Chapter 17

Animating Your Scenes

In This Chapter

- Looking at the various ways to create JavaFX animations
- Using JavaFX transitions
- Using the `KeyFrame` and `Timeline` classes
- Creating bouncing balls

You can go a long way toward improving the look and feel of your applications by applying special effects as described in Chapter 14, incorporating property bindings to make your controls more responsive, and using sound and media to provide audio and visual interest. In this chapter, I discuss how to take your applications one step further by incorporating simple animation effects. The effects you read about in this chapter make your applications come alive by enabling objects on the screen to move.

Please don't get your hopes set on winning an Oscar for Best Animation next year. No one will be fooled into thinking that you collaborated with Pixar on your application. Still, you can add some interesting whiz-bang to your applications using these techniques.

Introducing JavaFX Animation

The basic idea of JavaFX animations is to manipulate the value of one or more node properties at regular intervals. For example, suppose you have a circle that represents a ball and you want to move it from the left side of the screen to the right. Assuming the width of the screen is 600 pixels, you'd animate the circle by varying its `posX` property from 0 to 600.

Two factors will affect how fast the ball moves across the screen: the amount of time that elapses between each change to the `posX` property, and the increment you add to the `posX` property at each time interval. For example, if

you add 3 to the `posX` property at each time interval, it will take 200 intervals to get to 600 (3 x 200 = 600). If the intervals occur every 10 milliseconds (100 times per second), it will take 2,000 milliseconds — 2 full seconds — for the ball to traverse the screen from left to right.

Without JavaFX animations, you could implement the moving ball by using the Java `Timer` object to move the ball at regular intervals. The `Timer` class can be difficult to set up and use correctly. The JavaFX animation classes make animating your nodes a much simpler proposition.

JavaFX provides two basic ways to create animations, which I refer to in this chapter using extremely technical terms — the *hard way* and the *easy way:*

- The **hard way** requires that you set up timer events manually, and then write event listeners that are called when the timer events occur. In the event listeners, you manipulate the properties of the nodes you want to animate. For example, to move a ball across the screen, you'd set up a timer interval that ticks every 10 milliseconds. At each tick, you'd increase the `x` position of the ball by 3. You'd then set the timer to run a total of 200 times to move the ball. Setting up this animation requires that you use two classes: `Timeline` and `KeyFrame`, and that you write an `ActionEvent` listener to move the ball.
- The **easy way** takes advantage of shortcut classes provided by JavaFX to easily implement common types of animations. For example, you can use the `TranslateTransition` class to easily move a circle from one side of the screen to the other over a specified period of time. You just set up a `TranslateTransition` specifying that you want to vary the ball's `x` position from 0 to 600 over the course of 2 seconds. The `TranslateTransition` class will take care of the details.

In the remainder of this chapter, you discover both the easy way and the hard way. I start with the easy way.

Using Transition Classes

JavaFX comes with eight predefined animation effects — dubbed *transition classes* — that you can use to easily create an animation on most any node in your scene graph. (Most of the transitions will work with any node, but some will work only on shapes.) The eight transition types are

- **`FadeTransition`:** Varies the opacity value of any node. You can use this transition to fade an object in or out. Or, you can use it to make an object "wink" by quickly fading it out and then back in. You can also use it to create a flashing light that repeatedly fades in and then out.
- **`FillTransition`:** Varies the color of a shape's fill from a starting color to an ending color. For example, you can make a circle change from red to green.
- **`PathTranslation`:** Causes a shape to move along a predefined path. You can use any shape for the path.
- **`PauseTransition`:** This handy transition simply pauses for a moment; it's often used between two transitions to cause a break in the action.
- **`RotateTransition`:** Causes a node to rotate.
- **`ScaleTransition`:** Causes an object to increase or decrease in size.
- **`StrokeTransition`:** Varies the color used for a shape's outline stroke.
- **`TranslateTransition`:** Moves a node by translating it from one location to another.

These eight transition classes are all subclasses of the `Transition` class, which is in turn a subclass of the `Animation` class. Table 17-1 lists the methods that are defined by the `Transition` and `Animation` classes, and are therefore available to all transition classes.

Table 17-1 Methods of the Transition and Animation Classes

Method	*Explanation*
`void play()`	Plays the animation from its current position.
`void playFromStart()`	Plays the animation from the start.
`void pause()`	Temporarily suspends the animation. You can start it again by calling `play`.
`void stop()`	Stops the animation.
`void setCycleCount(int value)`	Sets the number of times the animation should repeat. To repeat the animation an indefinite number of times, specify `Animation.INDEFINITE`.

(continued)

Table 17-1 *(continued)*

Method	*Explanation*
`setAutoReverse(boolean value)`	If `true`, the animation reverses direction each time the cycle is repeated.
`setInterpolator(Interpolator value)`	Determines the method used to calculate the intermediate values of the property controlled by the transition. The possible values are `Interpolator.DISCRETE` `Interpolator.LINEAR` `Interpolator.EASE_IN` `Interpolator.EASE_OUT` `Interpolator.EASE_BOTH` The default setting is `EASE_BOTH`.

Most of the methods in Table 17-1 are straightforward, but the `setInterpolator` method merits a bit of explanation. The *interpolator* is the method used to calculate the intermediate values of the property being controlled by the transition. For example, in a `FadeTransition`, the interpolator determines how the value of the node's opacity is varied during the time that the animation is running; for a `TranslateTransition`, the interpolator determines how the x- and y-coordinates change during the animation.

The default interpolator setting is `Interpolator.EASE_BOTH`, which means that the change begins slowly, then speeds up though the middle of the animation, then slows down again just before the animation ends. For a `TranslateTransition`, this causes the movement of the node to start slowly, speed up, and then slow down toward the end.

The `EASE_IN` interpolator speeds up at the beginning but ends abruptly, while the `EASE_OUT` interpolator starts abruptly but slows down at the end. The `LINEAR` interpolator varies the property controlled by the transition at a constant rate throughout the animation. And the `DISCRETE` interpolator doesn't change the property value at all until the end of the animation has been reached; then, it immediately changes to the ending value.

Table 17-2 lists the most commonly used constructors and methods for each of the transition types.

Table 17-2 Transition Classes

FadeTransition	*Explanation*
`FadeTransition(Duration duration, Node node)`	Creates a fade transition of the given duration for the specified node.
`void setFromValue(Double value)`	Sets the starting opacity for the fade transition.
`void setToValue(Double value)`	Sets the ending opacity for the fade transition.
`void setByValue(Double value)`	If the ending opacity is not specified, this value is added to the starting value to determine the ending value.
FillTransition	***Explanation***
`FillTransition(Duration duration, Shape shape)`	Creates a fill transition of the given duration for the specified shape.
`void setFromValue(Color value)`	Sets the starting color for the fade transition.
`void setToValue(Color value)`	Sets the ending color for the fade transition.
PathTransition	***Explanation***
`PathTransition(Duration duration, Shape path, Shape shape)`	Creates a path transition of the given duration. The path translation causes the specified shape to travel along the specified path.
`void setFromValue(Color value)`	Sets the starting color for the fade transition.
`void setToValue(Color value)`	Sets the ending color for the fade transition.
PauseTransition	***Explanation***
`PauseTransition(Duration duration)`	Causes a delay of the specified duration.
RotateTransition	***Explanation***
`RotateTransition(Duration duration, Node node)`	Creates a rotate transition of the given duration on the specified node.
`void setFromAngle(Double value)`	Sets the starting angle for the rotation.

(continued)

Table 17-2 *(continued)*

RotateTransition	***Explanation***
`void setToAngle(Double value)`	Sets the ending angle for the rotation.
`void setByAngle(Double value)`	If the ending angle is not specified, this value is added to the starting angle to determine the ending angle.
ScaleTransition	***Explanation***
`ScaleTransition(Duration duration, Node node)`	Creates a scale transition of the given duration on the specified node.
`void setFromX(Double value)`	Sets the starting scale for the x-axis.
`void setFromY(Double value)`	Sets the starting scale for the y-axis.
`void setFromZ(Double value)`	Sets the starting scale for the z-axis.
`void setToX(Double value)`	Sets the ending scale for the x-axis.
`void setToY(Double value)`	Sets the ending scale for the y-axis.
`void setToZ(Double value)`	Sets the ending scale for the z-axis.
`void setByX(Double value)`	Sets the increment scale for the x-axis.
`void setByY(Double value)`	Sets the increment scale for the y-axis.
`void setByZ(Double value)`	Sets the increment scale for the z-axis.
StrokeTransition	***Explanation***
`StrokeTransition(Duration duration, Shape shape)`	Creates a stroke transition of the given duration for the specified shape.
`void setFromValue(Color value)`	Sets the starting color for the stroke transition.
`void setToValue(Color value)`	Sets the ending color for the stroke transition.

TranslateTransition	*Explanation*
`TranslateTransition(Duration duration, Node node)`	Creates a translate transition of the given duration on the specified node.
`void setFromX(Double value)`	Sets the starting point for the x-axis.
`void setFromY(Double value)`	Sets the starting point for the y-axis.
`void setFromZ(Double value)`	Sets the starting point for the z-axis.
`void setToX(Double value)`	Sets the ending point for the x-axis.
`void setToY(Double value)`	Sets the ending point for the y-axis.
`void setToZ(Double value)`	Sets the ending point for the z-axis.
`void setByX(Double value)`	Sets the increment point for the x-axis.
`void setByY(Double value)`	Sets the increment point for the y-axis.
`void setByZ(Double value)`	Sets the increment point for the z-axis.

Setting properties by, from, or to?

The `setFrom`, `setTo`, and `setBy` methods that appear in several of the transition classes listed in Table 17-2 deserve a little explanation.

By default, both the start and end values of a transition are the node's current values for the property being animated. Thus, the default starting and ending locations for a `TranslateTransition` are the node's current `x` and `y` positions.

Here are several ways to specify a different starting value, ending value, or both:

- Let the node's current values stand as the starting values and specify a new ending value by using the `setTo` methods.
- Let the node's current values stand as the starting values and specify a displacement to the ending value by using a `setBy` method. The value you specify in the `setBy` method will be added to the starting value to determine the ending value.
- Use the `setFrom` location to change the starting location from the node's current value. Then, omit both the `setTo` and `setBy` values to let the node's current value be the ending value.
- Use the `setFrom` location to change the starting location from the node's current value and use `setTo` to set the ending value.
- Use the `setFrom` location to change the starting location from the node's current value and use `setBy` to set a displacement value that will be added to the starting location to determine the ending location.

Looking at a Transition Example

The nine transitions listed in Table 17-2 all work essentially the same. To use any of them, you simply create the transition by calling its constructor and specifying the duration of the transition and the node you want animated. Then, if necessary, you call additional methods to set the transition parameters. Finally, you call the `play` method to start the animation.

The following example shows a transition that moves a circle named `c` from the top-left corner of the scene to location 300, 300:

```
TranslateTransition t = new TranslateTransition(
    Duration.millis(2000), c);
t.setFromX(0);
t.setFromY(0);
t.setToX(300);
t.setToY(300);
t.play();
```

Here, the duration of the animation is set to 2 seconds (2,000 milliseconds).

Listing 17-1 shows how this transition can be incorporated into a complete program. As you can see, the program is short. It simply displays a red ball at the left edge of the scene and then moves the ball to the right edge of the scene. The transition's cycle count is set to indefinite, and the autoreverse setting is set to true. As a result, the animation repeats itself indefinitely, giving the appearance that the ball is bouncing back and forth between the right and left edges of the screen.

Listing 17-1: The BouncingBall Program

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.shape.*;
import javafx.scene.paint.*;
import javafx.animation.*;
import javafx.util.*;

public class BouncingBall extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
```

```
    @Override public void start(Stage primaryStage)
    {

        RadialGradient g = new RadialGradient(                          →20
            0, 0,
            0.35, 0.35,
            0.5,
            true,
            CycleMethod.NO_CYCLE,
            new Stop(0.0, Color.WHITE),
            new Stop(1.0, Color.RED));

        Circle ball = new Circle(0,0,20);                               →29
        ball.setFill(g);

        Group root = new Group();                                       →32
        root.getChildren().add(ball);
        Scene scene = new Scene(root, 600, 600);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Bouncing Ball");
        primaryStage.show();

        TranslateTransition t = new TranslateTransition(                →39
            Duration.millis(2000), ball);
        t.setFromX(ball.getRadius());                                   →41
        t.setToX(scene.getWidth() - ball.getRadius());                  →42
        t.setFromY(scene.getHeight() / 2);                              →43
        t.setToY(scene.getHeight() / 2);
        t.setCycleCount(Transition.INDEFINITE);                         →45
        t.setAutoReverse(true);                                         →46
        t.setInterpolator(Interpolator.LINEAR);                         →47
        t.play();                                                       →48
    }

}
```

The following paragraphs highlight the key points in this program:

- → **20:** A radial gradient is created to give the ball a three-dimensional appearance.
- → **29:** The circle is created. Its radius is 20 pixels, and its fill is the gradient created in line 20.
- → **32:** A group object is created to serve as the root node for the scene. Then, the ball is added to the group and the group is used to create and display a scene.

→ **39:** A `TranslateTransition` is created to translate the ball. The duration is set to 2 seconds.

→ **41:** The `fromX` property is set to the radius of the ball. That positions the ball with its left edge on the left edge of the scene.

→ **42:** The `toX` property is set to the width of the screen less the radius of the ball. This positions the ball at the right edge of the screen at the end of the animation cycle.

→ **43:** The `fromY` and `toY` properties are set to half the height of the scene. That way, the ball will travel along a horizontal path centered vertically in the scene.

→ **45:** The cycle count is set to `INDEFINITE` so the ball will bounce forever.

→ **46:** `AutoReverse` is set to `true` so that each cycle will reverse the direction of the ball's travel.

→ **47:** The interpolator is set to linear so that the ball does not slow down at as it approaches the edges of the scene.

→ **48:** Play!

Figure 17-1 shows the Bouncing Ball program in action.

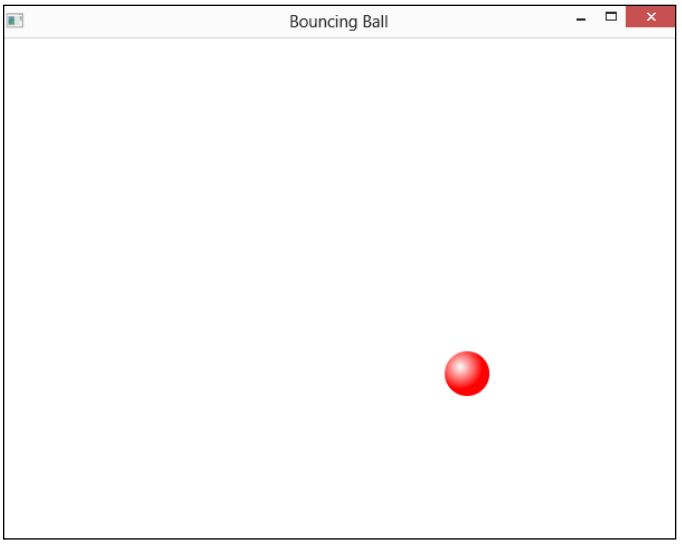

Figure 17-1: The Bouncing Ball program in action.

Combining Transitions

JavaFX provides two transition classes that are designed to let you combine transitions so that two or more transitions run one after the other or at the same time. The `SequentialTransition` class lets you run several transitions one after the other, whereas the `ParallelTransition` class lets you run several transitions at once.

Both classes have simple constructors that accept a list of transitions as arguments and a `play` method that lets you start the animations. For example, if you have three transitions named `t1`, `t2`, and `t3` already created, you can run them in sequence like this:

```
SequentialTransition s =
                new SequentialTransition(t1, t2, t3)
s.play();
```

When the `play` method is called, transition `t1` will run until completion and then transition `t2` will run. When `t2` finishes, transition `t3` will be run.

To run all three transitions simultaneously, use the `ParallelTransition` class instead:

```
ParallelTransition p =
    new ParallelTransition(t1, t2, t3)
p.play();
```

If you prefer, you can add animations after the constructor has been called by using the `getChildren` method. For example:

```
ParallelTransition p = new ParallelTransition()
p.getChildren().add(t1);
p.getChildren().add(t2);
p.getChildren().add(t3);
p.play();
```

Or:

```
ParallelTransition p = new ParallelTransition()
p.getChildren().addAll(t1, t2, t3);
p.play();
```

Note: An animation added to a `SequentialTransition` or `ParallelTransition` can itself be a `SequentialTransition` or a `ParallelTransition`. For example, suppose you have three transitions that animate one node (`t1`, `t2`, and `t3`) and a fourth transition that animates a second node (`t4`) and you want to run `t1`, `t2`, and `t3` in sequence while `t4` runs at the same time as the sequence. Here's how you can achieve that:

```
SequentialTransition s =
    new SequentialTransition(t1, t2, t3)
ParallelTransition p = new ParallelTransition(s, t4);
p.play();
```

To illustrate how transitions can be combined into a complete program, Listing 17-2 shows a variation of the BouncingBall program that was presented in Listing 17-1.

Listing 17-2: The TwoBouncingBalls Program

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.shape.*;
import javafx.scene.paint.*;
import javafx.animation.*;
import javafx.util.*;

public class TwoBouncingBalls extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    @Override public void start(Stage primaryStage)
    {

        RadialGradient g = new RadialGradient(
            0, 0,
            0.35, 0.35,
            0.5,
            true,
            CycleMethod.NO_CYCLE,
            new Stop(0.0, Color.WHITE),
            new Stop(1.0, Color.RED));

        Circle ball1 = new Circle(0,0,20);
        ball1.setFill(g);

        Circle ball2 = new Circle(0,0,20);
        ball2.setFill(g);
```

```
        Group root = new Group();
        root.getChildren().addAll(ball1, ball2);

        Scene scene = new Scene(root, 600, 600);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Two Bouncing Balls");
        primaryStage.show();

        // Bounce ball 1
        TranslateTransition t1 = new TranslateTransition(
            Duration.millis(2000), ball1);
        t1.setFromX(ball1.getRadius());
        t1.setToX(scene.getWidth() - ball1.getRadius());
        t1.setFromY(scene.getHeight() / 3);
        t1.setToY(scene.getHeight() / 3);
        t1.setCycleCount(Transition.INDEFINITE);
        t1.setAutoReverse(true);
        t1.setInterpolator(Interpolator.LINEAR);

        // Bounce ball 2
        TranslateTransition t2 = new TranslateTransition(
            Duration.millis(2000), ball2);
        t2.setFromX(scene.getWidth() - ball2.getRadius());
        t2.setToX(ball2.getRadius());
        t2.setFromY(scene.getHeight() / 3 * 2);
        t2.setToY(scene.getHeight() / 3 * 2);
        t2.setCycleCount(Transition.INDEFINITE);
        t2.setAutoReverse(true);
        t2.setInterpolator(Interpolator.LINEAR);

        // Bounce both balls at the same time
        ParallelTransition pt = new ParallelTransition(t1, t2);
        pt.play();
    }

}
```

This version of the program animates two balls traveling in opposite directions. A transition is created on the first ball to bounce it from left to right one third of the way down the scene. A transition is created for the second ball to animate it in the opposite direction two thirds of the way down the scene. Then, a `ParallelTransition` is used to animate both balls at the same time. Figure 17-2 shows the program in action.

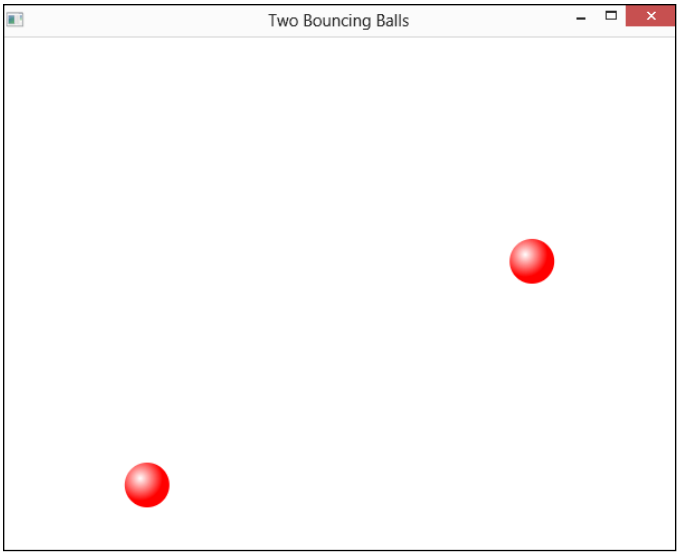

Figure 17-2: Bouncing two balls.

Animating the Hard Way

Now that you've seen the easy way to create animations (using transition classes), it's time to have a look at the more difficult way. Of course, as you might envision, the more difficult way is also the more flexible way.

For example, the bouncing balls shown in Listings 17-1 and 17-2 are interesting but not very practical, as they bounce back in forth in a strictly horizontal direction. If you wanted to use these balls to create a game, you'd want them to bounce around at angles, bouncing off all four edges of the scene. And you'd want them to bounce off each other as well. To achieve that, you need to work with two advanced animation classes: `KeyFrame` and `TimeLine`:

- **`KeyFrame`:** A `KeyFrame` is a timing interval that raises an `ActionEvent` when the time interval has expired.

  When you create a `KeyFrame`, you specify the duration of the time interval and provide an `ActionEvent` listener. Then, in the action listener, you provide the code that implements your animation. In the case of this bouncing ball program, the action event will examine each ball that's in motion and calculate the next position for each ball, taking into account the effect of balls bouncing off the edges of the scene or bouncing off each other.

- **Timeline:** A `Timeline` is a sequence of `KeyFrames`.

  When you call the `TimeLine`'s play method, each `KeyFrame` is executed in sequence. A `Timeline` also has a `cycleCount` property, which indicates how many times the timeline should be repeated, and `play`, `pause`, and `stop` methods that control the execution of the timeline. You can set the cycle count to `INDEFINATE` to continue the animation indefinitely.

To create a `KeyFrame`, call the `KeyFrame` constructor with two arguments: the duration of the keyframe (usually in milliseconds) and the `ActionEvent` listener that will be called when the timer expires. Here's an example that uses a Lambda expression to define a simple listener:

```
KeyFrame k = new KeyFrame(Duration.millis(10),
    e ->
    {
        // Action event listener code goes here
    } );
```

To add this `KeyFrame` to a `Timeline` and run the animation, use this code:

```
Timeline t = new Timeline(k);
t.setCycleCount(Timeline.INDEFINITE);
t.play();
```

Listing 17-3 shows a program that uses the `KeyFrame` and `Timeline` classes to send a ball bouncing off all four of the edges of a scene.

Listing 17-3: The Hard BouncingBall Program

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.event.*;
import javafx.scene.layout.*;
import javafx.scene.shape.*;
import javafx.scene.paint.*;
import javafx.animation.*;
import javafx.util.*;

public class HardBouncingBall extends Application {

    public static void main(String[] args)
    {
        launch(args);
    }
```

(continued)

Listing 17-3 ***(continued)***

```
    private Circle ball;                                                    →18
    private double x_speed = 2;
    private double y_speed = 3;
    final private int WIDTH = 600;
    final private int HEIGHT = 500;
    final private int BALL_SIZE = 20;

    @Override public void start(final Stage primaryStage)
    {
        Group root = new Group();

        RadialGradient g = new RadialGradient(                              →29
            0, 0,
            0.35, 0.35,
            0.5,
            true,
            CycleMethod.NO_CYCLE,
            new Stop(0.0, Color.WHITE),
            new Stop(1.0, Color.RED));

        ball = new Circle(BALL_SIZE, g);                                    →38
        ball.setCenterX(BALL_SIZE);
        ball.setCenterY(BALL_SIZE);

        root.getChildren().addAll(ball);                                    →42
        Scene scene = new Scene(root, WIDTH, HEIGHT);
        primaryStage.setTitle("Bouncing Ball");
        primaryStage.setScene(scene);
        primaryStage.show();

        KeyFrame k = new KeyFrame(Duration.millis(10),                      →48
            e ->
            {
                ball.setCenterX(ball.getCenterX() + x_speed);               →51
                ball.setCenterY(ball.getCenterY() + y_speed);

                if (ball.getCenterX() <= BALL_SIZE ||                       →54
                    ball.getCenterX() >= WIDTH - BALL_SIZE)
                    x_speed = -x_speed;

                if (ball.getCenterY() <= BALL_SIZE ||                       →58
                    ball.getCenterY() >= HEIGHT - BALL_SIZE)
                    y_speed = -y_speed;
            } );

        Timeline t = new Timeline(k);                                       →63
        t.setCycleCount(Timeline.INDEFINITE);
        t.play();
    }

}
```

The following paragraphs draw attention to the key points in this program:

- → **18:** These class variables are used within the `ActionEvent` handler.
- → **29:** The gradient that will be used to fill the ball is created here.
- → **38:** The ball is created. The initial x- and y-coordinates place the ball at the top-left corner of the scene.
- → **42:** The scene is set and the stage is displayed.
- → **48:** A `KeyFrame` is created with a timing duration of 10 milliseconds. A Lambda expression is used to create the event listener.
- → **51:** Within the event listener, the ball is moved by adding the current `x_speed` and `y_speed` values to the center `x` and `y` positions of the ball.
- → **54:** Next, the ball's `x` position is checked against the left and right edges of the scene. If the ball is at the edge, the `x_speed` value is inverted so that the ball will travel in the opposite `x` direction.
- → **58:** Similarly, the ball's `y` position is checked against the top and bottom edges of the scene. If the ball is at the edge, the `y_speed` value is inverted.
- → **63:** A `Timeline` is created using the `KeyFrame`. The cycle count is set to `INDEFINITE`, and the `play` method is called to start the ball moving.

Figure 17-3 shows the bouncing ball in action.

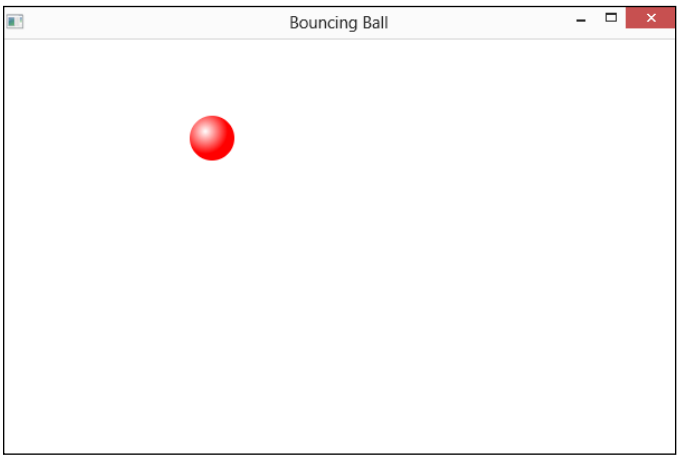

Figure 17-3: A ball that bounces off all four edges of the scene.

Improving the Ball Bouncer

The program shown in Listing 17-3 is finally beginning to resemble something that might be useful in a game program. With a little more programming effort, you can convert this program into something resembling the classic Pong game.

But to be more useful, the ball bouncing program needs to support more than one ball at a time within the scene. And after you add a second ball, the program should provide for the balls bouncing not only off the walls, but also off each other. Figure 17-4 shows a program that does just that. Here, a total of ten balls are flying around within the crowded scene. The balls bounce off the walls and each other.

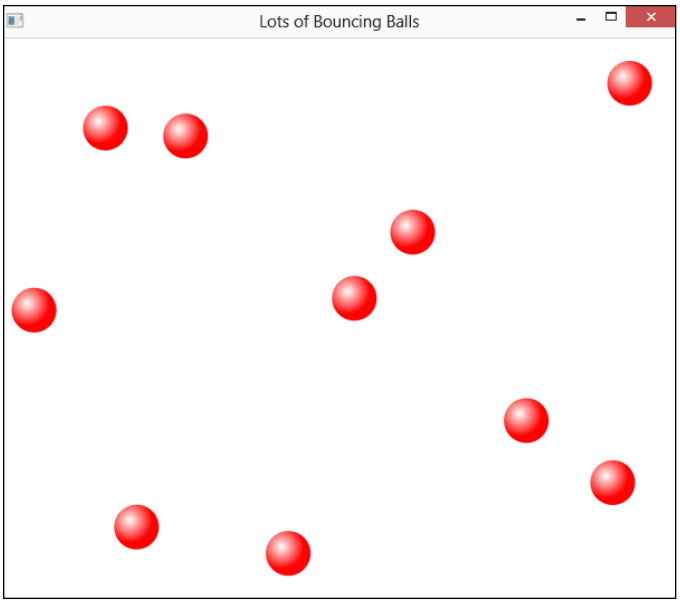

Figure 17-4: Lots of bouncing balls!

To implement this program, I created a separate class named `Ball` that extends the `Circle` class. The `Ball` class provides the following features:

- You specify the radius of the ball via the constructor. The ball is filled automatically with a red gradient to give it a three-dimensional appearance.
- You also specify the width and height of the bouncing area in the constructor. The ball is given an initial random position within this area, and automatically travels at a random speed within this specified area, bouncing off the edges when they're encountered.

- In addition, you pass an `ArrayList` of other balls to the constructor. As the ball moves, it automatically bounces off any other balls in this list that the ball happens to collide with. When balls collide, they trade their `x` and `y` speeds.
- The `Ball` class provides a `move` method that should be called from the animation `KeyFrame` action listener.

Listing 17-4 shows the complete code for the program, which includes the `Ball` class.

Listing 17-4: The ManyBalls Program

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.event.*;
import javafx.scene.layout.*;
import javafx.scene.shape.*;
import javafx.scene.paint.*;
import javafx.animation.*;
import javafx.util.*;
import java.util.*;

public class ManyBalls extends Application {

    public static void main(String[] args)
    {
        launch(args);
    }

    final private int WIDTH = 600;
    final private int HEIGHT = 500;
    final private int BALL_SIZE = 20;

    private ArrayList<Ball> balls = new ArrayList<Ball>();                  →23

    @Override public void start(final Stage primaryStage)
    {
        Group root = new Group();

        for (int i = 0; i < 10; i++)                                        →29
            balls.add(new Ball(BALL_SIZE, WIDTH, HEIGHT, balls));

        root.getChildren().addAll(balls);                                   →34

        Scene scene = new Scene(root, WIDTH, HEIGHT);                       →36
        primaryStage.setTitle("Lots of Bouncing Balls");
        primaryStage.setScene(scene);
        primaryStage.show();
```

(continued)

Listing 17-4 ***(continued)***

```
        KeyFrame k = new KeyFrame(Duration.millis(10),                    →41
            e ->
            {
                for (Ball ball : balls)
                    ball.move();
            });

        Timeline t = new Timeline(k);                                     →48
        t.setCycleCount(Timeline.INDEFINITE);
        t.play();
    }

    public class Ball extends Circle                                      →53
    {
        public double x_speed;                                            →55
        public double y_speed;
        public double radius;
        private double fieldWidth;
        private double fieldHeight;

        public Ball(double radius,                                        →61
                    double fieldWidth,
                    double fieldHeight,
                    ArrayList<Ball> balls)
        {
            super();                                                      →66

            this.radius = radius;                                         →68
            this.fieldWidth = fieldWidth;
            this.fieldHeight = fieldHeight;

            super.setRadius(radius);                                      →72

            super.setCenterX(                                             →74
                Math.random() * (fieldWidth - this.radius) + 1);
            super.setCenterY(
                Math.random() * (fieldHeight - this.radius) + 1);
            this.x_speed = Math.random() * 5 + 1;
            this.y_speed = Math.random() * 5 + 1;

            RadialGradient g = new RadialGradient(                        →81
                0, 0,
                0.35, 0.35,
                0.5,
                true,
                CycleMethod.NO_CYCLE,
                new Stop(0.0, Color.WHITE),
                new Stop(1.0, Color.RED));
            super.setFill(g);
        }
```

```
        public void move()                                                  →92
        {
            super.setCenterX(super.getCenterX() + this.x_speed);
            super.setCenterY(super.getCenterY() + this.y_speed);

            // Detect collision with left edge                              →97
            if (super.getCenterX() <= this.radius)
            {
                super.setCenterX(this.radius);
                this.x_speed = -this.x_speed;
            }

            // Detect collision with right edge                             →104
            if (super.getCenterX() >= this.fieldWidth - this.radius)
            {
                super.setCenterX(this.fieldWidth - this.radius);
                this.x_speed = -this.x_speed;
            }

            // Detect collision with top edge                               →111
            if (super.getCenterY() <= this.radius)
            {
                super.setCenterY(this.radius);
                this.y_speed = -this.y_speed;
            }

            // Detect collision with bottom edge                            →118
            if (super.getCenterY() >= this.fieldHeight - this.radius)
            {
                super.setCenterY(this.fieldHeight - this.radius);
                this.y_speed = -this.y_speed;
            }

            // Detect collision with other balls                            →125
            for (Ball b : balls)
            {
                if (b != this &&
                    b.intersects(super.getLayoutBounds()))
                {
                    double tempx = this.x_speed;                            →131
                    double tempy = this.y_speed;
                    this.x_speed = b.x_speed;
                    this.y_speed = b.y_speed;
                    b.x_speed = tempx;
                    b.y_speed = tempy;
                    break;                                                  →137
                }
            }
        }
    }
}
```

The following paragraphs explain the key points of this program:

→ **23:** An `ArrayList` named `balls` is used to hold the balls that will be animated by this program.

→ **29:** A `for` loop creates ten `Ball` objects and adds them to the array list. The constructor for the `Ball` objects passes the ball size, the width and height of the scene, and the `balls` array list.

→ **34:** The `balls` array list is added to the scene root.

→ **36:** The root is added to the scene, and the scene is displayed.

→ **41:** A `KeyFrame` is created with an interval of 10 milliseconds. The action listener is very simple: It simply calls the `move` method for every `Ball` in the `balls` collection.

→ **48:** A `Timeline` is created using the `KeyFrame` created in line 41. Then, the timeline is played to set the balls in motion.

→ **53:** The `Ball` class extends the `Circle` class.

→ **55:** Class fields are used to hold the internal values for the `x` and `y` speeds, the circle radius, and the playing field's width and height.

→ **61:** The constructor accepts four parameters: the radius, the playing field width and height, and an array list containing other `Ball` objects that should be checked for collisions.

→ **66:** The `Ball` constructor starts by calling the `super` constructor to create the `Circle` object from which this ball will be extended.

→ **68:** The radius, width, and height class variables are initialized from the values passed into the constructor.

→ **72:** The radius of the circle is set to match the radius passed to the constructor.

→ **74:** The center `x` and `y` positions as well as the `x` and `y` speed fields are set to random values.

→ **81:** The gradient fill is created.

→ **92:** The `move` method begins by adding the `x` and `y` speed fields to the circle's center `x` and `y` positions.

→ **97:** An `if` statement is used to detect a collision with the left edge. If the left edge collision occurs, the position of the ball is adjusted to bring it back within the playing field and the `x` speed is inverted so that the ball will change directions. The reason for repositioning the ball is that, depending on the previous position and the speed of the ball, the ball may have actually crossed outside the playing field.

→ **104:** Another `if` statement checks for a collision with the right edge.

→ **111:** And the top edge.

→ **118:** And the bottom edge.

→ **125:** Now it gets interesting. These lines check for collisions with other balls in the `balls` array. A `for` loop iterates over all the balls in the array list. The `if` statement first eliminates the current ball by checking if `b != this`. After all, a ball can't really collide with itself.

Next, the `if` statement checks to see whether the current ball has collided with any other ball. It does this by calling the `intersects` method, which is defined by the `Shape` class. This method accepts a `Bounds` object that represents the bounding rectangle within which a shape fits. You can get the bounding rectangle by calling the shape's `getLayoutBounds` method. Thus, this test works by checking whether a ball in the `balls` collection intersects with the bounding rectangle of the current ball. ***Note:*** This collision test isn't perfect; it sometimes treats near misses as collisions. But it's close enough.

→ **131:** If a collision is detected, the `x` and `y` speed values of the two balls are swapped. Not only do the balls bounce away from each other, but also the slower ball picks up speed and the faster ball slows down.

→ **137:** A `break` statement is executed if a collision is detected to prevent detecting collisions with more than one ball. Without this `break` statement, collisions that involve more than two balls usually result in pretty strange behavior. Try removing the `break` statement to see what happens. (Even with this `break` statement, the balls sometimes behave in unexpected ways. I think it's kind of fun to watch, but then again, I'm pretty easily entertained.)