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.

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.

```
// Fig. 22.14: BallAnimationTimerController.java
// Bounce a circle around a window using an Anim
import java.security.SecureRandom;
import javafx.animation.AnimationTimer;
import javafx.fxml.FXML;
import javafx.geometry.Bounds;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.util.Duration;

public class BallAnimationTimerController {
   @FXML private Circle c;
   @FXML private Pane pane;

   public void initialize() {
      SecureRandom random = new SecureRandom();

      // define a timeline animation
      AnimationTimer timer = new AnimationTimer(
         int dx = 1 + random.nextInt(5);
         int dy = 1 + random.nextInt(5);
         int velocity = 60; // used to scale dis
         long previousTime = System.nanoTime();

         // specify how to move Circle for curre
         @Override
         public void handle(long now) {
            double elapsedTime = (now - previous
            previousTime = now;
            double scale = elapsedTime * velocit

            Bounds bounds = pane.getBoundsInLoca
            c.setLayoutX(c.getLayoutX() + dx * s
            c.setLayoutY(c.getLayoutY() + dy * s

            if (hitRightOrLeftEdge(bounds)) {
               dx *= -1;
            }

            if (hitTopOrBottom(bounds)) {
               dy *= -1;
            }
         }
      };

      timer.start();
   }

   // determines whether the circle hit left/rig
   private boolean hitRightOrLeftEdge(Bounds bou
      return (c.getLayoutX() <= (bounds.getMinX(
         (c.getLayoutX() >= (bounds.getMaxX() -
   }

   // determines whether the circle hit top/bott
   private boolean hitTopOrBottom(Bounds bounds)
      return (c.getLayoutY() <= (bounds.getMinY(
         (c.getLayoutY() >= (bounds.getMaxY() -
   }
}
```

Fig. 22.14

Bounce a circle around a window using an `AnimationTimer` subclass.

Extending `abstract` Class `AnimationTimer`

Class `AnimationTimer` is an `abstract` class, so you must create a subclass. In this example, lines 19–44 create an anonymous inner class that extends `AnimationTimer`. Lines 20–23 define the anonymous inner class’s instance variables:

- 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.

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.
- 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.