section 17.16.

22.7 Transition Animations

- Animations in JavaFX apps transition a `Node`'s property values from one value to another in a specified amount of time
- Most properties of a `Node` can be animated
- This section focuses on several of JavaFX's predefined `Transition` animations from the `javafx.animations` package
- By default, the subclasses that define `Transition` animations change the values of specific `Node` properties
- For example, a `FadeTransition` changes the value of a `Node`'s `opacity` property (which specifies whether the `Node` is opaque or transparent) over time, whereas a `PathTransition` changes a `Node`'s location by moving it along a `Path` over time
- Though we show sample screen captures for all the animation examples, the best way to experience each is to run the examples yourself.

22.7.1 TransitionAnimations.fxml

- Figure 22.10 shows this app's GUI and screen captures of the running application
- When you click the `startButton` (lines 17–19), its `startButtonPressed` event handler in the app's controller creates a sequence of `Transition` animations for the `Rectangle` (lines 15–16) and plays them
- The `Rectangle` is styled with the following CSS which produces a rounded rectangle with a 10-pixel red border and yellow fill
  - ```
    Rectangle {
        -fx-stroke-width: 10;
        -fx-stroke: red;
        -fx-arc-width: 50;
        -fx-arc-height: 50;
        -fx-fill: yellow;
    }
    ```
    ```

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 22.10: TransitionAnimations.fxml -->
3  <!-- FXML for a Rectangle and Button -->
4
5  <?import javafx.scene.control.Button?>
6  <?import javafx.scene.layout.Pane?>
7  <?import javafx.scene.shape.Rectangle?>
8
9  <Pane id="Pane" prefHeight="200.0" prefWidth="180.0"
10    stylesheets="@TransitionAnimations.css"
11    xmlns="http://javafx.com/javafx/8.0.60"
12    xmlns:fx="http://javafx.com/fxml/1"
13    fx:controller="TransitionAnimationsController">
14    <children>
15       <Rectangle fx:id="rectangle" height="90.0" layoutX="45.0"
16          layoutY="45.0" width="90.0" />
17       <Button fx:id="startButton" layoutX="38.0" layoutY="161.0"
18          mnemonicParsing="false"
19          onAction="#startButtonPressed" text="Start Animations" />
20    </children>
21 </Pane>
```

Fig. 22.10 | FXML for a `Rectangle` and `Button`. (Part 1 of 7.)

a) Initial Rectangle

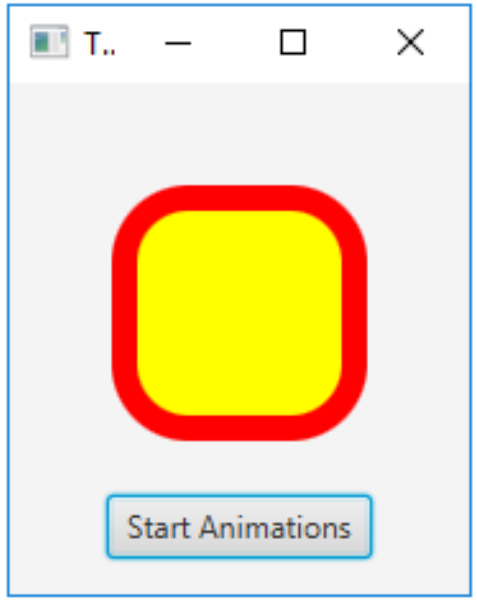


Fig. 22.10 | FXML for a **`Rectangle`** and **`Button`**. (Part 2 of 7.)

b) Rectangle undergoing parallel fill and stroke transitions

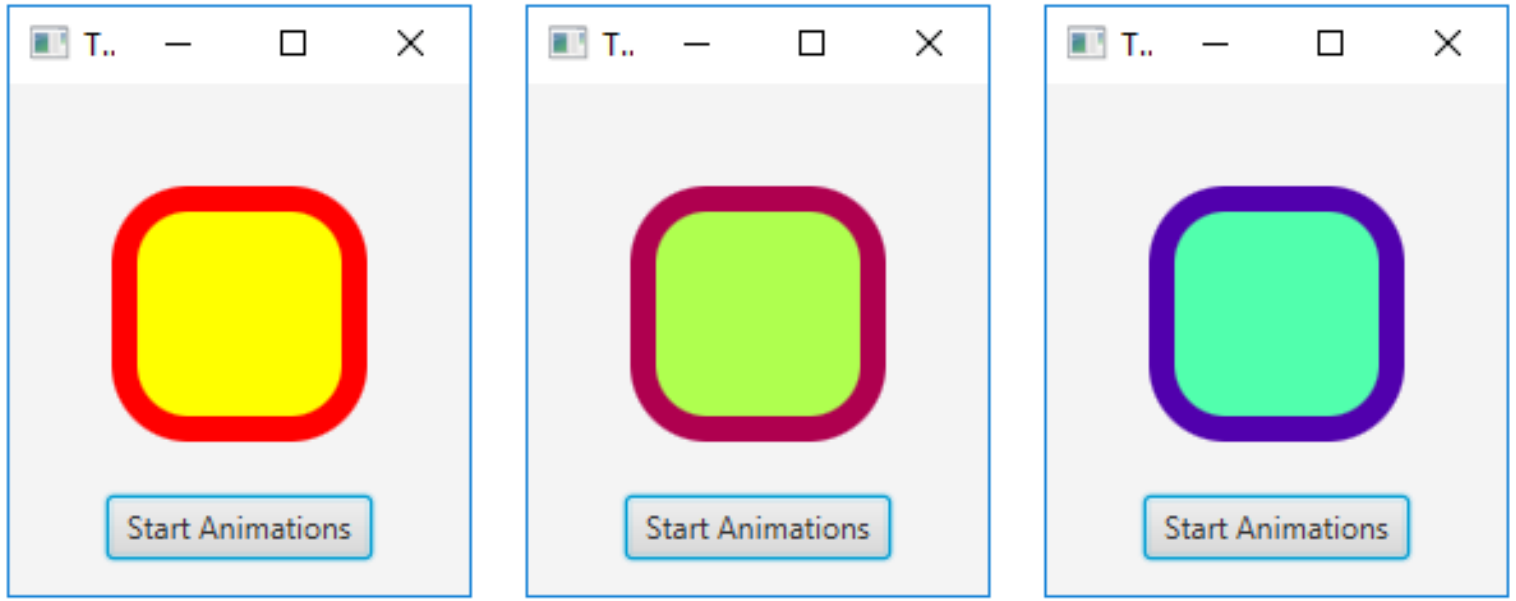


Fig. 22.10 | FXML for a **Rectangle** and **Button**. (Part 3 of 7.)

c) Rectangle undergoing a fade transition

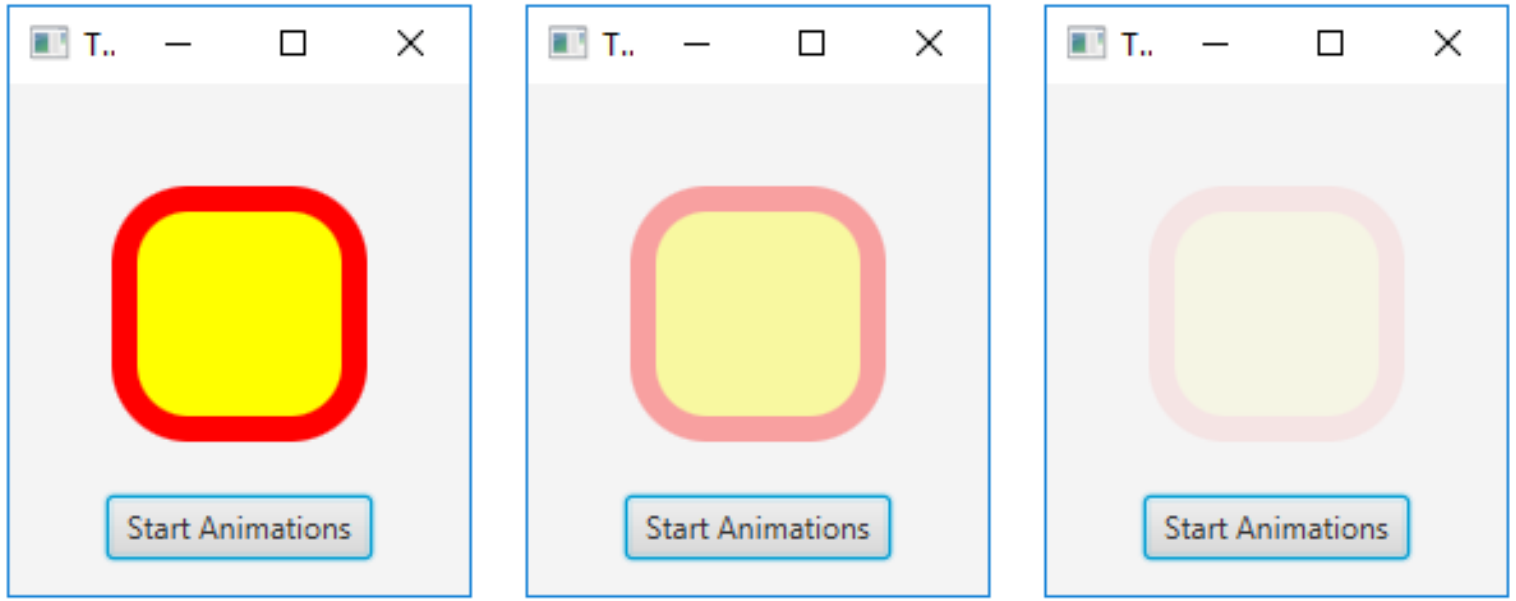


Fig. 22.10 | FXML for a **Rectangle** and **Button**. (Part 4 of 7.)

d) Rectangle undergoing a rotate transition

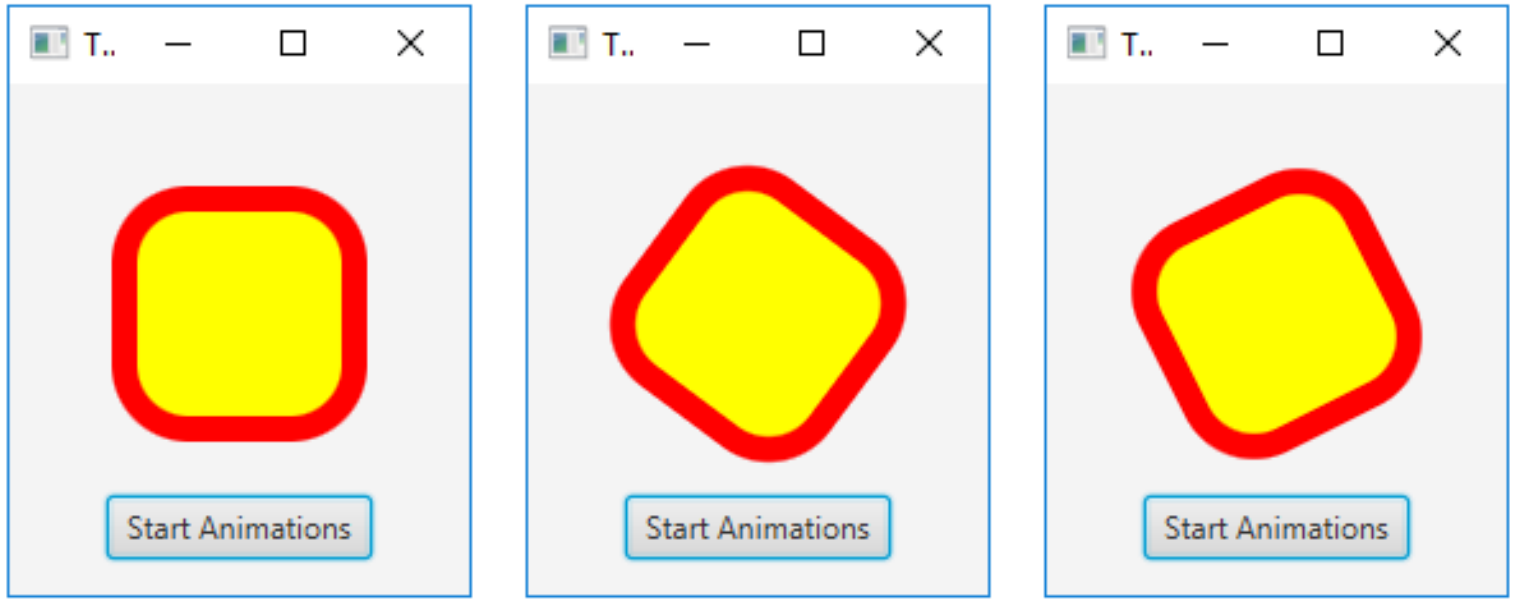


Fig. 22.10 | FXML for a **Rectangle** and **Button**. (Part 5 of 7.)

e) Rectangle undergoing a path transition

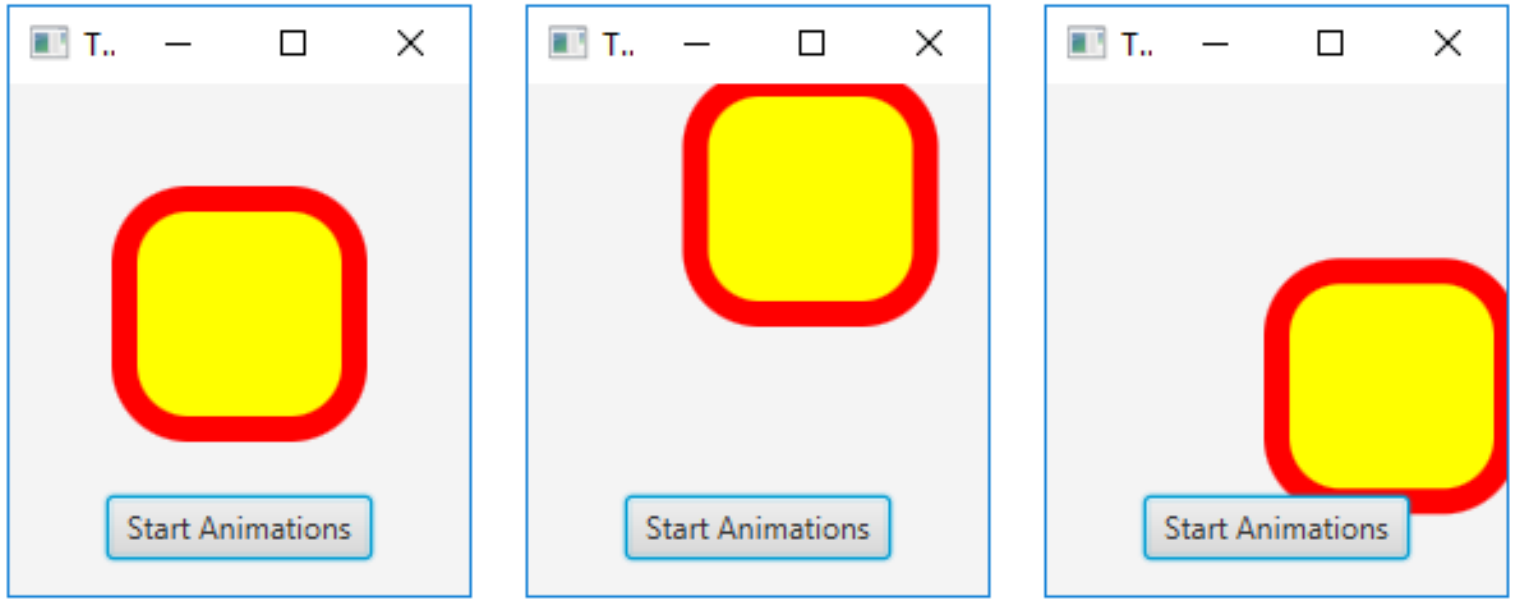


Fig. 22.10 | FXML for a **Rectangle** and **Button**. (Part 6 of 7.)

f) Rectangle undergoing a scale transition

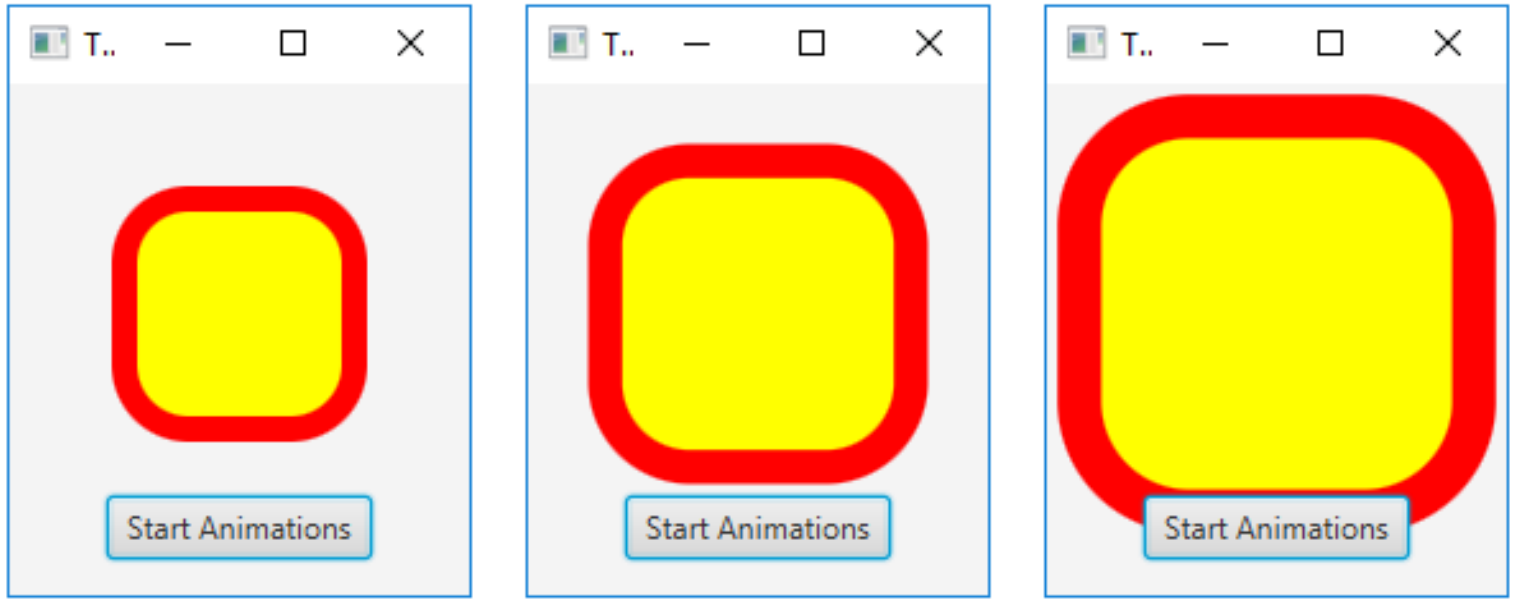


Fig. 22.10 | FXML for a **Rectangle** and **Button**. (Part 7 of 7.)

22.7.2 `TransitionAnimationsController` Class

- Figure 22.11 shows this app's controller class, which defines the `startButton`'s event handler (lines 25–87)
- This event handler defines several animations that are played in sequence
-

```
1  // Fig. 22.11: TransitionAnimationsController.java
2  // Applying Transition animations to a Rectangle.
3  import javafx.animation.FadeTransition;
4  import javafx.animation.FillTransition;
5  import javafx.animation.Interpolator;
6  import javafx.animation.ParallelTransition;
7  import javafx.animation.PathTransition;
8  import javafx.animation.RotateTransition;
9  import javafx.animation.ScaleTransition;
10 import javafx.animation.SequentialTransition;
11 import javafx.animation.StrokeTransition;
12 import javafx.event.ActionEvent;
13 import javafx.fxml.FXML;
14 import javafx.scene.paint.Color;
15 import javafx.scene.shape.LineTo;
16 import javafx.scene.shape.MoveTo;
17 import javafx.scene.shape.Path;
18 import javafx.scene.shape.Rectangle;
19 import javafx.util.Duration;
```

Fig. 22.11 | Applying `Transition` animations to a `Rectangle`. (Part 1 of 5.)

```
20
21   public class TransitionAnimationsController {
22      @FXML private Rectangle rectangle;
23
24      // configure and start transition animations
25      @FXML
26      private void startButtonPressed(ActionEvent event) {
27         // transition that changes a shape's fill
28         FillTransition fillTransition =
29            new FillTransition(Duration.seconds(1));
30         fillTransition.setToValue(Color.CYAN);
31         fillTransition.setCycleCount(2);
32
33         // each even cycle plays transition in reverse to restore original
34         fillTransition.setAutoReverse(true);
35
```

Fig. 22.11 | Applying `Transition` animations to a `Rectangle`. (Part 2 of 5.)

```
36          // transition that changes a shape's stroke over time
37          StrokeTransition strokeTransition =
38             new StrokeTransition(Duration.seconds(1));
39          strokeTransition.setToValue(Color.BLUE);
40          strokeTransition.setCycleCount(2);
41          strokeTransition.setAutoReverse(true);
42
43          // parallelizes multiple transitions
44          ParallelTransition parallelTransition =
45             new ParallelTransition(fillTransition, strokeTransition);
46
47          // transition that changes a node's opacity over time
48          FadeTransition fadeTransition =
49             new FadeTransition(Duration.seconds(1));
50          fadeTransition.setFromValue(1.0); // opaque
51          fadeTransition.setToValue(0.0); // transparent
52          fadeTransition.setCycleCount(2);
53          fadeTransition.setAutoReverse(true);
```

Fig. 22.11 | Applying `Transition` animations to a `Rectangle`. (Part 3 of 5.)

```
54
55          // transition that rotates a node
56          RotateTransition rotateTransition =
57             new RotateTransition(Duration.seconds(1));
58          rotateTransition.setByAngle(360.0);
59          rotateTransition.setCycleCount(2);
60          rotateTransition.setInterpolator(Interpolator.EASE_BOTH);
61          rotateTransition.setAutoReverse(true);
62
63          // transition that moves a node along a Path
64          Path path = new Path(new MoveTo(45, 45), new LineTo(45, 0),
65             new LineTo(90, 0), new LineTo(90, 90), new LineTo(0, 90));
66          PathTransition translateTransition =
67             new PathTransition(Duration.seconds(2), path);
68          translateTransition.setCycleCount(2);
69          translateTransition.setInterpolator(Interpolator.EASE_IN);
70          translateTransition.setAutoReverse(true);
71
```

Fig. 22.11 | Applying `Transition` animations to a `Rectangle`. (Part 4 of 5.)

```
72          // transition that scales a shape to make it larger or smaller
73          ScaleTransition scaleTransition =
74             new ScaleTransition(Duration.seconds(1));
75          scaleTransition.setByX(0.75);
76          scaleTransition.setByY(0.75);
77          scaleTransition.setCycleCount(2);
78          scaleTransition.setInterpolator(Interpolator.EASE_OUT);
79          scaleTransition.setAutoReverse(true);
80
81          // transition that applies a sequence of transitions to a node
82          SequentialTransition sequentialTransition =
83             new SequentialTransition (rectangle, parallelTransition,
84                fadeTransition, rotateTransition, translateTransition,
85                scaleTransition);
86          sequentialTransition.play(); // play the transition
87       }
88    }
```

Fig. 22.11 | Applying `Transition` animations to a `Rectangle`. (Part 5 of 5.)

22.7.2 `TransitionAnimationsController` Class (cont.)

`FillTransition`

- Lines 28–34 configure a one-second `FillTransition` that changes a shape's fill color
- Line 30 specifies the color (`CYAN`) to which the fill will transition
- Line 31 sets the animations cycle count to 2—this specifies the number of iterations of the transition to perform over the specified duration
- Line 34 specifies that the animation should automatically play itself in reverse once the initial transition is complete
- For this animation, during the first cycle the fill color changes from the original fill color to `CYAN`, and during the second cycle the animation transitions back to the original fill color

22.7.2 `TransitionAnimationsController` Class (cont.)

`StrokeTransition`

- Lines 37–41 configure a one-second `StrokeTransition` that changes a shape's stroke color
- Line 39 specifies the color (`BLUE`) to which the stroke will transition
- Line 40 sets the animations cycle count to 2, and line 41 specifies that the animation should automatically play itself in reverse once the initial transition is complete
- For this animation, during the first cycle the stroke color changes from the original stroke color to `BLUE`, and during the second cycle the animation transitions back to the original stroke color

22.7.2 TransitionAnimationsController Class (cont.)

ParallelTransition

- Lines 44–45 configure a `ParallelTransition` that performs multiple transitions at the same time (that is, in parallel)
- The `ParallelTransition` constructor receives a variable number of `Transitions` as a comma-separated list
- In this case, the `FillTransition` and `StrokeTransition` will be performed in parallel on the app's `Rectangle`

22.7.2 `TransitionAnimationsController` Class (cont.)

`FadeTransition`

- Lines 48–53 configure a one-second `FadeTransition` that changes a `Node`'s opacity
- Line 50 specifies the initial opacity—1.0 is fully opaque
- Line 51 specifies the final opacity—0.0 is fully transparent
- Once again, we set the cycle count to 2 and specified that the animation should auto-reverse itself.

22.7.2 TransitionAnimationsController Class (cont.)

RotateTransition

- Lines 56–61 configure a one-second `RotateTransition` that rotates a `Node`
- You can rotate- a `Node` by a specified number of degrees (line 58) or you can use other `RotateTransition` methods to specify a start angle and end angle
- Each `Transition` animation uses an `Interpolator` to calculate new property values throughout the animation's duration
- The default is a `LINEAR` `Interpolator` which evenly divides the property value changes over the animation's duration
- For the `RotateTransition`, line 60 uses the `Interpolator` `EASE_BOTH`, which changes the rotation slowly at first ("easing in"), speeds up the rotation in the middle of the animation, then slows the rotation again to complete the animation ("easing out")
- For a list of all the predefined `Interpolators`, see
  - `https://docs.oracle.com/javase/8/javafx/api/javafx/animation/Interpolator-.html`

22.7.2 `TransitionAnimationsController` Class (cont.)

`PathTransition`

- Lines 64–70 configure a two-second `PathTransition` that changes a shape's position by moving it along a `Path`
- Lines 64–65 create the `Path`, which is specified as the second argument to the `PathTransition` constructor
- A `LineTo` object draws a straight line from the previous `PathElement`'s endpoint to the specified location
- Line 69 specifies that this animation should use the `Interpolator` `EASE_IN`, which changes the position slowly at first, before performing the animation at full speed

22.7.2 `TransitionAnimationsController` Class (cont.)

`ScaleTransition`

- Lines 73–79 configure a one-second `ScaleTransition` that changes a `Node`'s size
- Line 75 specifies that the object will be scaled 75% larger along the *x*-axis (i.e., horizontally), and line 76 specifies that the object will be scaled 75% larger along the *y*-axis (i.e., vertically)
- Line 78 specifies that this animation should use the `Interpolator` `EASE_OUT`, which begins scaling the shape at full speed, then slows down as the animation completes

22.7.2 `TransitionAnimationsController` Class (cont.)

`SequentialTransition`

- Lines 82–86 configure a `SequentialTransition` that performs a sequence of transitions—as each completes, the next one in the sequence begins executing
- The `SequentialTransition` constructor receives the `Node` to which the sequence of animations will be applied, followed by a comma-separated list of `Transitions` to perform
- In fact, every transition animation class has a constructor that enables you to specify a `Node`
- For this example, we did not specify `Nodes` when creating the other transitions, because they're all applied by the `SequentialTransition` to the `Rectangle`
- Every `Transition` has a `play` method (line 86) that begins the animation
- Calling `play` on the `SequentialTransition` automatically calls `play` on each animation in the sequence.

22.8 Timeline Animations

- In this section, we continue our animation discussion with a `Timeline` animation that bounces a `Circle` object around the app's `Pane` over time
- A `Timeline` animation can change any `Node` property that's modifiable
- You specify how to change property values with one or more `KeyFrame` objects that the `Timeline` animation performs in sequence
- For this app, we'll specify a single `KeyFrame` that modifies a `Circle`'s location, then we'll play that `KeyFrame` indefinitely
- Figure 22.12 shows the app's FXML, which defines a `Circle` object with a five-pixel black border and the fill color `DODGERBLUE`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 22.12: TimelineAnimation.fxml -->
3  <!-- FXML for a Circle that will be animated by the controller -->
4
5  <?import javafx.scene.layout.Pane?>
6  <?import javafx.scene.shape.Circle?>
7
8  <Pane id="Pane" fx:id="pane" prefHeight="400.0"
9     prefWidth="600.0" xmlns:fx="http://javafx.com/fxml/1"
10    xmlns="http://javafx.com/javafx/8.0.60"
11    fx:controller="TimelineAnimationController">
12    <children>
13       <Circle fx:id="c" fill="DODGERBLUE" layoutX="142.0" layoutY="143.0"
14          radius="40.0" stroke="BLACK" strokeType="INSIDE"
15          strokeWidth="5.0" />
16    </children>
17 </Pane>
```

Fig. 22.12 | FXML for a `Circle` that will be animated by the controller.

22.8 `Timeline` Animations (cont.)

- The application's controller (Fig. 22.13) configures then plays the `Timeline` animation in the `initialize` method
- Lines 22–45 define the animation, line 48 specifies that the animation should cycle indefinitely (until the program terminates or the animation's `stop` method is called) and line 49 plays the animation

```
1  // Fig. 22.13: TimelineAnimationController.java
2  // Bounce a circle around a window using a Timeline animation
3  import java.security.SecureRandom;
4  import javafx.animation.KeyFrame;
5  import javafx.animation.Timeline;
6  import javafx.event.ActionEvent;
7  import javafx.event.EventHandler;
8  import javafx.fxml.FXML;
9  import javafx.geometry.Bounds;
10 import javafx.scene.layout.Pane;
11 import javafx.scene.shape.Circle;
12 import javafx.util.Duration;
13
```

Fig. 22.13 | Bounce a circle around a window using a `Timeline` animation. (Part 1 of 6.)

```
14  public class TimelineAnimationController {
15     @FXML Circle c;
16     @FXML Pane pane;
17
18     public void initialize() {
19        SecureRandom random = new SecureRandom();
20
21        // define a timeline animation
22        Timeline timelineAnimation = new Timeline(
23           new KeyFrame(Duration.millis(10),
24              new EventHandler<ActionEvent>() {
25                 int dx = 1 + random.nextInt(5);
26                 int dy = 1 + random.nextInt(5);
27
```

Fig. 22.13 | Bounce a circle around a window using a `Timeline` animation. (Part 2 of 6.)

```
28                  // move the circle by the dx and dy amounts
29                  @Override
30                  public void handle(final ActionEvent e) {
31                     c.setLayoutX(c.getLayoutX() + dx);
32                     c.setLayoutY(c.getLayoutY() + dy);
33                     Bounds bounds = pane.getBoundsInLocal();
34
35                     if (hitRightOrLeftEdge(bounds)) {
36                        dx *= -1;
37                     }
38
39                     if (hitTopOrBottom(bounds)) {
40                        dy *= -1;
41                     }
42                  }
43               }
44            )
45         );
46
47         // indicate that the timeline animation should run indefinitely
48         timelineAnimation.setCycleCount(Timeline.INDEFINITE);
49         timelineAnimation.play();
50      }
```

Fig. 22.13 | Bounce a circle around a window using a `Timeline` animation. (Part 3 of 6.)

```
51
52      // determines whether the circle hit the left or right of the window
53      private boolean hitRightOrLeftEdge(Bounds bounds) {
54         return (c.getLayoutX() <= (bounds.getMinX() + c.getRadius())) ||
55            (c.getLayoutX() >= (bounds.getMaxX() - c.getRadius()));
56      }
57
58      // determines whether the circle hit the top or bottom of the window
59      private boolean hitTopOrBottom(Bounds bounds) {
60         return (c.getLayoutY() <= (bounds.getMinY() + c.getRadius())) ||
61            (c.getLayoutY() >= (bounds.getMaxY() - c.getRadius()));
62      }
63   }
```

Fig. 22.13 | Bounce a circle around a window using a `Timeline` animation. (Part 4 of 6.)

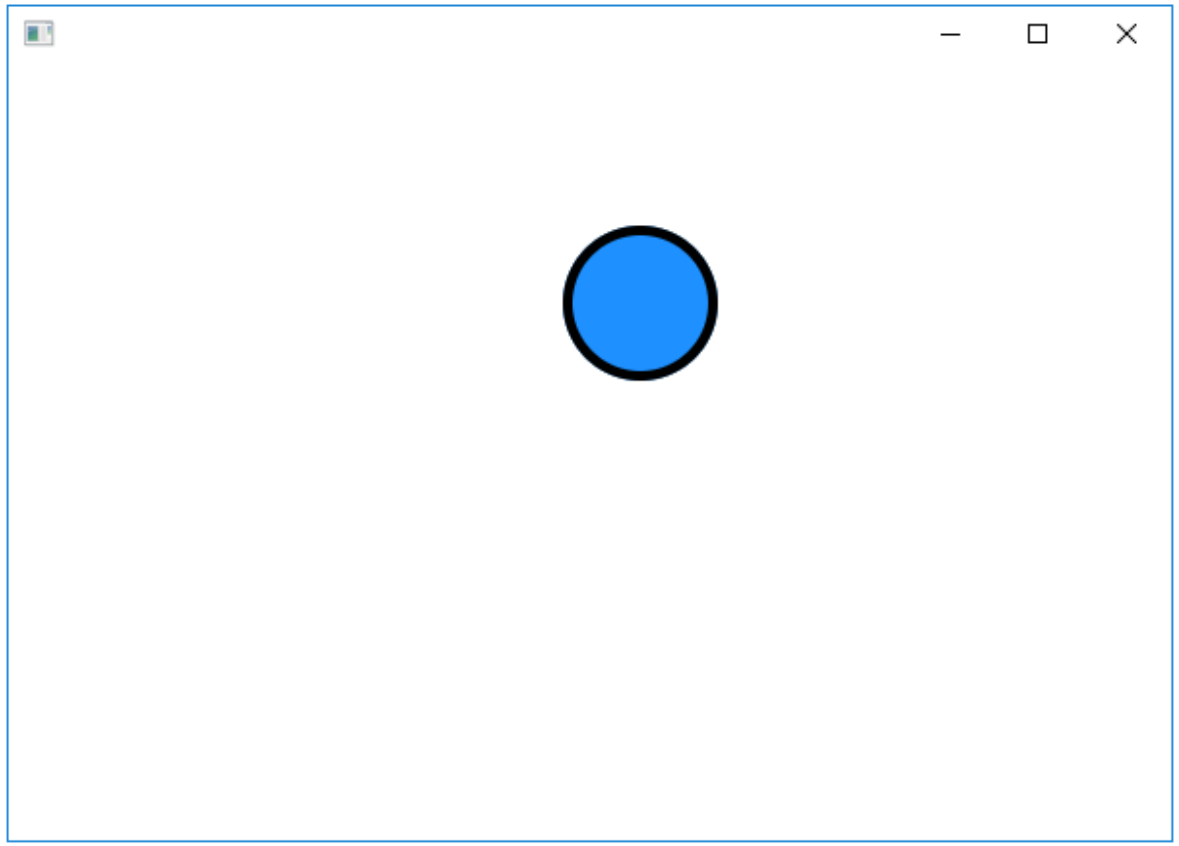


Fig. 22.13 | Bounce a circle around a window using a `Timeline` animation. (Part 5 of 6.)

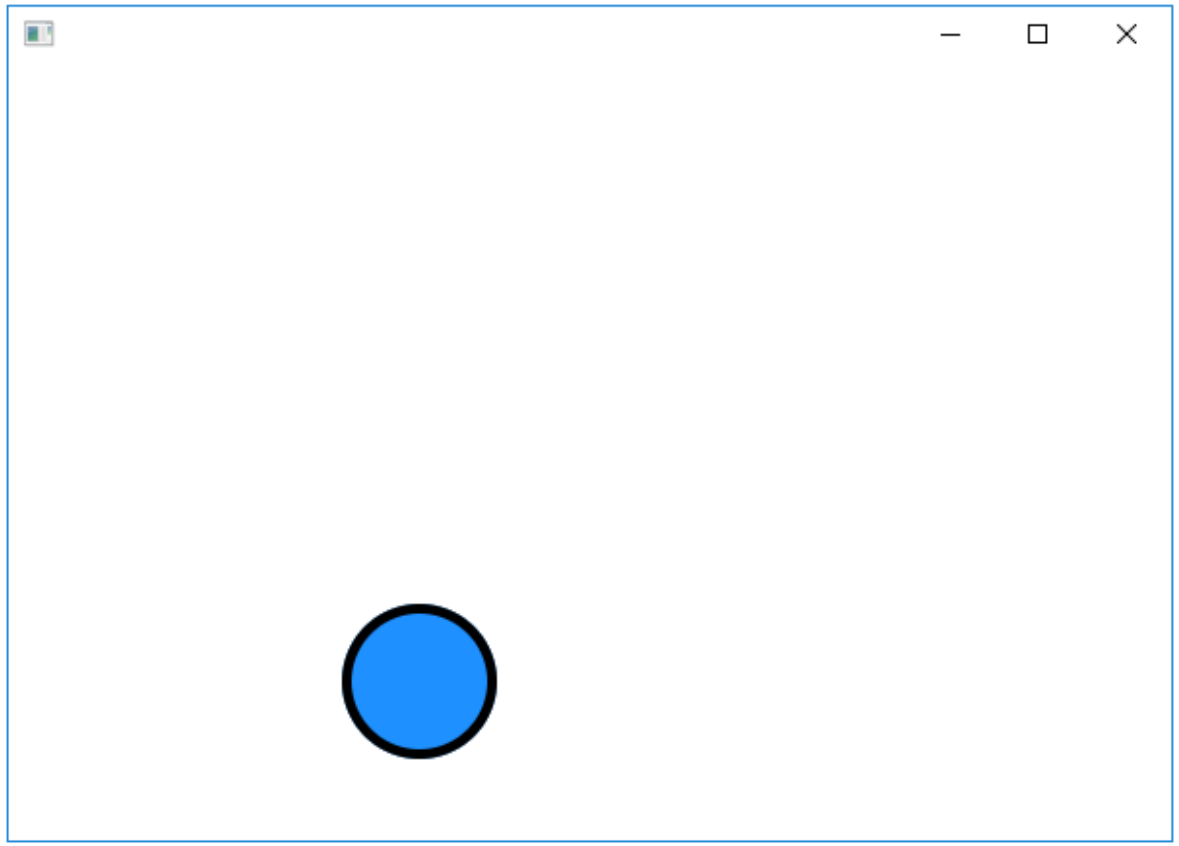


Fig. 22.13 | Bounce a circle around a window using a `Timeline` animation. (Part 6 of 6.)

22.8 `Timeline` Animations (cont.)

Creating the `Timeline`

- The `Timeline` constructor used in lines 22–45 can receive a comma-separated list of `KeyFrames` as arguments—in this case, a single `KeyFrame`
- Each `KeyFrame` issues an `ActionEvent` at a particular time in the animation
- The app can respond to the event by changing a `Node`'s property values
- The `KeyFrame` constructor used here specifies that, after 10 milliseconds, the `ActionEvent` will occur
- Because we set the `Timeline`'s cycle- count to `Timeline.INDEFINITE`, the `Timeline` will perform this `KeyFrame` every 10 milliseconds
- Lines 24–43 define the `EventHandler` for the `KeyFrame`'s `ActionEvent`

22.8 Timeline Animations (cont.)

KeyFrame's EventHandler

- In the `KeyFrame`'s `EventHandler` we define instance variables `dx` and `dy` (lines 25–26) and initialize them with randomly chosen values that will be used to change the `Circle`'s coordinates each time the `KeyFrame` plays
- The `EventHandler`'s `handle` method (lines 29–42) adds these values to the `Circle`'s *x*- and *y*-coordinates (lines 31–32)
- Next, lines 35–41 perform bounds checking to determine whether the `Circle` has collided with any of the `Pane`'s edges
- If the `Circle` hits the left or right edge, line 36 multiplies the value of `dx` by `-1` to reverse the `Circle`'s horizontal direction
- If the `Circle` hits the top or bottom edge, line 40 multiplies the value of `dx` by `-1` to reverse the `Circle`'s horizontal direction

22.9 Frame-by-Frame Animation with `AnimationTimer`

- A third way to implement JavaFX animations is via an `AnimationTimer` (package `javafx.animation`), which enables you to define frame-by-frame animations
- You specify how your objects should move in a given frame, then JavaFX aggregates all of the drawing operations and displays the frame
- This can be used with objects in the scene graph or to draw shapes in a `Canvas`
- JavaFX calls the `handle` method of every `AnimationTimer` before it draws an animation frame

22.9 Frame-by-Frame Animation with `AnimationTimer` (cont.)

- For smooth animation, JavaFX tries to display animation frames at 60 frames per second
- This frame rate varies based on the animation's complexity, the processor speed and how busy the processor is at a given time
- For this reason, method `handle` receives a time stamp in nanoseconds (billionths of a second) that you can use to determine the elapsed time since the last animation frame, then you can scale the movements of your objects accordingly
- This enables you to define animations that operate at the same overall speed, regardless of the frame rate on a given device.
- Figure 22.14 reimplements the animation in Fig. 22.13 using an `AnimationTimer`
- The FXML is identical (other than the filename and controller class name)
- Much of the code is identical to Fig. 22.13—we've highlighted the key changes, which we discuss below

```
1  // Fig. 22.14: BallAnimationTimerController.java
2  // Bounce a circle around a window using an AnimationTimer subclass.
3  import java.security.SecureRandom;
4  import javafx.animation.AnimationTimer;
5  import javafx.fxml.FXML;
6  import javafx.geometry.Bounds;
7  import javafx.scene.layout.Pane;
8  import javafx.scene.shape.Circle;
9  import javafx.util.Duration;
10
11 public class BallAnimationTimerController {
12    @FXML private Circle c;
13    @FXML private Pane pane;
14
```

Fig. 22.14 | Bounce a circle around a window using an `AnimationTimer` subclass. (Part 1 of 4.)

```
15      public void initialize() {
16         SecureRandom random = new SecureRandom();
17
18         // define a timeline animation
19         AnimationTimer timer = new AnimationTimer() {
20            int dx = 1 + random.nextInt(5);
21            int dy = 1 + random.nextInt(5);
22            int velocity = 60; // used to scale distance changes
23            long previousTime = System.nanoTime(); // time since app launch
24
25            // specify how to move Circle for current animation frame
26            @Override
27            public void handle(long now) {
28               double elapsedTime = (now - previousTime) / 1000000000.0;
29               previousTime = now;
30               double scale = elapsedTime * velocity;
```

Fig. 22.14 | Bounce a circle around a window using an `AnimationTimer` subclass. (Part 2 of 4.)

```
31
32                  Bounds bounds = pane.getBoundsInLocal();
33                  c.setLayoutX(c.getLayoutX() + dx * scale);
34                  c.setLayoutY(c.getLayoutY() + dy * scale);
35
36                  if (hitRightOrLeftEdge(bounds)) {
37                     dx *= -1;
38                  }
39
40                  if (hitTopOrBottom(bounds)) {
41                     dy *= -1;
42                  }
43               }
44            };
45
46            timer.start();
47      }
```

Fig. 22.14 | Bounce a circle around a window using an `AnimationTimer` subclass. (Part 3 of 4.)

```
48
49      // determines whether the circle hit left/right of the window
50      private boolean hitRightOrLeftEdge(Bounds bounds) {
51         return (c.getLayoutX() <= (bounds.getMinX() + c.getRadius())) ||
52            (c.getLayoutX() >= (bounds.getMaxX() - c.getRadius()));
53      }
54
55      // determines whether the circle hit top/bottom of the window
56      private boolean hitTopOrBottom(Bounds bounds) {
57         return (c.getLayoutY() <= (bounds.getMinY() + c.getRadius())) ||
58            (c.getLayoutY() >= (bounds.getMaxY() - c.getRadius()));
59      }
60   }
```

Fig. 22.14 | Bounce a circle around a window using an `AnimationTimer` subclass. (Part 4 of 4.)

22.9 Frame-by-Frame Animation with `AnimationTimer` (cont.)

Extending `abstract` Class `AnimationTimer`

- Class `AnimationTimer` is an `abstract` class, so you must create a subclass
  - As in Fig. 22.13, `dx` and `dy` incrementally change the `Circle`'s position and are chosen randomly so the `Circle` moves at different speeds during each execution.
  - Variable `velocity` is used as a multiplier to determine the actual distance moved in each animation frame—we discuss this again momentarily.
  - Variable `previousTime` represents the time stamp (in nanoseconds) of the previous animation frame—this will be used to determine the elapsed time between frames. We initialized `previousTime` to `System.nanoTime()`, which returns the number of nanoseconds since the JVM launched the app. Each call to `handle` also receives as its argument the number of nanoseconds since the JVM launched the app.

22.9 Frame-by-Frame Animation with `AnimationTimer` (cont.)

Overriding Method `handle`

- Lines 26–43 override `AnimationTimer` method `handle`, which specifies what to do during each animation frame:
  - Line 28 calculates the `elapsedTime` in *seconds* since the last animation frame. If method `handle` truly is called 60 times per second, the elapsed time between frames will be approximately 0.0167 seconds—that is, 1/60 of a second.
  - Line 29 stores the time stamp in `previousTime` for use in the *next* animation frame.

22.9 Frame-by-Frame Animation with `AnimationTimer` (cont.)

Overriding Method `handle`

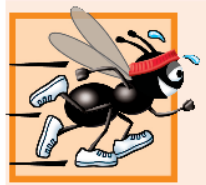
- When we change the `Circle`'s `layoutX` and `layoutY` (lines 33–34), we multiply `dx` and `dy` by the `scale` (line 30). In Fig. 22.13, the `Circle`'s speed was determined by moving between one and five pixels along the *x*- and *y*-axes every 10 milliseconds—the larger the values, the faster the `Circle` moved. If we scale `dx` or `dy` by just `elapsedTime`, we'd move the `Circle` only small fractions of `dx` and `dy` during each frame—approximately 0.0167 seconds (1/60 of a second) to 0.083 seconds (5/60 of a second), based on their randomly chosen values. For this reason, we multiply the `elapsedTime` by the `velocity` (60) to scale the movement in each frame. This results in values that are approximately one to five pixels, as in Fig. 22.13.

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`.
- A `GraphicsContext` object enables you to specify the same drawing characteristics that you've previously used on `Shape` objects

22.10 Drawing on a Canvas

- 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
```

Fig. 22.15 | Drawing on a Canvas. (Part 1 of 5.)

```
16  public class CanvasShapesController {
17     // instance variables that refer to GUI components
18     @FXML private Canvas drawingCanvas;
19
20     // draw on the Canvas
21     public void initialize() {
22        GraphicsContext gc = drawingCanvas.getGraphicsContext2D();
23        gc.setLineWidth(10); // set all stroke widths
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 transparent
31        gc.setLineCap(StrokeLineCap.ROUND);
32        gc.setStroke(Color.GREEN);
33        gc.strokeLine(100, 10, 10, 100);
34
```

Fig. 22.15 | Drawing on a Canvas. (Part 2 of 5.)

```
35          gc.setGlobalAlpha(1.0); // reset alpha transparency
36
37          // draw rounded rect with red border and yellow fill
38          gc.setStroke(Color.RED);
39          gc.setFill(Color.YELLOW);
40          gc.fillRoundRect(120, 10, 90, 90, 50, 50);
41          gc.strokeRoundRect(120, 10, 90, 90, 50, 50);
42
43          // draw circle with blue border and red/white radial gradient fill
44          gc.setStroke(Color.BLUE);
45          Stop[] stopsRadial =
46             {new Stop(0, Color.RED), new Stop(1, Color.WHITE)};
47          RadialGradient radialGradient = new RadialGradient(0, 0, 0.5, 0.5,
48             0.6, true, CycleMethod.NO_CYCLE, stopsRadial);
49          gc.setFill(radialGradient);
50          gc.fillOval(230, 10, 90, 90);
51          gc.strokeOval(230, 10, 90, 90);
52
```

Fig. 22.15 | Drawing on a Canvas. (Part 3 of 5.)

```
53          // draw ellipse with green border and image fill
54          gc.setStroke(Color.GREEN);
55          gc.setFill(new ImagePattern(new Image("yellowflowers.png")));
56          gc.fillOval(340, 10, 200, 90);
57          gc.strokeOval(340, 10, 200, 90);
58
59          // draw arc with purple border and cyan/white linear gradient fill
60          gc.setStroke(Color.PURPLE);
61          Stop[] stopsLinear =
62             {new Stop(0, Color.CYAN), new Stop(1, Color.WHITE)};
63          LinearGradient linearGradient = new LinearGradient(0, 0, 1, 0,
64             true, CycleMethod.NO_CYCLE, stopsLinear);
65          gc.setFill(linearGradient);
66          gc.fillArc(560, 10, 90, 90, 45, 270, ArcType.ROUND);
67          gc.strokeArc(560, 10, 90, 90, 45, 270, ArcType.ROUND);
68       }
69    }
```

Fig. 22.15 | Drawing on a Canvas. (Part 4 of 5.)

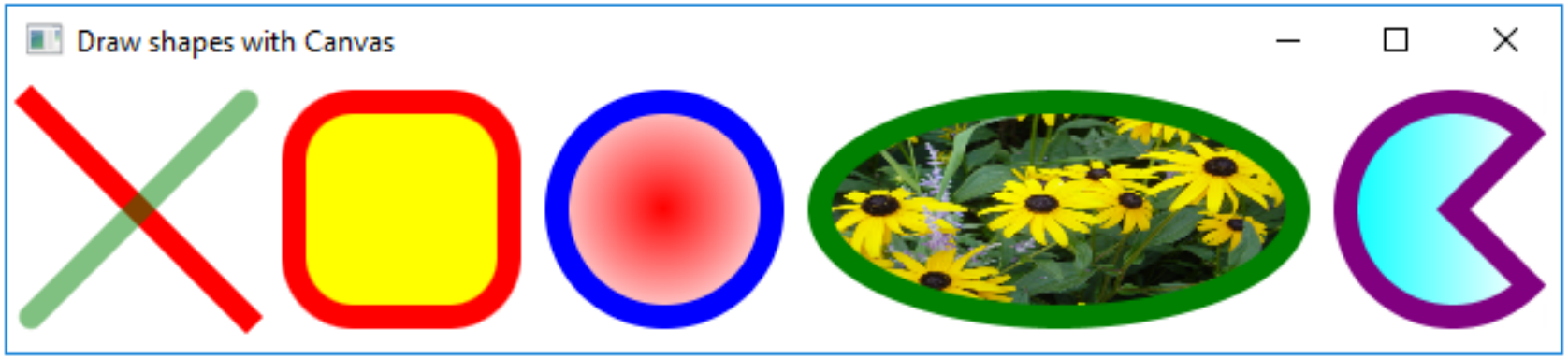


Fig. 22.15 | Drawing on a **Canvas**. (Part 5 of 5.)

22.10 Drawing on a Canvas (cont.)

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-stroke-width` CSS attribute we specified for all shapes in Fig. 22.4.

22.10 Drawing on a Canvas (cont.)

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.

22.10 Drawing on a Canvas (cont.)

Drawing Lines

- `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.

22.10 Drawing on a Canvas (cont.)

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.

22.10 Drawing on a Canvas (cont.)

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 `Stops` indicate that the radial gradient will transition from red at the gradient's start point to white at its end point

22.10 Drawing on a Canvas (cont.)

- 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

22.10 Drawing on a Canvas (cont.)

- 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.

22.10 Drawing on a Canvas (cont.)

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.

22.10 Drawing on a Canvas (cont.)

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.

22.10 Drawing on a Canvas (cont.)

- 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`)

22.10 Drawing on a Canvas (cont.)

Additional `GraphicsContext` *Features*

- There are many additional `GraphicsContext` features, which you can explore at
  - `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/canvas/GraphicsContext.html`
- 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.

22.11 Three-Dimensional Shapes

- Throughout this chapter, we've demonstrated many two-dimensional graphics capabilities
- In Java SE 8, JavaFX added several three-dimensional shapes and corresponding capabilities
- The three-dimensional shapes are subclasses of `Shape3D` from the package `javafx.scene.shape`
- In this section, you'll use Scene Builder to create a `Box`, a `Cylinder` and a `Sphere` and specify several of their properties
- Then, in the app's controller, you'll create so-called *materials* that apply color and images to the 3D shapes.

22.11 Three-Dimensional Shapes (cont.)

***FXML for the* `Box`, `Cylinder` *and* `Sphere`**

- Figure 22.16 shows the completed FXML that we created with Scene Builder:
  - Lines 16–21 define the `Box` object.
  - Lines 22–27 define the `Cylinder` object.
  - Lines 28–29 define the `Sphere` object.
- We dragged objects of each of these `Shape3D` subclasses from the Scene Builder Library's Shapes section onto the design area and gave them the fx:id values `box`, `cylinder` and `sphere`, respectively
- We also set the controller to `ThreeDimensionalShapesController`.

22.11 Three-Dimensional Shapes (cont.)

- As you can see in the screen capture of Figure 22.16, all three shapes initially are gray
- The shading you see in Scene Builder comes from the scene's default lighting
- Though we do not use them in this example, package `javafx.scene`'s `AmbientLight` and `PointLight` classes can be used to add your own lighting effects
- You can also use camera objects to view the scene from different angles and distances
- These are located in the Scene Builder Library's 3D section
- For more information on lighting and cameras, see
  - `https://docs.oracle.com/javase/8/javafx/graphics-tutorial/javafx-3d-graphics.htm`

22.11 Three-Dimensional Shapes (cont.)

Box ***Properties***

- Configure the Box's properties in Scene Builder as follows:
  - Set Width, Height and Depth to `100`, making a cube. The depth is measured along the *z*-axis which runs perpendicular to your screen—when you move objects along the *z*-axis they get bigger as they're brought toward you and smaller as they're moved away from you.
  - Set Layout X and Layout Y to `100` to specify the location of the cube.
  - Set Rotate to `30` to specify the rotation angle in degrees. Positive values rotate counter-clockwise.
  - For Rotation Axis, set the X, Y and Z values to `1` to indicate that the Rotate angle should be used to rotate the cube 30 degrees around *each* axis.
- To see how the Rotate angle and Rotation Axis values affect the Box's rotation, try setting two of the three Rotation Axis values to `0`, then changing the Rotate angle.

22.11 Three-Dimensional Shapes (cont.)

Cylinder Properties

- Configure the `Cylinder`'s properties in Scene Builder as follows:
  - Set Height to `100.0` and Radius to `50`.
  - Set Layout X and Layout Y to `265` and `100`, respectively.
  - Set Rotate to `-45` to specify the rotation angle in degrees. Negative values rotate clockwise.
  - For Rotation Axis, set the X, Y and Z values to `1` to indicate that the Rotate angle should be applied to all three axes.

22.11 Three-Dimensional Shapes (cont.)

Sphere ***Properties***

- Configure the `Sphere`'s properties in Scene Builder as follows:
  - Set Radius to `60`.
  - Set Layout `X` and Layout `Y` to `430` and `100`, respectively.

22.11 Three-Dimensional Shapes (cont.)

ThreeDimensionalShapesController ***Class***

- Figure 22.17 shows this app's controller and final output. The colors and images you see on the final shapes are created by applying so-called materials to the shapes
- JavaFX class `PhongMaterial` (package `javafx.scene.paint`) is used to define materials
- The name "Phong" is a 3D graphics term—*phong shading* is technique for applying color and shading to 3D surfaces
- For more details on this technique, visit
  - https://en.wikipedia.org/wiki/Phong_shading

22.11 Three-Dimensional Shapes (cont.)

*`PhongMaterial` **for the** `Box`*

- Lines 20–22 configure and set the `Box` object's `PhongMaterial`
- Method `setDiffuseColor` sets the color that's applied to the `Box`'s surfaces (that is, sides)
- The scene's lighting effects determine the shades of the color applied to each visible surface
- These shades change, based on the angle from which the light shines on the objects

22.11 Three-Dimensional Shapes (cont.)

*`PhongMaterial` **for the** `Cylinder`*

- Lines 25–27 configure and set the `Cylinder` object's `PhongMaterial`
- Method `setDiffuseMap` sets the `Image` that's applied to the `Cylinder`'s surfaces
- Again, the scene's lighting affects how the image is shaded on the surfaces
- In the output, notice that the image is darker at the left and right edges (where less light reaches) and barely visible on the bottom (where almost no light reaches).

22.11 Three-Dimensional Shapes (cont.)

*`PhongMaterial` **for the** `Sphere`*

- Lines 30–34 configure and set the `Sphere` object's `PhongMaterial`
- We set the diffuse color to red
- Method `setSpecularColor` sets the color of a bright spot that makes a 3D shape appear shiny
- Method `setSpecularPower` determines the intensity of that spot
- Try experimenting with different specular powers to see changes in the bright spot's intensity.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- ThreeDimensionalShapes.fxml -->
3  <!-- FXML that displays a Box, Cylinder and Sphere -->
4
5  <?import javafx.geometry.Point3D?>
6  <?import javafx.scene.layout.Pane?>
7  <?import javafx.scene.shape.Box?>
8  <?import javafx.scene.shape.Cylinder?>
9  <?import javafx.scene.shape.Sphere?>
10
11 <Pane prefHeight="200.0" prefWidth="510.0"
12    xmlns="http://javafx.com/javafx/8.0.60"
13    xmlns:fx="http://javafx.com/fxml/1"
14    fx:controller="ThreeDimensionalShapesController">
15    <children>
16       <Box fx:id="box" depth="100.0" height="100.0" layoutX="100.0"
17          layoutY="100.0" rotate="30.0" width="100.0">
18          <rotationAxis>
19             <Point3D x="1.0" y="1.0" z="1.0" />
20          </rotationAxis>
21       </Box>
```

Fig. 22.16 | FXML that displays a `Box`, `Cylinder` and `Sphere`. (Part 1 of 2.)

```
22          <Cylinder fx:id="cylinder" height="100.0" layoutX="265.0"
23             layoutY="100.0" radius="50.0" rotate="-45.0">
24             <rotationAxis>
25                <Point3D x="1.0" y="1.0" z="1.0" />
26             </rotationAxis>
27          </Cylinder>
28          <Sphere fx:id="sphere" layoutX="430.0" layoutY="100.0"
29             radius="60.0" />
30       </children>
31    </Pane>
```

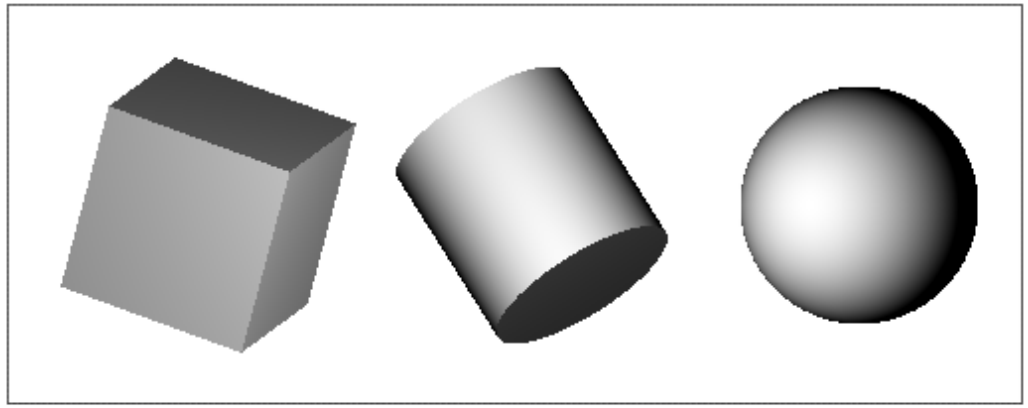


Fig. 22.16 | FXML that displays a **Box**, **Cylinder** and **Sphere**. (Part 2 of 2.)

```
1  // Fig. 22.17: ThreeDimensionalShapesController.java
2  // Setting the material displayed on 3D shapes.
3  import javafx.fxml.FXML;
4  import javafx.scene.paint.Color;
5  import javafx.scene.paint.PhongMaterial;
6  import javafx.scene.image.Image;
7  import javafx.scene.shape.Box;
8  import javafx.scene.shape.Cylinder;
9  import javafx.scene.shape.Sphere;
10
11 public class ThreeDimensionalShapesController {
12    // instance variables that refer to 3D shapes
13    @FXML private Box box;
14    @FXML private Cylinder cylinder;
15    @FXML private Sphere sphere;
16
```

Fig. 22.17 | Setting the material displayed on 3D shapes. (Part 1 of 3.)

```
17      // set the material for each 3D shape
18      public void initialize() {
19          // define material for the Box object
20          PhongMaterial boxMaterial = new PhongMaterial();
21          boxMaterial.setDiffuseColor(Color.CYAN);
22          box.setMaterial(boxMaterial);
23
24          // define material for the Cylinder object
25          PhongMaterial cylinderMaterial = new PhongMaterial();
26          cylinderMaterial.setDiffuseMap(new Image("yellowflowers.png"));
27          cylinder.setMaterial(cylinderMaterial);
28
29          // define material for the Sphere object
30          PhongMaterial sphereMaterial = new PhongMaterial();
31          sphereMaterial.setDiffuseColor(Color.RED);
32          sphereMaterial.setSpecularColor(Color.WHITE);
33          sphereMaterial.setSpecularPower(32);
34          sphere.setMaterial(sphereMaterial);
35      }
36  }
```

Fig. 22.17 | Setting the material displayed on 3D shapes. (Part 2 of 3.)

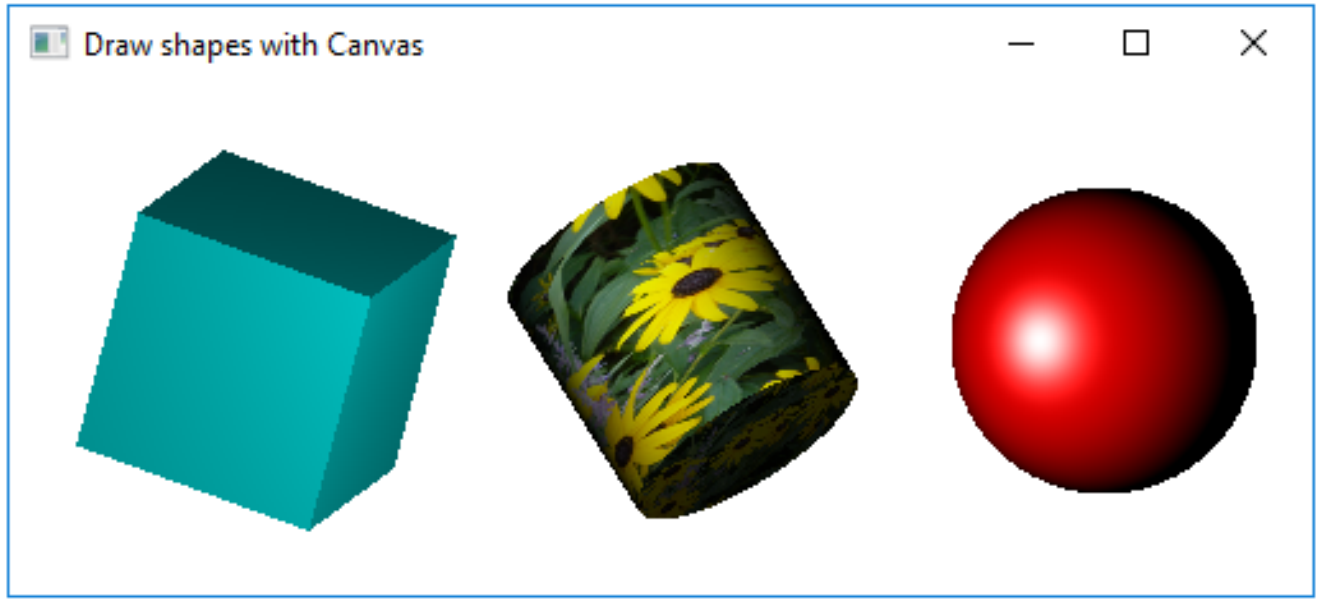


Fig. 22.17 | Setting the material displayed on 3D shapes. (Part 3 of 3.)