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 from the file `TransitionAnimations.css`:

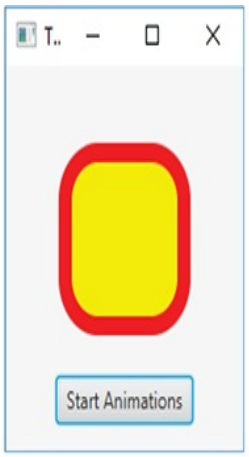
```
Rectangle {
   -fx-stroke-width: 10;
   -fx-stroke: red;
   -fx-arc-width: 50;
   -fx-arc-height: 50;
   -fx-fill: yellow;
}
```

which produces a rounded rectangle with a 10-pixel red border and yellow fill.

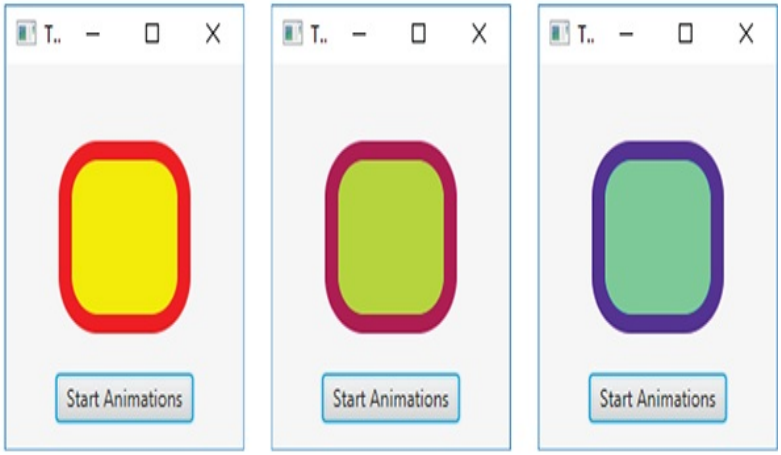
```
    1   <?xml version="1.0" encoding="UTF-8"?>
2   <!-- Fig. 22.10: TransitionAnimations.fxml -->
   3   <!-- FXML for a Rectangle and Button -->
                        4
    5   <?import javafx.scene.control.Button?>
      6   <?import javafx.scene.layout.Pane?>
    7   <?import javafx.scene.shape.Rectangle?>
                        8
9   <Pane id="Pane" prefHeight="200.0" prefWidth="180
 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"
   16            layoutY="45.0" width="90.0" />
17         <Button fx:id="startButton" layoutX="38.0"
```

```
        18            mnemonicParsing="false"
   19           onAction="#startButtonPressed" text="St
                 20      </children>
                     21   </Pane>
```

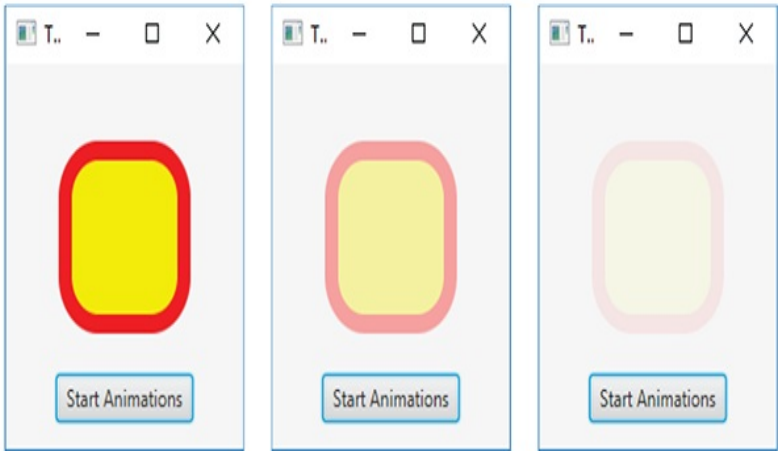
a) Initial `Rectangle`



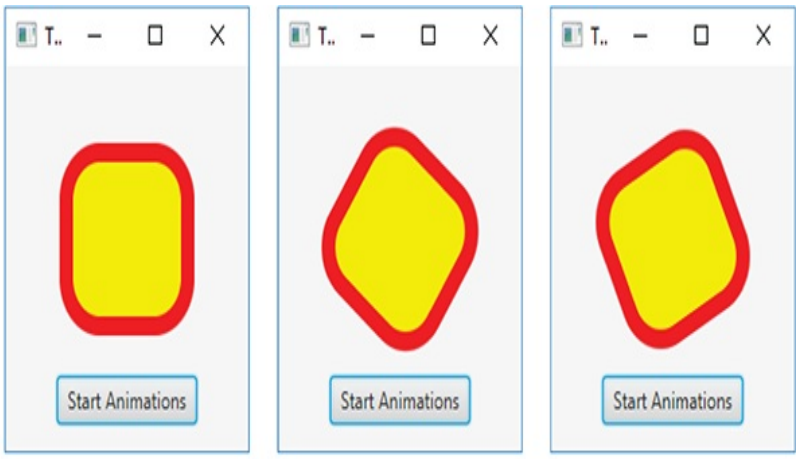
b) `Rectangle` undergoing parallel fill and stroke transitions



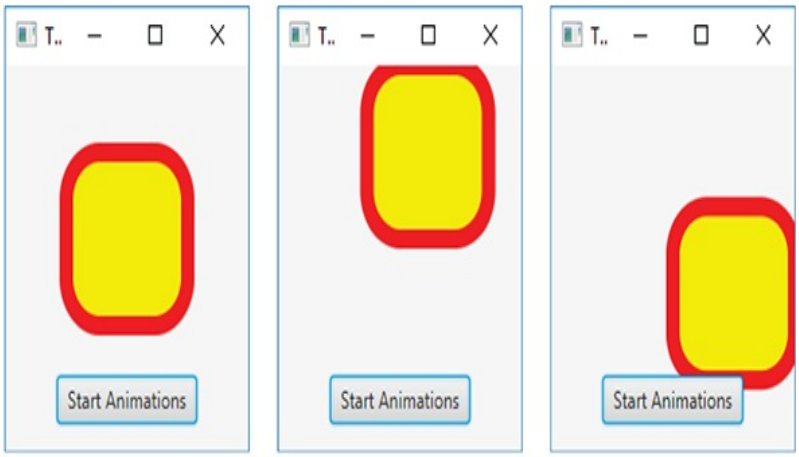
c) `Rectangle` undergoing a fade transition



d) `Rectangle` undergoing a rotate transition



e) `Rectangle` undergoing a path transition



f) `Rectangle` undergoing a scale transition

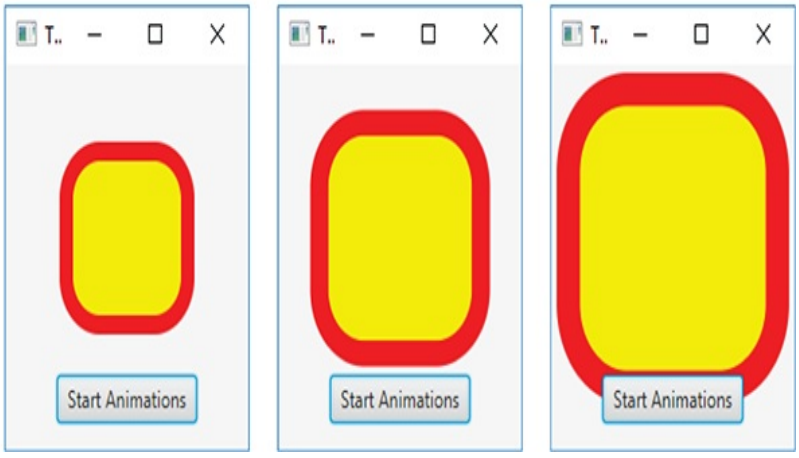


Fig. 22.10

FXML for a `Rectangle` and `Button`.

Description

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.jav
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;
20
21  public class TransitionAnimationsController {
```

```
    @FXML private Rectangle rectangle;

// configure and start transition animations
    @FXML
private void startButtonPressed(ActionEvent e
    // transition that changes a shape's fill
        FillTransition fillTransition =
        new FillTransition(Duration.seconds(1))
    fillTransition.setToValue(Color.CYAN);
        fillTransition.setCycleCount(2);

    // each even cycle plays transition in rev
        fillTransition.setAutoReverse(true);

    // transition that changes a shape's strok
        StrokeTransition strokeTransition =
        new StrokeTransition(Duration.seconds(1
    strokeTransition.setToValue(Color.BLUE);
        strokeTransition.setCycleCount(2);
    strokeTransition.setAutoReverse(true);

    // parallelizes multiple transitions
    ParallelTransition parallelTransition =
        new ParallelTransition(fillTransition,

    // transition that changes a node's opacit
        FadeTransition fadeTransition =
        new FadeTransition(Duration.seconds(1))
    fadeTransition.setFromValue(1.0); // opaqu
    fadeTransition.setToValue(0.0); // transpa
        fadeTransition.setCycleCount(2);
    fadeTransition.setAutoReverse(true);

        // transition that rotates a node
        RotateTransition rotateTransition =
        new RotateTransition(Duration.seconds(1
        rotateTransition.setByAngle(360.0);
        rotateTransition.setCycleCount(2);
    rotateTransition.setInterpolator(Interpola
    rotateTransition.setAutoReverse(true);
```

```
                        62
63          // transition that moves a node along a Pa
64          Path path = new Path(new MoveTo(45, 45), n
65             new LineTo(90, 0), new LineTo(90, 90),
 66          PathTransition translateTransition =
67             new PathTransition(Duration.seconds(2),
 68          translateTransition.setCycleCount(2);
69          translateTransition.setInterpolator(Interp
70          translateTransition.setAutoReverse(true);
                        71
72          // transition that scales a shape to make
   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(Interpolat
 79          scaleTransition.setAutoReverse(true);
                        80
81          // transition that applies a sequence of t
82          SequentialTransition sequentialTransition
83             new SequentialTransition (rectangle, pa
84                fadeTransition, rotateTransition, tr
        85                scaleTransition);
86          sequentialTransition.play(); // play the t
                    87      }
                      88   }
```

Fig. 22.11

Applying `Transition` animations to a `Rectangle`.

`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.

`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.

`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 `Transition`s as a comma-separated list. In this case, the `FillTransition` and `StrokeTransition` will be performed in parallel on the app's `Rectangle`.

`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.

`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 (known as "easing in"), speeds up the rotation in the middle of the animation, then slows the rotation again to complete the

animation (known as “easing out”). For a list of all the predefined `Interpolator`s, see

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

`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.

`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.

`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 `Transition`s 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 `Node`s 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.