

# CHAPTER 15

## EVENT-DRIVEN PROGRAMMING AND ANIMATIONS

### Objectives

- To get a taste of event-driven programming (§15.1).
- To describe events, event sources, and event classes (§15.2).
- To define handler classes, register handler objects with the source object, and write the code to handle events (§15.3).
- To define handler classes using inner classes (§15.4).
- To define handler classes using anonymous inner classes (§15.5).
- To simplify event handling using lambda expressions (§15.6).
- To develop a GUI application for a loan calculator (§15.7).
- To write programs to deal with **MouseEvent**s (§15.8).
- To write programs to deal with **KeyEvent**s (§15.9).
- To create listeners for processing a value change in an observable object (§15.10).
- To use the **Animation**, **PathTransition**, **FadeTransition**, and **Timeline** classes to develop animations (§15.11).
- To develop an animation for simulating a bouncing ball (§15.12).
- To draw, color, and resize a US map (§15.13).



15.1 Introduction

You can write code to process events such as a button click, mouse movement, and keystrokes.

problem



Suppose you wish to write a GUI program that lets the user enter a loan amount, annual interest rate, and number of years then click the *Calculate* button to obtain the monthly payment and total payment, as shown in Figure 15.1. How do you accomplish the task? You have to use *event-driven programming* to write the code to respond to the button-clicking event.

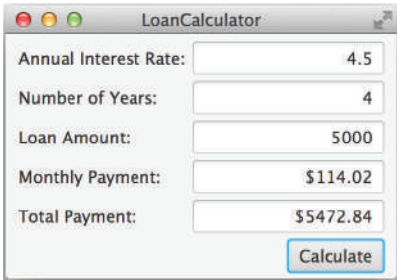


FIGURE 15.1 The program computes loan payments.

Before delving into event-driven programming, it is helpful to get a taste using a simple example. The example displays two buttons in a pane, as shown in Figure 15.2.

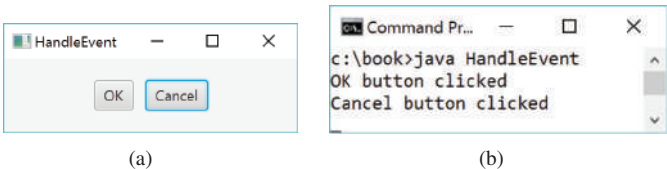


FIGURE 15.2 (a) The program displays two buttons. (b) A message is displayed in the console when a button is clicked.

To respond to a button click, you need to write the code to process the button-clicking action. The button is an *event source object*—where the action originates. You need to create an object capable of handling the action event on a button. This object is called an *event handler*, as shown in Figure 15.3.

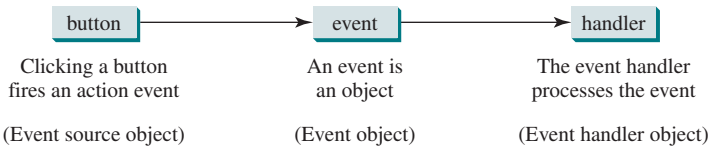


FIGURE 15.3 An event handler processes the event fired from the source object.

Not all objects can be handlers for an action event. To be a handler of an action event, two requirements must be met:

EventHandler interface

setOnAction(handler)

1. The object must be an instance of the `EventHandler<T extends Event>` interface. This interface defines the common behavior for all handlers. `<T extends Event>` denotes that `T` is a generic type that is a subtype of `Event`.
2. The `EventHandler` object `handler` must be registered with the event source object using the method `source.setOnAction(handler)`.

The `EventHandler<ActionEvent>` interface contains the `handle(ActionEvent)` method for processing the action event. Your handler class must override this method to respond to the event. Listing 15.1 gives the code that processes the `ActionEvent` on the two buttons. When you click the *OK* button, the message “OK button clicked” is displayed. When you click the *Cancel* button, the message “Cancel button clicked” is displayed, as shown in Figure 15.2.

### LISTING 15.1 HandleEvent.java

```

1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.layout.HBox;
6  import javafx.stage.Stage;
7  import javafx.event.ActionEvent;
8  import javafx.event.EventHandler;
9
10 public class HandleEvent extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         // Create a pane and set its properties
14         HBox pane = new HBox(10);
15         pane.setAlignment(Pos.CENTER);
16         Button btOK = new Button("OK");
17         Button btCancel = new Button("Cancel");
18         OKHandlerClass handler1 = new OKHandlerClass();           create handler
19         btOK.setOnAction(handler1);                                register handler
20         CancelHandlerClass handler2 = new CancelHandlerClass();   create handler
21         btCancel.setOnAction(handler2);                             register handler
22         pane.getChildren().addAll(btOK, btCancel);
23
24         // Create a scene and place it in the stage
25         Scene scene = new Scene(pane);
26         primaryStage.setTitle("HandleEvent"); // Set the stage title
27         primaryStage.setScene(scene); // Place the scene in the stage
28         primaryStage.show(); // Display the stage
29     }
30 }                                                                    main method omitted
31
32 class OKHandlerClass implements EventHandler<ActionEvent> {       handler class
33     @Override
34     public void handle(ActionEvent e) {                            handle event
35         System.out.println("OK button clicked");
36     }
37 }
38
39 class CancelHandlerClass implements EventHandler<ActionEvent> {   handler class
40     @Override
41     public void handle(ActionEvent e) {                            handle event
42         System.out.println("Cancel button clicked");
43     }
44 }

```

Two handler classes are defined in lines 32-44. Each handler class implements `EventHandler<ActionEvent>` to process `ActionEvent`. The object `handler1` is an instance of `OKHandlerClass` (line 18), which is registered with the button `btOK` (line 19). When the *OK* button is clicked, the `handle(ActionEvent)` method (line 34) in `OKHandlerClass` is

invoked to process the event. The object `handler2` is an instance of `CancelHandlerClass` (line 20), which is registered with the button `btCancel1` in line 21. When the *Cancel* button is clicked, the `handle(ActionEvent)` method (line 41) in `CancelHandlerClass` is invoked to process the event.

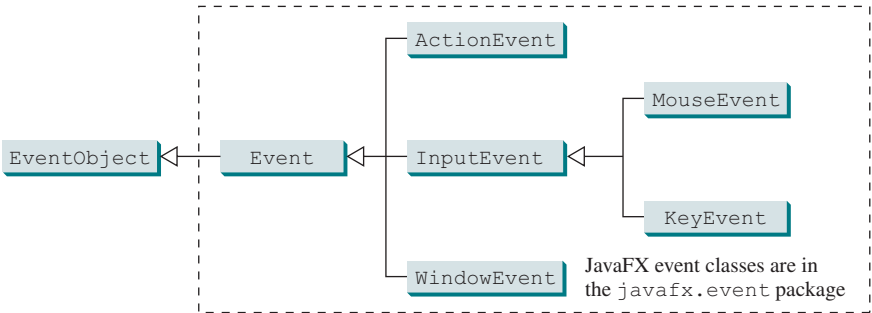
You now have seen a glimpse of event-driven programming in JavaFX. You probably have many questions, such as why a handler class is defined to implement the `EventHandler<ActionEvent>`. The following sections will give you all the answers.

## 15.2 Events and Event Sources

*An event is an object created from an event source. Firing an event means to create an event and delegate the handler to handle the event.*

When you run a Java GUI program, the program interacts with the user and the events drive its execution. This is called *event-driven programming*. An *event* can be defined as a signal to the program that something has happened. Events are triggered by external user actions, such as mouse movements, mouse clicks, and keystrokes. The program can choose to respond to or ignore an event. The example in the preceding section gave you a taste of event-driven programming.

The component that creates an event and fires it is called the *event source object*, or simply *source object* or *source component*. For example, a button is the source object for a button-clicking action event. An event is an instance of an event class. The root class of the Java event classes is `java.util.EventObject`. The root class of the JavaFX event classes is `javafx.event.Event`. The hierarchical relationships of some event classes are shown in Figure 15.4.



**FIGURE 15.4** An event in JavaFX is an object of the `javafx.event.Event` class.

An *event object* contains whatever properties are pertinent to the event. You can identify the source object of an event using the `getSource()` instance method in the `EventObject` class. The subclasses of `EventObject` deal with specific types of events, such as action events, window events, mouse events, and key events. The first three columns in Table 15.1 list some external user actions, source objects, and event types fired. For example, when clicking a button, the button creates and fires an `ActionEvent`, as indicated in the first line of this table. Here, the button is an event source object, and an `ActionEvent` is the event object fired by the source object, as shown in Figure 15.3.



### Note

If a component can fire an event, any subclass of the component can fire the same type of event. For example, every JavaFX shape, layout pane, and control can fire `MouseEvent` and `KeyEvent` since `Node` is the superclass for shapes, layout panes, and controls and `Node` can fire `MouseEvent` and `KeyEvent`.



event-driven programming  
event

fire event  
event source object  
source object

event object  
`getSource()`

**TABLE 15.1** User Action, Source Object, Event Type, Handler Interface, and Handler

User Action	Source Object	Event Type Fired	Event Registration Method
Click a button	<b>Button</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Press Enter in a text field	<b>TextField</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Check or uncheck	<b>RadioButton</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Check or uncheck	<b>CheckBox</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Select a new item	<b>ComboBox</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Mouse pressed	<b>Node, Scene</b>	<b>MouseEvent</b>	<code>setOnMousePressed(EventHandler&lt;MouseEvent&gt;)</code>
Mouse released			<code>setOnMouseReleased(EventHandler&lt;MouseEvent&gt;)</code>
Mouse clicked			<code>setOnMouseClicked(EventHandler&lt;MouseEvent&gt;)</code>
Mouse entered			<code>setOnMouseEntered(EventHandler&lt;MouseEvent&gt;)</code>
Mouse exited			<code>setOnMouseExited(EventHandler&lt;MouseEvent&gt;)</code>
Mouse moved			<code>setOnMouseMoved(EventHandler&lt;MouseEvent&gt;)</code>
Mouse dragged			<code>setOnMouseDragged(EventHandler&lt;MouseEvent&gt;)</code>
Key pressed	<b>Node, Scene</b>	<b>KeyEvent</b>	<code>setOnKeyPressed(EventHandler&lt;KeyEvent&gt;)</code>
Key released			<code>setOnKeyReleased(EventHandler&lt;KeyEvent&gt;)</code>
Key typed			<code>setOnKeyTyped(EventHandler&lt;KeyEvent&gt;)</code>

**15.2.1** What is an event source object? What is an event object? Describe the relationship between an event source object and an event object.

**15.2.2** Can a button fire a **MouseEvent**? Can a button fire a **KeyEvent**? Can a button fire an **ActionEvent**?



## 15.3 Registering Handlers and Handling Events

*A handler is an object that must be registered with an event source object and it must be an instance of an appropriate event-handling interface.*



Java uses a delegation-based model for event handling: A source object fires an event, and an object interested in the event handles it. The latter object is called an *event handler* or an *event listener*. For an object to be a handler for an event on a source object, two things are needed, as shown in Figure 15.5.

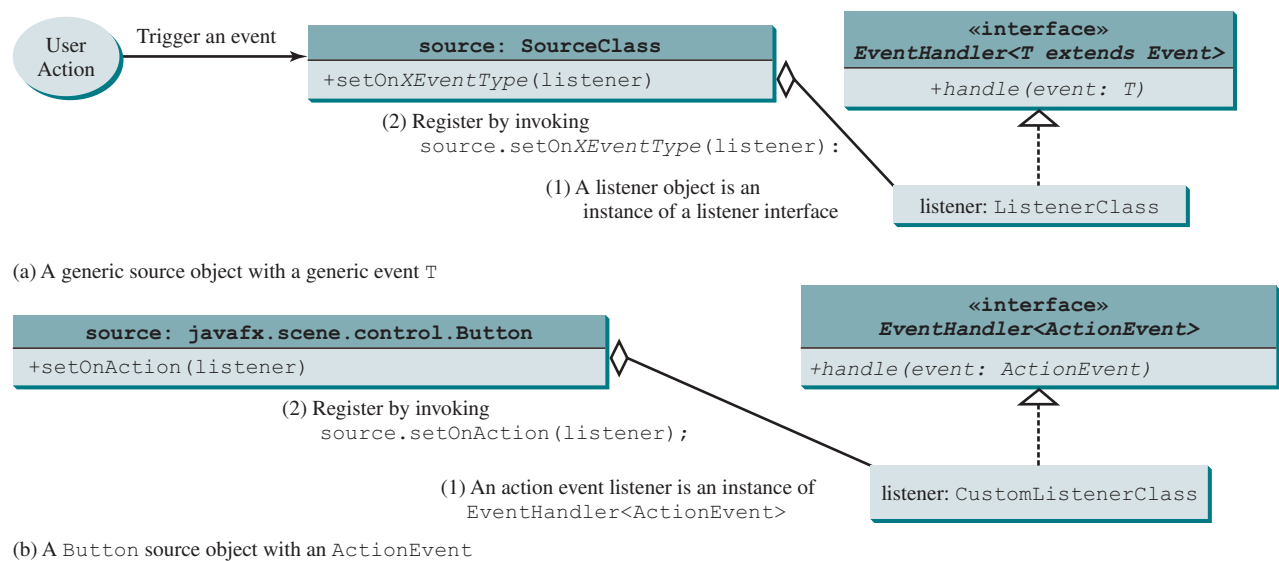
event delegation  
event handler

1. *The handler object must be an instance of the corresponding event–handler interface* to ensure the handler has the correct method for processing the event. JavaFX defines a unified handler interface **EventHandler<T extends Event>** for an event **T**. The handler interface contains the **handle(T e)** method for processing the event. For example, the handler interface for **ActionEvent** is **EventHandler<ActionEvent>**; each handler for **ActionEvent** should implement the **handle(ActionEvent e)** method for processing an **ActionEvent**.
2. *The handler object must be registered by the source object.* Registration methods depend on the event type. For **ActionEvent**, the method is **setOnAction**. For a mouse-pressed event, the method is **setOnMousePressed**. For a key-pressed event, the method is **setOnKeyPressed**.

event–handler interface  
**EventHandler<T extends Event>**  
event handler

register handler

Let's revisit Listing 15.1, `HandleEvent.java`. Since a **Button** object fires **ActionEvent**, a handler object for **ActionEvent** must be an instance of **EventHandler<ActionEvent>**, so



**FIGURE 15.5** A listener must be an instance of a listener interface and must be registered with a source object.

the handler class implements `EventHandler<ActionEvent>` in line 32. The source object invokes `setOnAction(handler)` to register a handler, as follows:

```
// Line 16 in Listing 15.1
create source object    Button btOK = new Button("OK");

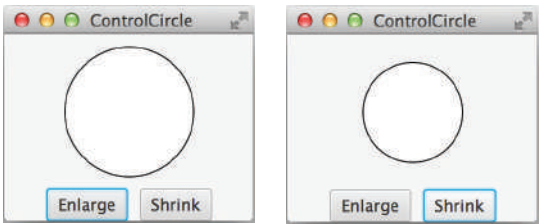
// Line 18 in Listing 15.1
create handler object   OKHandlerClass handler1 = new OKHandlerClass();

// Line 19 in Listing 15.1
register handler        btOK.setOnAction(handler1);
```

When you click the button, the `Button` object fires an `ActionEvent` and passes it to invoke the handler's `handle(ActionEvent)` method to handle the event. The event object contains information pertinent to the event, which can be obtained using the methods. For example, you can use `e.getSource()` to obtain the source object that fired the event.

We now write a program that uses two buttons to enlarge and shrink a circle, as shown in Figure 15.6. We will develop this program incrementally. First, we write the program in Listing 15.2 that displays the user interface with a circle in the center (lines 15–19) and two buttons on the bottom (lines 21–27).

first version



**FIGURE 15.6** The user clicks the *Enlarge* and *Shrink* buttons to enlarge and shrink the circle.

### LISTING 15.2 `ControlCircleWithoutEventHandlering.java`

```
1 import javafx.application.Application;
2 import javafx.geometry.Pos;
3 import javafx.scene.Scene;
```



```

4 import javafx.scene.control.Button;
5 import javafx.scene.layout.StackPane;
6 import javafx.scene.layout.HBox;
7 import javafx.scene.layout.BorderPane;
8 import javafx.scene.paint.Color;
9 import javafx.scene.shape.Circle;
10 import javafx.stage.Stage;
11
12 public class ControlCircleWithoutEventHandling extends Application {
13     @Override // Override the start method in the Application class
14     public void start(Stage primaryStage) {
15         StackPane pane = new StackPane();
16         Circle circle = new Circle(50);
17         circle.setStroke(Color.BLACK);
18         circle.setFill(Color.WHITE);
19         pane.getChildren().add(circle);
20
21         HBox hBox = new HBox();
22         hBox.setSpacing(10);
23         hBox.setAlignment(Pos.CENTER);
24         Button btEnlarge = new Button("Enlarge");
25         Button btShrink = new Button("Shrink");
26         hBox.getChildren().add(btEnlarge);
27         hBox.getChildren().add(btShrink);
28
29         BorderPane borderPane = new BorderPane();
30         borderPane.setCenter(pane);
31         borderPane.setBottom(hBox);
32         BorderPane.setAlignment(hBox, Pos.CENTER);
33
34         // Create a scene and place it in the stage
35         Scene scene = new Scene(borderPane, 200, 150);
36         primaryStage.setTitle("ControlCircle"); // Set the stage title
37         primaryStage.setScene(scene); // Place the scene in the stage
38         primaryStage.show(); // Display the stage
39     }
40 }

```

circle

buttons

main method omitted

How do you use the buttons to enlarge or shrink the circle? When the *Enlarge* button is clicked, you want the circle to be repainted with a larger radius. How can you accomplish this? You can expand and modify the program in Listing 15.2 into Listing 15.3 with the following features:

second version

1. Define a new class named **CirclePane** for displaying the circle in a pane (lines 51–68). This new class displays a circle and provides the **enlarge** and **shrink** methods for increasing and decreasing the radius of the circle (lines 60–62 and 64–67). It is a good strategy to design a class to model a circle pane with supporting methods so these related methods along with the circle are coupled in one object.
2. Create a **CirclePane** object and declare **circlePane** as a data field to reference this object (line 15) in the **ControlCircle** class. The methods in the **ControlCircle** class can now access the **CirclePane** object through this data field.
3. Define a handler class named **EnlargeHandler** that implements **EventHandler<ActionEvent>** (lines 43–48). To make the reference variable **circlePane** accessible from the **handle** method, define **EnlargeHandler** as an inner class of the **ControlCircle** class. (*Inner classes* are defined inside another class. We use an inner class here and will introduce it fully in the next section.)
4. Register the handler for the *Enlarge* button (line 29) and implement the **handle** method in **EnlargeHandler** to invoke **circlePane.enlarge()** (line 46).

inner class



## VideoNote

Handler and its registration

## LISTING 15.3 ControlCircle.java

```

1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.geometry.Pos;
5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.layout.StackPane;
8  import javafx.scene.layout.HBox;
9  import javafx.scene.layout.BorderPane;
10 import javafx.scene.paint.Color;
11 import javafx.scene.shape.Circle;
12 import javafx.stage.Stage;
13
14 public class ControlCircle extends Application {
15     private CirclePane circlePane = new CirclePane();
16
17     @Override // Override the start method in the Application class
18     public void start(Stage primaryStage) {
19         // Hold two buttons in an HBox
20         HBox hBox = new HBox();
21         hBox.setSpacing(10);
22         hBox.setAlignment(Pos.CENTER);
23         Button btEnlarge = new Button("Enlarge");
24         Button btShrink = new Button("Shrink");
25         hBox.getChildren().add(btEnlarge);
26         hBox.getChildren().add(btShrink);
27
28         // Create and register the handler
29         btEnlarge.setOnAction(new EnlargeHandler());
30
31         BorderPane borderPane = new BorderPane();
32         borderPane.setCenter(circlePane);
33         borderPane.setBottom(hBox);
34         BorderPane.setAlignment(hBox, Pos.CENTER);
35
36         // Create a scene and place it in the stage
37         Scene scene = new Scene(borderPane, 200, 150);
38         primaryStage.setTitle("ControlCircle"); // Set the stage title
39         primaryStage.setScene(scene); // Place the scene in the stage
40         primaryStage.show(); // Display the stage
41     }
42
43     class EnlargeHandler implements EventHandler<ActionEvent> {
44         @Override // Override the handle method
45         public void handle(ActionEvent e) {
46             circlePane.enlarge();
47         }
48     }
49 }
50
51 class CirclePane extends StackPane {
52     private Circle circle = new Circle(50);
53
54     public CirclePane() {
55         getChildren().add(circle);
56         circle.setStroke(Color.BLACK);
57         circle.setFill(Color.WHITE);
58     }

```

create/register handler

handler class

main method omitted

CirclePane class



```

59
60 public void enlarge() {
61     circle.setRadius(circle.getRadius() + 2);
62 }
63
64 public void shrink() {
65     circle.setRadius(circle.getRadius() > 2 ?
66         circle.getRadius() - 2 : circle.getRadius());
67 }
68 }

```

enlarge method

As an exercise, add the code for handling the *Shrink* button to display a smaller circle when the *Shrink* button is clicked.

- 15.3.1** Why must a handler be an instance of an appropriate handler interface?  
**15.3.2** Explain how to register a handler object and how to implement a handler interface.  
**15.3.3** What is the handler method for the `EventHandler<ActionEvent>` interface?  
**15.3.4** What is the registration method for a button to register an `ActionEvent` handler?



## 15.4 Inner Classes

*An inner class, or nested class, is a class defined within the scope of another class. Inner classes are useful for defining handler classes.*



The approach of this book is to introduce difficult programming concepts using practical examples. We introduce inner classes, anonymous inner classes, and lambda expressions using practical examples in this section and following two sections.

Inner classes are used in the preceding section. This section introduces inner classes in detail. First, let us see the code in Figure 15.7. The code in Figure 15.7a defines two separate classes, `Test` and `A`. The code in Figure 15.7b defines `A` as an inner class in `Test`.

```

public class Test {
    ...
}

public class A {
    ...
}

```

(a)

```

public class Test {
    ...
    // Inner class
    public class A {
        ...
    }
}

```

(b)

```

// OuterClass.java: inner class demo
public class OuterClass {
    private int data;

    /** A method in the outer class */
    public void m() {
        // Do something
    }

    // An inner class
    class InnerClass {
        /** A method in the inner class */
        public void mi() {
            // Directly reference data and method
            // defined in its outer class
            data++;
            m();
        }
    }
}

```

(c)

**FIGURE 15.7** An inner class is defined as a member of another class.

The class `InnerClass` defined inside `OuterClass` in Figure 15.7c is another example of an inner class. An inner class may be used just like a regular class. Normally, you define

a class as an inner class if it is used only by its outer class. An inner class has the following features:

- An inner class is compiled into a class named `OuterClassName$InnerClassName.class`. For example, the inner class `A` in `Test` is compiled into `Test$A.class` in Figure 15.7b.
- An inner class can reference the data and the methods defined in the outer class in which it nests, so you need not pass the reference of an object of the outer class to the constructor of the inner class. For this reason, inner classes can make programs simple and concise. For example, `circlePane` is defined in `ControlCircle` in Listing 15.3 (line 15). It can be referenced in the inner class `EnlargeHandler` in line 46.
- An inner class can be defined with a visibility modifier subject to the same visibility rules applied to a member of the class.
- An inner class can be defined as `static`. A `static` inner class can be accessed using the outer class name. A `static` inner class cannot access nonstatic members of the outer class.
- Objects of an inner class are often created in the outer class. However, you can also create an object of an inner class from another class. If the inner class is nonstatic, you must first create an instance of the outer class, then use the following syntax to create an object for the inner class:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

- If the inner class is static, use the following syntax to create an object for it:

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

A simple use of inner classes is to combine dependent classes into a primary class. This reduces the number of source files. It also makes class files easy to organize since they are all named with the primary class as the prefix. For example, rather than creating the two source files `Test.java` and `A.java` as shown in Figure 15.7a, you can merge class `A` into class `Test` and create just one source file, `Test.java` as shown in Figure 15.7b. The resulting class files are `Test.class` and `Test$A.class`.

Another practical use of inner classes is to avoid class-naming conflicts. Two versions of `A` are defined in Figure 15.7a and 15.7b. You can define them as inner classes to avoid a conflict.

A handler class is designed specifically to create a handler object for a GUI component (e.g., a button). The handler class will not be shared by other applications and therefore is appropriate to be defined inside the main class as an inner class.



**15.4.1** Can an inner class be used in a class other than the class in which it nests?

**15.4.2** Can the modifiers `public`, `protected`, `private`, and `static` be used for inner classes?

## 15.5 Anonymous Inner-Class Handlers

*An anonymous inner class is an inner class without a name. It combines defining an inner class and creating an instance of the class into one step.*

Inner-class handlers can be shortened using *anonymous inner classes*. The inner class in Listing 15.3 can be replaced by an anonymous inner class as shown below. The complete code is available at [liveexample.pearsoncmg.com/html/ControlCircleWithAnonymousInnerClass.html](http://liveexample.pearsoncmg.com/html/ControlCircleWithAnonymousInnerClass.html).



anonymous inner class

```

public void start(Stage primaryStage) {
    // Omitted

    btEnlarge.setOnAction(
        new EnlargeHandler());
}

class EnlargeHandler
    implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) {
        circlePane.enlarge();
    }
}

```

(a) Inner class EnlargeListener

```

public void start(Stage primaryStage) {
    // Omitted

    btEnlarge.setOnAction(
        new class EnlargeHandler
            implements EventHandler<ActionEvent>() {
            public void handle(ActionEvent e) {
                circlePane.enlarge();
            }
        });
}

```

(b) Anonymous inner class

The syntax for an anonymous inner class is shown below.

```

new SuperClassName/InterfaceName() {
    // Implement or override methods in superclass or interface

    // Other methods if necessary
}

```

Since an anonymous inner class is a special kind of inner class, it is treated like an inner class with the following features:

- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit **extends** or **implements** clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is **Object()**.
- An anonymous inner class is compiled into a class named **OuterClassName\$n.class**. For example, if the outer class **Test** has two anonymous inner classes, they are compiled into **Test\$1.class** and **Test\$2.class**.

Listing 15.4 gives an example that displays a text and uses four buttons to move a text up, down, left, and right, as shown in Figure 15.8.

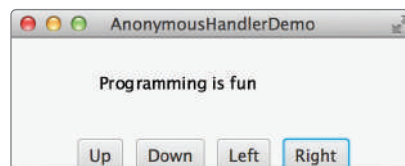


FIGURE 15.8 The program handles the events from four buttons.

## LISTING 15.4 AnonymousHandlerDemo.java

```

1 import javafx.application.Application;
2 import javafx.event.ActionEvent;
3 import javafx.event.EventHandler;
4 import javafx.geometry.Pos;

```



VideoNote

Anonymous handler

```

5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.layout.BorderPane;
8  import javafx.scene.layout.HBox;
9  import javafx.scene.layout.Pane;
10 import javafx.scene.text.Text;
11 import javafx.stage.Stage;
12
13 public class AnonymousHandlerDemo extends Application {
14     @Override // Override the start method in the Application class
15     public void start(Stage primaryStage) {
16         Text text = new Text(40, 40, "Programming is fun");
17         Pane pane = new Pane(text);
18
19         // Hold four buttons in an HBox
20         Button btUp = new Button("Up");
21         Button btDown = new Button("Down");
22         Button btLeft = new Button("Left");
23         Button btRight = new Button("Right");
24         HBox hBox = new HBox(btUp, btDown, btLeft, btRight);
25         hBox.setSpacing(10);
26         hBox.setAlignment(Pos.CENTER);
27
28         BorderPane borderPane = new BorderPane(pane);
29         borderPane.setBottom(hBox);
30
31         // Create and register the handler
32         btUp.setOnAction(new EventHandler<ActionEvent>() {
33             @Override // Override the handle method
34             public void handle(ActionEvent e) {
35                 text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
36             }
37         });
38
39         btDown.setOnAction(new EventHandler<ActionEvent>() {
40             @Override // Override the handle method
41             public void handle(ActionEvent e) {
42                 text.setY(text.getY() < pane.getHeight() ?
43                     text.getY() + 5 : pane.getHeight());
44             }
45         });
46
47         btLeft.setOnAction(new EventHandler<ActionEvent>() {
48             @Override // Override the handle method
49             public void handle(ActionEvent e) {
50                 text.setX(text.getX() > 0 ? text.getX() - 5 : 0);
51             }
52         });
53
54         btRight.setOnAction(new EventHandler<ActionEvent>() {
55             @Override // Override the handle method
56             public void handle(ActionEvent e) {
57                 text.setX(text.getX() < pane.getWidth() - 100 ?
58                     text.getX() + 5 : pane.getWidth() - 100);
59             }
60         });
61
62         // Create a scene and place it in the stage
63         Scene scene = new Scene(borderPane, 400, 350);
64         primaryStage.setTitle("AnonymousHandlerDemo"); // Set title

```

anonymous handler

handle event

```

65     primaryStage.setScene(scene); // Place the scene in the stage
66     primaryStage.show(); // Display the stage
67 }
68 }

```

The program creates four handlers using anonymous inner classes (lines 32–60). Without using anonymous inner classes, you would have to create four separate classes. An anonymous handler works the same way as that of an inner-class handler. The program is condensed using an anonymous inner class. Another benefit of using anonymous inner class is that the handler can access local variables. In this example, the event handler references local variable **text** (lines 35, 42, 50, and 57).

The anonymous inner classes in this example are compiled into **AnonymousHandlerDemo\$1.class**, **AnonymousHandlerDemo\$2.class**, **AnonymousHandlerDemo\$3.class**, and **AnonymousHandlerDemo\$4.class**.

**15.5.1** If class **A** is an inner class in class **B**, what is the .class file for **A**? If class **B** contains two anonymous inner classes, what are the .class file names for these two classes?



**15.5.2** What is wrong in the following code?

```

public class Test extends Application {
    public void start(Stage stage) {
        Button btOK = new Button("OK");
    }

    private class Handler implements
        EventHandler<ActionEvent> {
        public void handle(ActionEvent e) {
            System.out.println(e.getSource());
        }
    }
}

```

(a)

```

public class Test extends Application {
    public void start(Stage stage) {
        Button btOK = new Button("OK");

        btOK.setOnAction(
            new EventHandler<ActionEvent> {
                public void handle
                    (ActionEvent e) {
                        System.out.println
                            (e.getSource());
                    }
            } // Something missing here
        );
    }
}

```

(b)

## 15.6 Simplifying Event Handling Using Lambda Expressions

*Lambda expressions can be used to greatly simplify coding for event handling.*

**Lambda expression** is a new feature in Java 8. Lambda expressions can be viewed as an anonymous class with a concise syntax. For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines. Note that the interface **EventHandler<ActionEvent>** and the method **handle** in (a) are removed in (b). This simplification is possible because that the Java compiler can automatically infer that the **setOnAction** method requires an instance of **EventHandler<ActionEvent>** and the **handle** is the only method in the **EventHandler<ActionEvent>** interface. The complete code that contains the lambda expression in (b) can be seen at [liveexample.pearsoncmg.com/html/ControlCircleWithLambdaExpression.html](http://liveexample.pearsoncmg.com/html/ControlCircleWithLambdaExpression.html).



lambda expression

```

btEnlarge.setOnAction {
    new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent e) {
            // Code for processing event e
        }
    }
});

```

(a) Anonymous inner class event handler

```

btEnlarge.setOnAction(e -> {
    // Code for processing event e
});

```

(b) Lambda expression event handler

The basic syntax for a lambda expression is either

`(type1 param1, type2 param2, . . . ) -> expression`

or

`(type1 param1, type2 param2, . . . ) -> { statements; }`

The data type for a parameter may be explicitly declared or implicitly inferred by the compiler. The parentheses can be omitted if there is only one parameter without an explicit data type. The curly braces can be omitted if there is only one statement. For example, the following lambda expressions are all equivalent. *Note there is no semicolon after the statement in (d).*

```
(ActionEvent e) -> {
    circlePane.enlarge(); }
```

(a) Lambda expression with one statement

```
(e) -> {
    circlePane.enlarge(); }
```

(b) Omit parameter data type

```
e -> {
    circlePane.enlarge(); }
```

(c) Omit parentheses

```
e ->
    circlePane.enlarge()
```

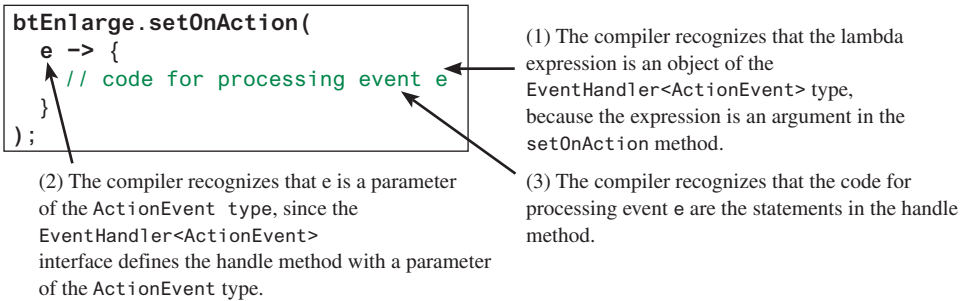
(d) Omit braces

The compiler treats a lambda expression as if it is an object created from an anonymous inner class. The compiler processes a lambda expression in three steps: (1) identify the lambda expression type, (2) identify the parameter types, and (3) identify statements. Consider the following lambda expression:

```
btEnlarge.setOnAction(
    e -> {
        // Code for processing event e
    }
);
```

It is processed as follows:

Step 1: The compiler recognizes that the object must be an instance of **EventHandler<ActionEvent>**, since the expression is an argument of the **setOnAction** method as shown in the following figure:



Step 2: Since the **EventHandler** interface defines the **handle** method with a parameter of the **ActionEvent** type, the compiler recognizes that **e** is a parameter of the **ActionEvent** type.

Step 3: The compiler recognizes that the code for processing **e** is the statements in the body of the **handle** method.

The **EventHandler** interface contains just one method named **handle**. The statements in the lambda expression are all for that method. If it contains multiple methods, the compiler

will not be able to compile the lambda expression. Therefore, for the compiler to understand lambda expressions, the interface must contain exactly one abstract method. Such an interface is known as a *Single Abstract Method (SAM) interface*.

In essence, a lambda expression creates an object and the object performs a function by invoking this single method. Thus, a SAM interface is also known as a *functional interface*, and an instance of a functional interface is known as a *function object*. Since a lambda expression is squarely on defining a function, a lambda expression is also called a *lambda function*. The terms lambda expression and lambda function are interchangeable.

Listing 15.4 can be simplified using lambda expressions as shown in Listing 15.5.

SAM interface

functional interface

function object

lambda function

functional programming

### LISTING 15.5 LambdaHandlerDemo.java

```

1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.geometry.Pos;
5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.layout.BorderPane;
8  import javafx.scene.layout.HBox;
9  import javafx.scene.layout.Pane;
10 import javafx.scene.text.Text;
11 import javafx.stage.Stage;
12
13 public class LambdaHandlerDemo extends Application {
14     @Override // Override the start method in the Application class
15     public void start(Stage primaryStage) {
16         Text text = new Text(40, 40, "Programming is fun");
17         Pane pane = new Pane(text);
18
19         // Hold four buttons in an HBox
20         Button btUp = new Button("Up");
21         Button btDown = new Button("Down");
22         Button btLeft = new Button("Left");
23         Button btRight = new Button("Right");
24         HBox hBox = new HBox(btUp, btDown, btLeft, btRight);
25         hBox.setSpacing(10);
26         hBox.setAlignment(Pos.CENTER);
27
28         BorderPane borderPane = new BorderPane(pane);
29         borderPane.setBottom(hBox);
30
31         // Create and register the handler
32         btUp.setOnAction((ActionEvent e) -> {
33             text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
34         });
35
36         btDown.setOnAction((e) -> {
37             text.setY(text.getY() < pane.getHeight() ?
38                 text.getY() + 5 : pane.getHeight());
39         });
40
41         btLeft.setOnAction(e -> {
42             text.setX(text.getX() > 0 ? text.getX() - 5 : 0);
43         });
44
45         btRight.setOnAction(e ->
46             text.setX(text.getX() < pane.getWidth() - 100 ?
47                 text.getX() + 5 : pane.getWidth() - 100)
48     );

```

lambda handler

lambda handler

lambda handler

lambda handler



```

49
50     // Create a scene and place it in the stage
51     Scene scene = new Scene(borderPane, 400, 350);
52     primaryStage.setTitle("AnonymousHandlerDemo"); // Set title
53     primaryStage.setScene(scene); // Place the scene in the stage
54     primaryStage.show(); // Display the stage
55 }
56 }


```

The program creates four handlers using lambda expressions (lines 32–48). Using lambda expressions, the code is shorter and cleaner. As seen in this example, lambda expressions may have many variations. Line 32 uses a declared type. Line 36 uses an inferred type since the type can be determined by the compiler. Line 41 omits the parentheses for a single inferred type. Line 45 omits the braces for a single statement in the body.

You can handle events by defining handler classes using inner classes, anonymous inner classes, or lambda expressions. We recommend you use lambda expressions because it produces a shorter, clearer, and cleaner code.

Using lambda expressions not only simplifies the syntax, but also simplifies the event-handling concept. For the statement in line 45,

(1) When the button is clicked                      (2) This function is performed



```

btRight.setOnAction(e -> move the text right);

```

you can now simply say that when the **btRight** button is clicked, the lambda function is invoked to move the text right.

You can define a custom functional interface and use it in a lambda expression. Consider the following example in Listing 15.6:

### LISTING 15.6 TestLambda.java

```

1  public class TestLambda {
2      public static void main(String[] args) {
3          TestLambda test = new TestLambda();
4          test.setAction1(() -> System.out.print("Action 1! "));
5          test.setAction2(e -> System.out.print(e + " "));
6          System.out.println(test.getValue((e1, e2) -> e1 + e2));
7      }
8
9      public void setAction1(T1 t) {
10         t.m1();
11     }
12
13     public void setAction2(T2 t) {
14         t.m2(4.5);
15     }
16
17     public int getValue(T3 t) {
18         return t.m3(5, 2);
19     }
20 }
21
22 @FunctionalInterface
23 interface T1 {
24     public void m1();
25 }
26
27 @FunctionalInterface
28 interface T2 {

```

inner class, anonymous class,  
or Lambda?

simplify syntax  
simplify concept

```

29     public void m2(Double d);
30 }
31
32 @FunctionalInterface
33 interface T3 {
34     public int m3(int d1, int d2);
35 }

```

The annotation `@FunctionalInterface` tells the compiler that the interface is a functional interface. Since `T1`, `T2`, and `T3` are all functional interfaces, a lambda expression can be used with the methods `setAction1(T1)`, `setAction2(T2)`, and `getValue(T3)`. The statement in line 4 is equivalent to using an anonymous inner class, as follows:

```

test.setAction1(new T1() {
    @Override
    public void m1() {
        System.out.print("Action 1! ");
    }
});

```

- 15.6.1** What is a lambda expression? What is the benefit of using lambda expressions for event handling? What is the syntax of a lambda expression?
- 15.6.2** What is a functional interface? Why is a functional interface required for a lambda expression?
- 15.6.3** Replace the code in lines 5 and 6 in `TestLambda.java` using anonymous inner classes.



## 15.7 Case Study: Loan Calculator

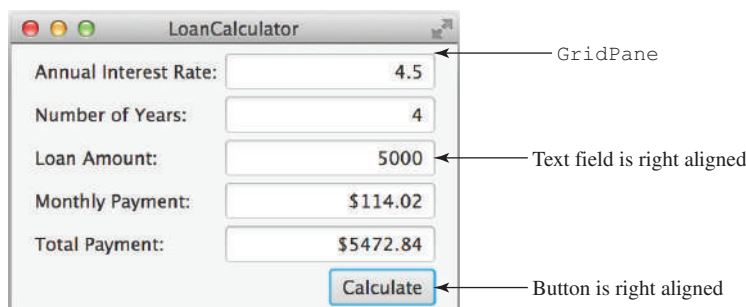
*This case study develops a loan calculator using event-driven programming with GUI controls.*



Now, we will write the program for the loan-calculator problem presented at the beginning of this chapter. Here are the major steps in the program:

1. Create the user interface, as shown in Figure 15.9.
  - a. Create a `GridPane`. Add labels, text fields, and button to the pane.
  - b. Set the alignment of the button to the right.
2. Process the event.

Create and register the handler for processing the button-clicking action event. The handler obtains the user input on the loan amount, interest rate, and number of years, computes the monthly and total payments, and displays the values in the text fields.



**FIGURE 15.9** The program computes loan payments.

The complete program is given in Listing 15.7.

### LISTING 15.7 LoanCalculator.java

```

1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.geometry.HPos;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.control.Label;
7  import javafx.scene.control.TextField;
8  import javafx.scene.layout.GridPane;
9  import javafx.stage.Stage;
10
11 public class LoanCalculator extends Application {
12     private TextField tfAnnualInterestRate = new TextField();
13     private TextField tfNumberOfYears = new TextField();
14     private TextField tfLoanAmount = new TextField();
15     private TextField tfMonthlyPayment = new TextField();
16     private TextField tfTotalPayment = new TextField();
17     private Button btCalculate = new Button("Calculate");
18
19     @Override // Override the start method in the Application class
20     public void start(Stage primaryStage) {
21         // Create UI
22         GridPane gridPane = new GridPane();
23         gridPane.setHgap(5);
24         gridPane.setVgap(5);
25         gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26         gridPane.add(tfAnnualInterestRate, 1, 0);
27         gridPane.add(new Label("Number of Years:"), 0, 1);
28         gridPane.add(tfNumberOfYears, 1, 1);
29         gridPane.add(new Label("Loan Amount:"), 0, 2);
30         gridPane.add(tfLoanAmount, 1, 2);
31         gridPane.add(new Label("Monthly Payment:"), 0, 3);
32         gridPane.add(tfMonthlyPayment, 1, 3);
33         gridPane.add(new Label("Total Payment:"), 0, 4);
34         gridPane.add(tfTotalPayment, 1, 4);
35         gridPane.add(btCalculate, 1, 5);
36
37         // Set properties for UI
38         gridPane.setAlignment(Pos.CENTER);
39         tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40         tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41         tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42         tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43         tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44         tfMonthlyPayment.setEditable(false);
45         tfTotalPayment.setEditable(false);
46         GridPane.setHalignment(btCalculate, HPos.RIGHT);
47
48         // Process events
49         btCalculate.setOnAction(e -> calculateLoanPayment());
50
51         // Create a scene and place it in the stage
52         Scene scene = new Scene(gridPane, 400, 250);
53         primaryStage.setTitle("LoanCalculator"); // Set title
54         primaryStage.setScene(scene); // Place the scene in the stage
55         primaryStage.show(); // Display the stage
56     }

```

text fields

button

create a grid pane

add to grid pane

register handler

```

57
58 private void calculateLoanPayment() {
59     // Get values from text fields
60     double interest =
61         Double.parseDouble(tfAnnualInterestRate.getText());           get input
62     int year = Integer.parseInt(tfNumberOfYears.getText());
63     double loanAmount =
64         Double.parseDouble(tfLoanAmount.getText());
65
66     // Create a loan object. Loan defined in Listing 10.2
67     Loan loan = new Loan(interest, year, loanAmount);                 create loan
68
69     // Display monthly payment and total payment
70     tfMonthlyPayment.setText(String.format("%.2f",
71         loan.getMonthlyPayment()));                                   set result
72     tfTotalPayment.setText(String.format("%.2f",
73         loan.getTotalPayment()));
74 }
75 }

```

The user interface is created in the **start** method (lines 22–46). The button is the source of the event. A handler is created and registered with the button (line 49). The button handler invokes the **calculateLoanPayment()** method to get the interest rate (line 60), number of years (line 62), and loan amount (line 64). Invoking **tfAnnualInterestRate.getText()** returns the string text in the **tfAnnualInterestRate** text field. The **Loan** class is used for computing the loan payments. This class was introduced in Listing 10.2, *Loan.java*. Invoking **loan.getMonthlyPayment()** returns the monthly payment for the loan (line 71). The **String.format** method, introduced in Section 10.10.7, is used to format a number into a desirable format and returns it as a string (lines 70 and 72). Invoking the **setText** method on a text field sets a string value in the text field.

## 15.8 Mouse Events

A **MouseEvent** is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene.

The **MouseEvent** object captures the event, such as the number of clicks associated with it, the location (the *x*- and *y*-coordinates) of the mouse, or which mouse button was pressed, as shown in Figure 15.10.



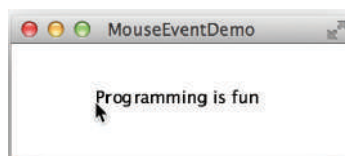
<b>javafx.scene.input.MouseEvent</b>	
+getButton(): MouseButton	Indicates which mouse button has been clicked.
+getClickCount(): int	Returns the number of mouse clicks associated with this event.
+getX(): double	Returns the <i>x</i> -coordinate of the mouse point in the event source node.
+getY(): double	Returns the <i>y</i> -coordinate of the mouse point in the event source node.
+getSceneX(): double	Returns the <i>x</i> -coordinate of the mouse point in the scene.
+getSceneY(): double	Returns the <i>y</i> -coordinate of the mouse point in the scene.
+getScreenX(): double	Returns the <i>x</i> -coordinate of the mouse point in the screen.
+getScreenY(): double	Returns the <i>y</i> -coordinate of the mouse point in the screen.
+isAltDown(): boolean	Returns true if the <b>Alt</b> key is pressed on this event.
+isControlDown(): boolean	Returns true if the <b>Control</b> key is pressed on this event.
+isMetaDown(): boolean	Returns true if the mouse <b>Meta</b> button is pressed on this event.
+isShiftDown(): boolean	Returns true if the <b>Shift</b> key is pressed on this event.

**FIGURE 15.10** The **MouseEvent** class encapsulates information for mouse events.

detect mouse buttons

Four constants—**PRIMARY**, **SECONDARY**, **MIDDLE**, and **NONE**—are defined in **MouseButton** to indicate the left, right, middle, and none mouse buttons, respectively. You can use the **getButton()** method to detect which button is pressed. For example, **getButton() == MouseButton.SECONDARY** tests if the right button was pressed. You can also use the **isPrimaryButtonDown()**, **isSecondaryButtonDown()**, and **isMiddleButtonDown()** to test if the primary button, second button, or middle button is pressed.

The mouse events and their corresponding registration methods for handlers are listed in Table 15.1. To demonstrate using mouse events, we give an example that displays a message in a pane and enables the message to be moved using a mouse. The message moves as the mouse is dragged, and it is always displayed at the mouse point. Listing 15.8 gives the program. A sample run of the program is shown in Figure 15.11.



**FIGURE 15.11** You can move the message by dragging the mouse.



#### VideoNote

Move message using the mouse

create a pane  
create a text  
add text to a pane  
lambda handler  
reset text position

### LISTING 15.8 MouseEventDemo.java

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.text.Text;
5  import javafx.stage.Stage;
6
7  public class MouseEventDemo extends Application {
8      @Override // Override the start method in the Application class
9      public void start(Stage primaryStage) {
10         // Create a pane and set its properties
11         Pane pane = new Pane();
12         Text text = new Text(20, 20, "Programming is fun");
13         pane.getChildren().addAll(text);
14         text.setOnMouseDragged(e -> {
15             text.setX(e.getX());
16             text.setY(e.getY());
17         });
18
19         // Create a scene and place it in the stage
20         Scene scene = new Scene(pane, 300, 100);
21         primaryStage.setTitle("MouseEventDemo"); // Set the stage title
22         primaryStage.setScene(scene); // Place the scene in the stage
23         primaryStage.show(); // Display the stage
24     }
25 }

```

Each node or scene can fire mouse events. The program creates a **Text** (line 12) and registers a handler to handle move dragged event (line 14). Whenever a mouse is dragged, the text's x- and y-coordinates are set to the mouse position (lines 15 and 16).



Check  
Point

- 15.8.1** What method do you use to get the mouse-point position for a mouse event?
- 15.8.2** What methods do you use to register a handler for mouse-pressed, -released, -clicked, -entered, -exited, -moved, and -dragged events?

## 15.9 Key Events

A **KeyEvent** is fired whenever a key is pressed, released, or typed on a node or a scene.



Key events enable the use of the keys to control and perform actions, or get input from the keyboard. The **KeyEvent** object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key, as shown in Figure 15.12.

<b>javafx.scene.input.KeyEvent</b>	
<code>+getCharacter(): String</code>	Returns the character associated with the key in this event.
<code>+getCode(): KeyCode</code>	Returns the key code associated with the key in this event.
<code>+getText(): String</code>	Returns a string describing the key code.
<code>+isAltDown(): boolean</code>	Returns true if the <b>Alt</b> key is pressed on this event.
<code>+isControlDown(): boolean</code>	Returns true if the <b>Control</b> key is pressed on this event.
<code>+isMetaDown(): boolean</code>	Returns true if the mouse <b>Meta</b> button is pressed on this event.
<code>+isShiftDown(): boolean</code>	Returns true if the <b>Shift</b> key is pressed on this event.

**FIGURE 15.12** The **KeyEvent** class encapsulates information about key events.

The key events key pressed, key released, and key typed and their corresponding registration methods for handlers are listed in Table 15.1. The key pressed handler is invoked when a key is pressed, the key released handler is invoked when a key is released, and the key typed handler is invoked when a Unicode character is entered. If a key does not have a Unicode (e.g., function keys, modifier keys, action keys, arrow keys, and control keys), the key typed handler will not be invoked.

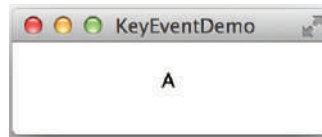
Every key event has an associated code that is returned by the `getCode()` method in **KeyEvent**. The *key codes* are constants defined in **KeyCode**. Table 15.2 lists some constants. **KeyCode** is an **enum** type. For use of **enum** types, see Appendix I. For the key-pressed and key-released events, `getCode()` returns the value as defined in the table, `getText()` returns a string that describes the key code, and `getCharacter()` returns an empty string. For the key-typed event, `getCode()` returns **UNDEFINED** and `getCharacter()` returns the Unicode character or a sequence of characters associated with the key-typed event.

key code

**TABLE 15.2** **KeyCode** Constants

Constant	Description	Constant	Description
<b>HOME</b>	The Home key	<b>CONTROL</b>	The Control key
<b>END</b>	The End key	<b>SHIFT</b>	The Shift key
<b>PAGE_UP</b>	The Page Up key	<b>BACK_SPACE</b>	The Backspace key
<b>PAGE_DOWN</b>	The Page Down key	<b>CAPS</b>	The Caps Lock key
<b>UP</b>	The up-arrow key	<b>NUM_LOCK</b>	The Num Lock key
<b>DOWN</b>	The down-arrow key	<b>ENTER</b>	The Enter key
<b>LEFT</b>	The left-arrow key	<b>UNDEFINED</b>	The <b>keyCode</b> unknown
<b>RIGHT</b>	The right-arrow key	<b>F1 to F12</b>	The function keys from F1 to F12
<b>ESCAPE</b>	The Esc key	<b>0 to 9</b>	The number keys from 0 to 9
<b>TAB</b>	The Tab key	<b>A to Z</b>	The letter keys from A to Z

The program in Listing 15.9 displays a user-input character. The user can move the character up, down, left, and right, using the up-, down-, left-, and right-arrow keys, respectively. Figure 15.13 contains a sample run of the program.



**FIGURE 15.13** The program responds to key events by displaying a character and moving it up, down, left, or right.

### LISTING 15.9 KeyEventDemo.java

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.text.Text;
5  import javafx.stage.Stage;
6
7  public class KeyEventDemo extends Application {
8      @Override // Override the start method in the Application class
9      public void start(Stage primaryStage) {
10         // Create a pane and set its properties
11         Pane pane = new Pane();
12         Text text = new Text(20, 20, "A");
13
14         pane.getChildren().add(text);
15         text.setOnKeyPressed(e -> {
16             switch (e.getCode()) {
17                 case DOWN: text.setY(text.getY() + 10); break;
18                 case UP: text.setY(text.getY() - 10); break;
19                 case LEFT: text.setX(text.getX() - 10); break;
20                 case RIGHT: text.setX(text.getX() + 10); break;
21                 default:
22                     if (e.getText().length() > 0)
23                         text.setText(e.getText());
24             }
25         });
26
27         // Create a scene and place it in the stage
28         Scene scene = new Scene(pane);
29         primaryStage.setTitle("KeyEventDemo"); // Set the stage title
30         primaryStage.setScene(scene); // Place the scene in the stage
31         primaryStage.show(); // Display the stage
32
33         text.requestFocus(); // text is focused to receive key input
34     }
35 }

```

create a pane

register handler  
get the key pressed  
move a character

set a new character

request focus on text

The program creates a pane (line 11), creates a text (line 12), and places the text into the pane (line 14). The text registers the handler for the key-pressed event in lines 15–25. When a key is pressed, the handler is invoked. The program uses `e.getCode()` (line 16) to obtain the key code and `e.getText()` (line 23) to get the character for the key. Note for a nonprintable character such as a CTRL key or SHIFT key, `e.getText()` returns an empty string. When a non-arrow key is pressed, the character is displayed (lines 22 and 23). When an arrow key is pressed, the character moves in the direction indicated by the arrow key (lines 17–20). Note in



a switch statement for an enum-type value, the cases are for the enum constants (lines 16–24). The constants are unqualified. For example, using `KeyCode.DOWN` in the case clause would be wrong (see Appendix I).

Only a focused node can receive `KeyEvent`. Invoking `requestFocus()` on `text` enables `text` to receive key input (line 33). This method must be invoked after the stage is displayed. The program would work fine if `text` is replaced by `scene` in line 15 as follows:

```
scene.setOnKeyPressed(e -> { ... });
```

You don't need to invoke `scene.requestFocus()` because scene is a top-level container for receiving key events.

We can now add more control for our `ControlCircle` example in Listing 15.3 to increase/decrease the circle radius by clicking the left/right mouse button or by pressing the up and down arrow keys. The new program is given in Listing 15.10.

### LISTING 15.10 `ControlCircleWithMouseAndKey.java`

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.input.KeyCode;
6  import javafx.scene.input.MouseButton;
7  import javafx.scene.layout.HBox;
8  import javafx.scene.layout.BorderPane;
9  import javafx.stage.Stage;
10
11 public class ControlCircleWithMouseAndKey extends Application {
12     private CirclePane circlePane = new CirclePane();
13
14     @Override // Override the start method in the Application class
15     public void start(Stage primaryStage) {
16         // Hold two buttons in an HBox
17         HBox hBox = new HBox();
18         hBox.setSpacing(10);
19         hBox.setAlignment(Pos.CENTER);
20         Button btEnlarge = new Button("Enlarge");
21         Button btShrink = new Button("Shrink");
22         hBox.getChildren().add(btEnlarge);
23         hBox.getChildren().add(btShrink);
24
25         // Create and register the handler
26         btEnlarge.setOnAction(e -> circlePane.enlarge());
27         btShrink.setOnAction(e -> circlePane.shrink());
28
29         BorderPane borderPane = new BorderPane();
30         borderPane.setCenter(circlePane);
31         borderPane.setBottom(hBox);
32         borderPane.setAlignment(hBox, Pos.CENTER);
33
34         // Create a scene and place it in the stage
35         Scene scene = new Scene(borderPane, 200, 150);
36         primaryStage.setTitle("ControlCircle"); // Set the stage title
37         primaryStage.setScene(scene); // Place the scene in the stage
38         primaryStage.show(); // Display the stage
39
40         circlePane.setOnMouseClicked(e -> {
41             if (e.getButton() == MouseButton.PRIMARY) {
42                 circlePane.enlarge();
43             }
44         });
```

`requestFocus()`

button handler

mouse-click handler

```

44         else if (e.getButton() == MouseButton.SECONDARY) {
45             circlePane.shrink();
46         }
47     });
48
49     scene.setOnKeyPressed(e -> {
50         if (e.getCode() == KeyCode.UP) {
51             circlePane.enlarge();
52         }
53         else if (e.getCode() == KeyCode.DOWN) {
54             circlePane.shrink();
55         }
56     });
57 }
58 }

```

key-pressed handler  
Up-arrow key pressed

Down-arrow key pressed

The **CirclePane** class (line 12) is already defined in Listing 15.3 and can be reused in this program.

A handler for mouse-clicked events is created in lines 40–47. If the left mouse button is clicked, the circle is enlarged (lines 41–43); if the right mouse button is clicked, the circle is shrunk (lines 44–46).

A handler for key-pressed events is created in lines 49–56. If the up arrow key is pressed, the circle is enlarged (lines 50–52); if the down arrow key is pressed, the circle is shrunk (lines 53–55).



**15.9.1** What methods do you use to register handlers for key-pressed, key-released, and key-typed events? In which classes are these methods defined? (See Table 15.1.)

**15.9.2** What method do you use to get the key character for a key-typed event? What method do you use to get the key code for a key-pressed or key-released event?

**15.9.3** How do you set focus on a node so it can listen for key events?

**15.9.4** If the following code is inserted in line 57 in Listing 15.9, what is the output if the user presses the key for letter A? What is the output if the user presses the up arrow key?

```

circlePane.setOnKeyPressed(e ->
    System.out.println("Key pressed " + e.getCode()));
circlePane.setOnKeyTyped(e ->
    System.out.println("Key typed " + e.getCode()));

```

## 15.10 Listeners for Observable Objects

*You can add a listener to process a value change in an observable object.*



An instance of **Observable** is known as an *observable object*, which contains the **addListener(InvalidationListener listener)** method for adding a listener. The listener class must implement the functional interface **InvalidationListener** to override the **invalidated(Observable o)** method for handling the value change. Once the value is changed in the **Observable** object, the listener is notified by invoking its **invalidated(Observable o)** method. Every binding property is an instance of **Observable**. Listing 15.11 gives an example of observing and handling a change in a **DoubleProperty** object **balance**.

### LISTING 15.11 ObservablePropertyDemo.java

```

1 import javafx.beans.InvalidationListener;
2 import javafx.beans.Observable;
3 import javafx.beans.property.DoubleProperty;

```

observable object

```

4 import javafx.beans.property.SimpleDoubleProperty;
5
6 public class ObservablePropertyDemo {
7     public static void main(String[] args) {
8         DoubleProperty balance = new SimpleDoubleProperty();
9         balance.addListener(new InvalidationListener() {
10             public void invalidated(Observable ov) {
11                 System.out.println("The new value is " +
12                     balance.doubleValue());
13             }
14         });
15
16         balance.set(4.5);
17     }
18 }

```

observable property  
add listener  
handle change

The new value is 4.5



When line 16 is executed, it causes a change in balance, which notifies the listener by invoking the listener's **invalidated** method.

Note the anonymous inner class in lines 9–14 can be simplified using a lambda expression as follows:

```

balance.addListener(ov -> {
    System.out.println("The new value is " +
        balance.doubleValue());
});

```

Listing 15.12 gives a program that displays a circle with its bounding rectangle, as shown in Figure 15.14. The circle and rectangle are automatically resized when the user resizes the window.

### LISTING 15.12 ResizableCircleRectangle.java

```

1 import javafx.application.Application;
2 import javafx.scene.paint.Color;
3 import javafx.scene.shape.Circle;
4 import javafx.scene.shape.Rectangle;
5 import javafx.stage.Stage;
6 import javafx.scene.Scene;
7 import javafx.scene.control.Label;
8 import javafx.scene.layout.StackPane;
9
10 public class ResizableCircleRectangle extends Application {
11     // Create a circle and a rectangle
12     private Circle circle = new Circle(60);
13     private Rectangle rectangle = new Rectangle(120, 120);
14
15     // Place clock and label in border pane
16     private StackPane pane = new StackPane();
17
18     @Override // Override the start method in the Application class
19     public void start(Stage primaryStage) {
20         circle.setFill(Color.GRAY);
21         rectangle.setFill(Color.WHITE);
22         rectangle.setStroke(Color.BLACK);
23         pane.getChildren().addAll(rectangle, circle);

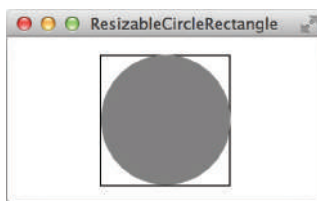
```

```

24
25     // Create a scene and place the pane in the stage
26     Scene scene = new Scene(pane, 140, 140);
27     primaryStage.setTitle("ResizableCircleRectangle");
28     primaryStage.setScene(scene); // Place the scene in the stage
29     primaryStage.show(); // Display the stage
30
31     pane.widthProperty().addListener(ov -> resize());
32     pane.heightProperty().addListener(ov -> resize());
33 }
34
35 private void resize() {
36     double length = Math.min(pane.getWidth(), pane.getHeight());
37     circle.setRadius(length / 2 - 15);
38     rectangle.setWidth(length - 30);
39     rectangle.setHeight(length - 30);
40 }
41 }

```

set a new width for clock  
set a new height for clock



**FIGURE 15.14** The program places a rectangle and a circle inside a stack pane, and automatically sets their sizes when the window is resized.

The program registers the listeners for the stack pane's **width** and **height** properties (lines 31 and 32). When the user resizes the window, the pane's size is changed, so the listeners are called to invoke the **resize()** method to change the size of the circle and rectangle (lines 35–40).



**15.10.1** What would happen if you replace **pane** with **scene** or **primaryStage** in lines 31–32?



## 15.11 Animation

*JavaFX provides the **Animation** class with the core functionality for all animations.*

Suppose you want to write a program that animates a rising flag, as shown in Figure 15.15. How do you accomplish the task? There are several ways to program this. An effective one is to use the subclasses of the JavaFX **Animation** class, which is the subject of this section.



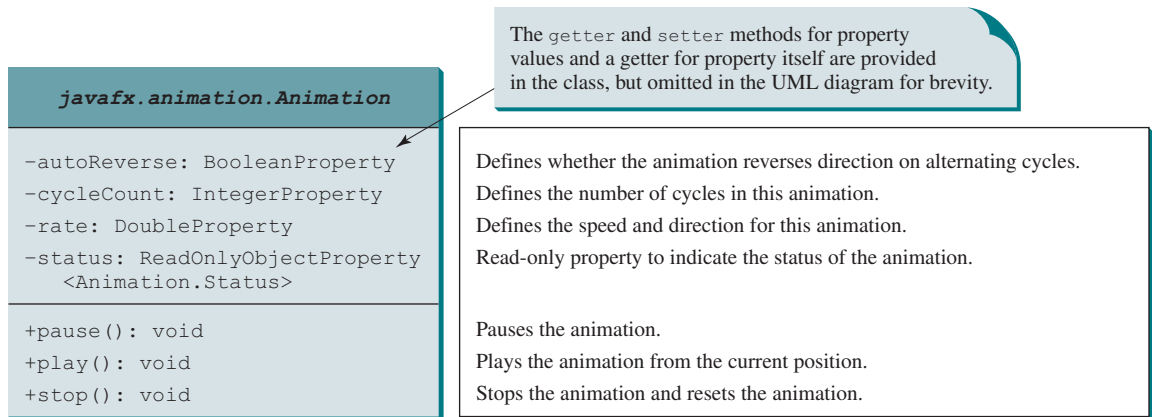
VideoNote

Animate a rising flag



**FIGURE 15.15** The animation simulates a flag rising. *Source:* booka/Fotolia.

The abstract **Animation** class provides the core functionalities for animations in JavaFX, as shown in Figure 15.16. Many concrete subclasses of **Animation** are provided in JavaFX. This section introduces **PathTransition**, **FadeTransition**, and **Timeline**.

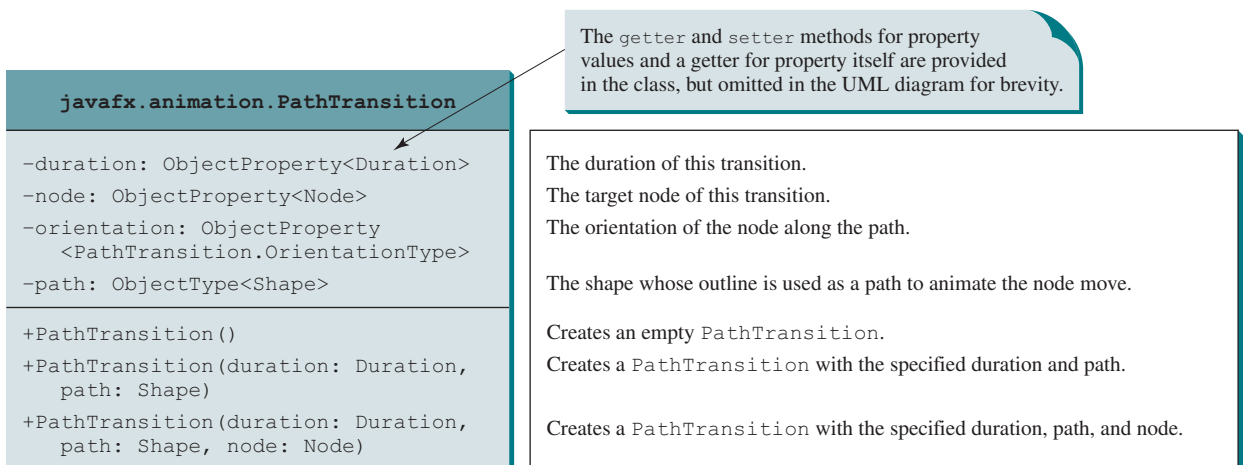


**FIGURE 15.16** The abstract **Animation** class is the root class for JavaFX animations.

The **autoReverse** is a Boolean property that indicates whether an animation will reverse its direction on the next cycle. The **cycleCount** indicates the number of the cycles for the animation. You can use the constant **Timeline.INDEFINITE** to indicate an indefinite number of cycles. The **rate** defines the speed of the animation. A negative rate value indicates the opposite direction for the animation. The **status** is a read-only property that indicates the status of the animation (**Animation.Status.PAUSED**, **Animation.Status.RUNNING**, and **Animation.Status.STOPPED**). The methods **pause()**, **play()**, and **stop()** pause, play, and stop an animation, respectively.

### 15.11.1 PathTransition

The **PathTransition** class animates the moves of a node along a path from one end to the other over a given time. **PathTransition** is a subtype of **Animation**. The UML class diagram for the class is shown in Figure 15.17.



**FIGURE 15.17** The **PathTransition** class defines an animation for a node along a path.

The **Duration** class defines a duration of time. It is an immutable class. The class defines constants **INDEFINITE**, **ONE**, **UNKNOWN**, and **ZERO** to represent an indefinite duration, one millisecond, unknown, and zero duration, respectively. You can use **new Duration(double millis)** to create an instance of **Duration**, the **add**, **subtract**, **multiply**, and **divide** methods to perform arithmetic operations, and the **toHours()**, **toMinutes()**, **toSeconds()**, and **toMillis()** to return the number of hours, minutes, seconds, and milliseconds in this duration, respectively. You can also use **compareTo** to compare two durations.

The constants **NONE** and **ORTHOGONAL\_TO\_TANGENT** are defined in **PathTransition.OrientationType**. The latter specifies that the node is kept perpendicular to the path's tangent along the geometric path.

Listing 15.13 gives an example that moves a rectangle along the outline of a circle, as shown in Figure 15.18a.

### LISTING 15.13 PathTransitionDemo.java

```

1  import javafx.animation.PathTransition;
2  import javafx.animation.Timeline;
3  import javafx.application.Application;
4  import javafx.scene.Scene;
5  import javafx.scene.layout.Pane;
6  import javafx.scene.paint.Color;
7  import javafx.scene.shape.Rectangle;
8  import javafx.scene.shape.Circle;
9  import javafx.stage.Stage;
10 import javafx.util.Duration;
11
12 public class PathTransitionDemo extends Application {
13     @Override // Override the start method in the Application class
14     public void start(Stage primaryStage) {
15         // Create a pane
16         Pane pane = new Pane();
17
18         // Create a rectangle
19         Rectangle rectangle = new Rectangle (0, 0, 25, 50);
20         rectangle.setFill(Color.ORANGE);
21
22         // Create a circle
23         Circle circle = new Circle(125, 100, 50);
24         circle.setFill(Color.WHITE);
25         circle.setStroke(Color.BLACK);
26
27         // Add circle and rectangle to the pane
28         pane.getChildren().add(circle);
29         pane.getChildren().add(rectangle);
30
31         // Create a path transition
32         PathTransition pt = new PathTransition();
33         pt.setDuration(Duration.millis(4000));
34         pt.setPath(circle);
35         pt.setNode(rectangle);
36         pt.setOrientation(
37             PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
38         pt.setCycleCount(Timeline.INDEFINITE);
39         pt.setAutoReverse(true);
40         pt.play(); // Start animation
41
42         circle.setOnMousePressed(e -> pt.pause());
43         circle.setOnMouseReleased(e -> pt.play());

```

create a pane

create a rectangle

create a circle

add circle to pane

add rectangle to pane

create a PathTransition

set transition duration

set path in transition

set node in transition

set orientation

set cycle count indefinite

set auto reverse true

play animation

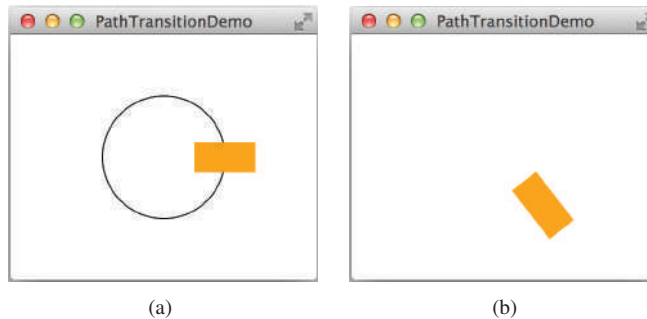
pause animation

resume animation

```

44
45     // Create a scene and place it in the stage
46     Scene scene = new Scene(pane, 250, 200);
47     primaryStage.setTitle("PathTransitionDemo"); // Set the stage title
48     primaryStage.setScene(scene); // Place the scene in the stage
49     primaryStage.show(); // Display the stage
50 }
51 }

```



**FIGURE 15.18** The `PathTransition` animates a rectangle moving along the circle.

The program creates a pane (line 16), a rectangle (line 19), and a circle (line 23). The circle and rectangle are placed in the pane (lines 28 and 29). If the circle was not placed in the pane, you will see the screen shot as shown in Figure 15.18b.

The program creates a path transition (line 32), sets its duration to 4 seconds for one cycle of animation (line 33), sets circle as the path (line 34), sets rectangle as the node (line 35), and sets the orientation to orthogonal to tangent (line 36).

The cycle count is set to indefinite (line 38) so the animation continues forever. The auto reverse is set to true (line 39) so the direction of the move is reversed in the alternating cycle. The program starts animation by invoking the `play()` method (line 40).

If the `pause()` method is replaced by the `stop()` method in line 42, the animation will start over from the beginning when it restarts.

Listing 15.14 gives the program that animates a flag rising, as shown in Figure 15.14.

### LISTING 15.14 `FlagRisingAnimation.java`

```

1  import javafx.animation.PathTransition;
2  import javafx.application.Application;
3  import javafx.scene.Scene;
4  import javafx.scene.image.ImageView;
5  import javafx.scene.layout.Pane;
6  import javafx.scene.shape.Line;
7  import javafx.stage.Stage;
8  import javafx.util.Duration;
9
10 public class FlagRisingAnimation extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         // Create a pane
14         Pane pane = new Pane();
15
16         // Add an image view and add it to pane
17         ImageView imageView = new ImageView("image/us.gif");
18         pane.getChildren().add(imageView);

```

create a pane

create an image view  
add image view to pane



```

19
20 // Create a path transition
create a path transition 21 PathTransition pt = new PathTransition(Duration.millis(10000),
22     new Line(100, 200, 100, 0), imageView);
set cycle count 23 pt.setCycleCount(5);
play animation 24 pt.play(); // Start animation
25
26 // Create a scene and place it in the stage
27 Scene scene = new Scene(pane, 250, 200);
28 primaryStage.setTitle("FlagRisingAnimation"); // Set the stage title
29 primaryStage.setScene(scene); // Place the scene in the stage
30 primaryStage.show(); // Display the stage
31 }
32 }

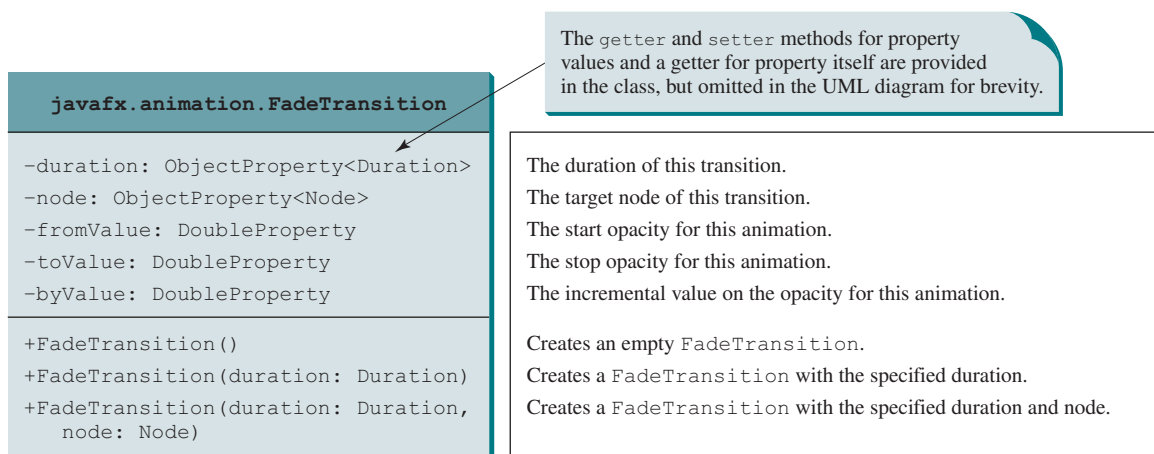
```

The program creates a pane (line 14), an image view from an image file (line 17), and places the image view to the pane (line 18). A path transition is created with a duration of 10 seconds using a line as a path and the image view as the node (lines 21 and 22). The image view will move along the line. Since the line is not placed in the scene, you will not see the line in the window.

The cycle count is set to 5 (line 23) so the animation is repeated five times.

### 15.11.2 FadeTransition

The **FadeTransition** class animates the change of the opacity in a node over a given time. **FadeTransition** is a subtype of **Animation**. The UML class diagram for the class is shown in Figure 15.19.



**FIGURE 15.19** The **FadeTransition** class defines an animation for the change of opacity in a node.

Listing 15.15 gives an example that applies a fade transition to the filled color in an ellipse, as shown in Figure 15.20.

### LISTING 15.15 FadeTransitionDemo.java

```

1 import javafx.animation.FadeTransition;
2 import javafx.animation.Timeline;
3 import javafx.application.Application;
4 import javafx.scene.Scene;
5 import javafx.scene.layout.Pane;
6 import javafx.scene.paint.Color;
7 import javafx.scene.shape.Ellipse;
8 import javafx.stage.Stage;

```

```

9  import javafx.util.Duration;
10
11 public class FadeTransitionDemo extends Application {
12     @Override // Override the start method in the Application class
13     public void start(Stage primaryStage) {
14         // Place an ellipse to the pane
15         Pane pane = new Pane();
16         Ellipse ellipse = new Ellipse(10, 10, 100, 50);
17         ellipse.setFill(Color.RED);
18         ellipse.setStroke(Color.BLACK);
19         ellipse.centerXProperty().bind(pane.widthProperty().divide(2));
20         ellipse.centerYProperty().bind(pane.heightProperty().divide(2));
21         ellipse.radiusXProperty().bind(
22             pane.widthProperty().multiply(0.4));
23         ellipse.radiusYProperty().bind(
24             pane.heightProperty().multiply(0.4));
25         pane.getChildren().add(ellipse);
26
27         // Apply a fade transition to ellipse
28         FadeTransition ft =
29             new FadeTransition(Duration.millis(3000), ellipse);
30         ft.setFromValue(1.0);
31         ft.setToValue(0.1);
32         ft.setCycleCount(Timeline.INDEFINITE);
33         ft.setAutoReverse(true);
34         ft.play(); // Start animation
35
36         // Control animation
37         ellipse.setOnMousePressed(e -> ft.pause());
38         ellipse.setOnMouseReleased(e -> ft.play());
39
40         // Create a scene and place it in the stage
41         Scene scene = new Scene(pane, 200, 150);
42         primaryStage.setTitle("FadeTransitionDemo"); // Set the stage title
43         primaryStage.setScene(scene); // Place the scene in the stage
44         primaryStage.show(); // Display the stage
45     }
46 }

```

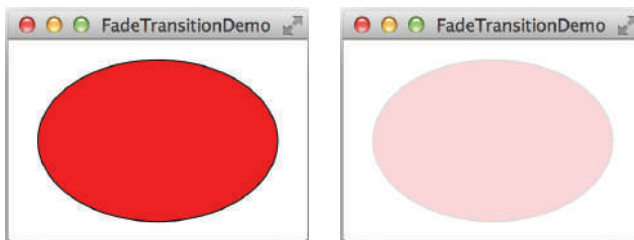
create a pane  
create an ellipse  
set ellipse fill color  
set ellipse stroke color  
bind ellipse properties

add ellipse to pane

create a FadeTransition

set start opaque value  
set end opaque value  
set cycle count  
set auto reverse true  
play animation

pause animation  
resume animation



**FIGURE 15.20** The **FadeTransition** animates the change of opacity in the ellipse.

The program creates a pane (line 15) and an ellipse (line 16) and places the ellipse into the pane (line 25). The ellipse's **centerX**, **centerY**, **radiusX**, and **radiusY** properties are bound to the pane's size (lines 19–24).

A fade transition is created with a duration of 3 seconds for the ellipse (line 29). It sets the start opaque to 1.0 (line 30) and the stop opaque to 0.1 (line 31). The cycle count is set to infinite so the animation is repeated indefinitely (line 32). When the mouse is pressed, the animation is paused (line 37). When the mouse is released, the animation resumes from where it was paused (line 38).

## 15.11.3 Timeline

**PathTransition** and **FadeTransition** define specialized animations. The **Timeline** class can be used to program any animation using one or more **KeyFrame**s. Each **KeyFrame** is executed sequentially at a specified time interval. **Timeline** inherits from **Animation**. You can construct a **Timeline** using the constructor `new Timeline(KeyFrame...keyframes)`. A **KeyFrame** can be constructed using

```
new KeyFrame(Duration duration, EventHandler<ActionEvent> onFinished)
```

The handler **onFinished** is called when the duration for the key frame is elapsed.

Listing 15.15 gives an example that displays a flashing text, as shown in Figure 15.21. The text is on and off alternating to animate flashing.



VideoNote

Flashing text

## LISTING 15.16 TimelineDemo.java

```

1  import javafx.animation.Animation;
2  import javafx.application.Application;
3  import javafx.stage.Stage;
4  import javafx.animation.KeyFrame;
5  import javafx.animation.Timeline;
6  import javafx.event.ActionEvent;
7  import javafx.event.EventHandler;
8  import javafx.scene.Scene;
9  import javafx.scene.layout.StackPane;
10 import javafx.scene.paint.Color;
11 import javafx.scene.text.Text;
12 import javafx.util.Duration;
13
14 public class TimelineDemo extends Application {
15     @Override // Override the start method in the Application class
16     public void start(Stage primaryStage) {
17         StackPane pane = new StackPane();
18         Text text = new Text(20, 50, "Programming is fun");
19         text.setFill(Color.RED);
20         pane.getChildren().add(text); // Place text into the stack pane
21
22         // Create a handler for changing text
23         EventHandler<ActionEvent> eventHandler = e -> {
24             if (text.getText().length() != 0) {
25                 text.setText("");
26             }
27             else {
28                 text.setText("Programming is fun");
29             }
30         };
31
32         // Create an animation for alternating text
33         Timeline animation = new Timeline(
34             new KeyFrame(Duration.millis(500), eventHandler));
35         animation.setCycleCount(Timeline.INDEFINITE);
36         animation.play(); // Start animation
37
38         // Pause and resume animation
39         text.setOnMouseClicked(e -> {
40             if (animation.getStatus() == Animation.Status.PAUSED) {
41                 animation.play();
42             }
43             else {
44                 animation.pause();

```

create a stack pane

create a text

add text to pane

handler for changing text

set text empty

set text

create a Timeline

create a KeyFrame for handler

set cycle count indefinite

play animation

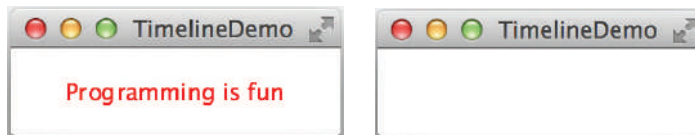
resume animation

pause animation

```

45     }
46   });
47
48   // Create a scene and place it in the stage
49   Scene scene = new Scene(pane, 250, 250);
50   primaryStage.setTitle("TimelineDemo"); // Set the stage title
51   primaryStage.setScene(scene); // Place the scene in the stage
52   primaryStage.show(); // Display the stage
53 }
54 }

```



**FIGURE 15.21** The handler is called to set the text to “Programming is fun” or empty in turn.

The program creates a stack pane (line 17) and a text (line 18) and places the text into the pane (line 20). A handler is created to change the text to empty (lines 24–26) if it is not empty or to **Programming is fun** if it is empty (lines 27–29). A **KeyFrame** is created to run an action event in every half second (line 34). A **Timeline** animation is created to contain a key frame (lines 33 and 34). The animation is set to run indefinitely (line 35).

The mouse-clicked event is set for the text (lines 39–46). A mouse click on the text resumes the animation if the animation is paused (lines 40–42), and a mouse click on the text pauses the animation if the animation is running (lines 43–45).

In Section 14.12, Case Study: The **ClockPane** Class, you drew a clock to show the current time. The clock does not tick after it is displayed. What can you do to make the clock display a new current time every second? The key to making the clock tick is to repaint it every second with a new current time. You can use a **Timeline** to control the repainting of the clock with the code in Listing 15.17. The sample run of the program is shown in Figure 15.22.

### LISTING 15.17 ClockAnimation.java

```

1  import javafx.application.Application;
2  import javafx.stage.Stage;
3  import javafx.animation.KeyFrame;
4  import javafx.animation.Timeline;
5  import javafx.event.ActionEvent;
6  import javafx.event.EventHandler;
7  import javafx.scene.Scene;
8  import javafx.util.Duration;
9
10 public class ClockAnimation extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         ClockPane clock = new ClockPane(); // Create a clock
14
15         // Create a handler for animation
16         EventHandler<ActionEvent> eventHandler = e -> {
17             clock.setCurrentTime(); // Set a new clock time
18         };
19
20         // Create an animation for a running clock
21         Timeline animation = new Timeline(

```

create a clock

create a handler

create a time line

create a key frame  
set cycle count indefinite  
play animation

```

22     new KeyFrame(Duration.millis(1000), eventHandler));
23     animation.setCycleCount(Timeline.INDEFINITE);
24     animation.play(); // Start animation
25
26     // Create a scene and place it in the stage
27     Scene scene = new Scene(clock, 250, 50);
28     primaryStage.setTitle("ClockAnimation"); // Set the stage title
29     primaryStage.setScene(scene); // Place the scene in the stage
30     primaryStage.show(); // Display the stage
31 }
32 }

```

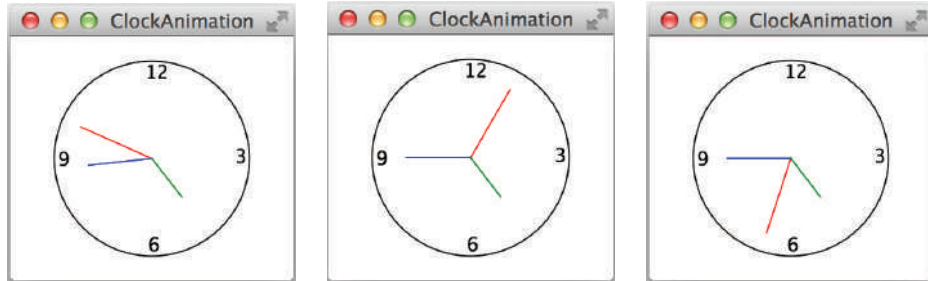


FIGURE 15.22 A live clock is displayed in the window.

The program creates an instance `clock` of `ClockPane` for displaying a clock (line 13). The `ClockPane` class is defined in Listing 14.21. The clock is placed in the scene in line 27. An event handler is created for setting the current time in the clock (lines 16–18). This handler is called every second in the key frame in the time line animation (lines 21–24). Thus, the clock time is updated every second in the animation.



- 15.11.1 How do you set the cycle count of an animation to infinite? How do you auto reverse an animation? How do you start, pause, and stop an animation?
- 15.11.2 Are `PathTransition`, `FadeTransition`, and `Timeline` subtypes of `Animation`?
- 15.11.3 How do you create a `PathTransition`? How do you create a `FadeTransition`? How do you create a `Timeline`?
- 15.11.4 How do you create a `KeyFrame`?

## 15.12 Case Study: Bouncing Ball

*This section presents an animation that displays a ball bouncing in a pane.*



The program uses `Timeline` to animate ball bouncing, as shown in Figure 15.23.

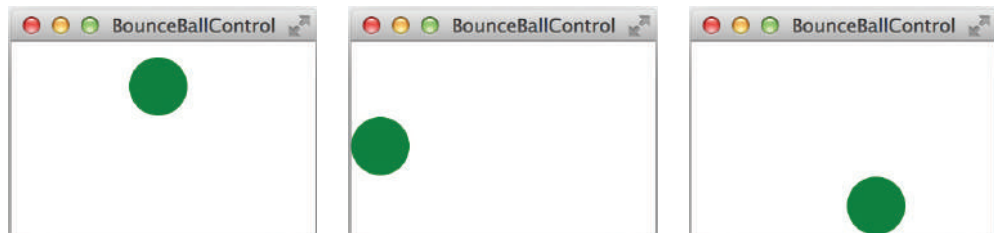


FIGURE 15.23 A ball is bouncing in a pane.

Here are the major steps to write this program:

1. Define a subclass of **Pane** named **BallPane** to display a ball bouncing, as shown in Listing 15.18.
2. Define a subclass of **Application** named **BounceBallControl** to control the bouncing ball with mouse actions, as shown in Listing 15.19. The animation pauses when the mouse is pressed, and resumes when the mouse is released. Pressing the up and down arrow keys increases/decreases the animation speed.

The relationship among these classes is shown in Figure 15.24.

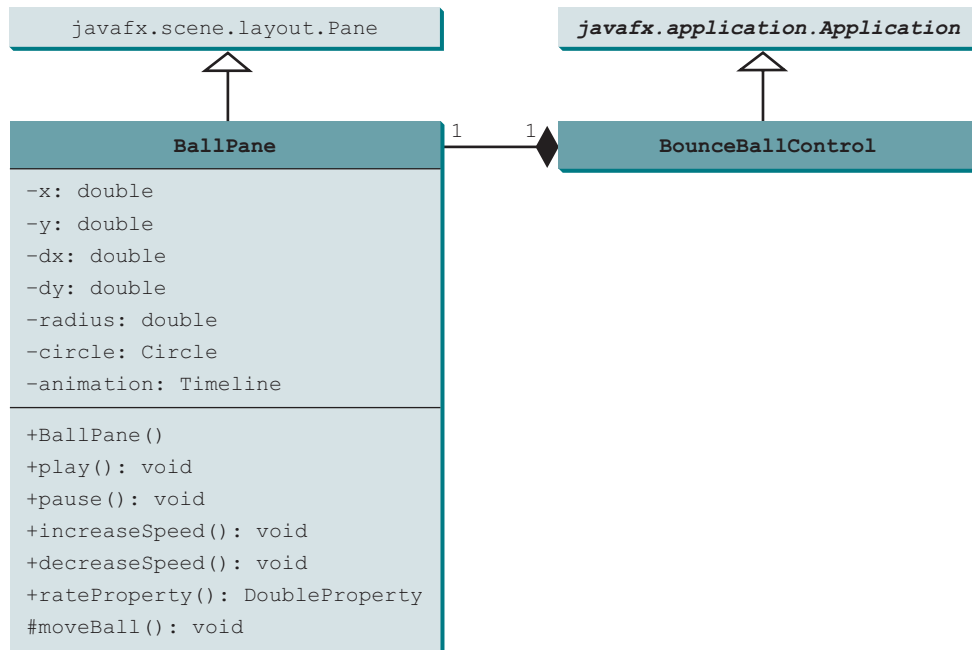


FIGURE 15.24 **BounceBallControl** contains **BallPane**.

### LISTING 15.18 BallPane.java

```

1  import javafx.animation.KeyFrame;
2  import javafx.animation.Timeline;
3  import javafx.beans.property.DoubleProperty;
4  import javafx.scene.layout.Pane;
5  import javafx.scene.paint.Color;
6  import javafx.scene.shape.Circle;
7  import javafx.util.Duration;
8
9  public class BallPane extends Pane {
10     public final double radius = 20;
11     private double x = radius, y = radius;
12     private double dx = 1, dy = 1;
13     private Circle circle = new Circle(x, y, radius);
14     private Timeline animation;
15
16     public BallPane() {
17         circle.setFill(Color.GREEN); // Set ball color
18         getChildren().add(circle); // Place a ball into this pane
  
```

```

19
20     // Create an animation for moving the ball
create animation 21     animation = new Timeline(
22         new KeyFrame(Duration.millis(50), e -> moveBall()));
keep animation running 23     animation.setCycleCount(Timeline.INDEFINITE);
start animation 24     animation.play(); // Start animation
25 }
26
27 public void play() {
play animation 28     animation.play();
29 }
30
31 public void pause() {
pause animation 32     animation.pause();
33 }
34
35 public void increaseSpeed() {
increase animation rate 36     animation.setRate(animation.getRate() + 0.1);
37 }
38
39 public void decreaseSpeed() {
decrease animation rate 40     animation.setRate(
41         animation.getRate() > 0 ? animation.getRate() - 0.1 : 0);
42 }
43
44 public DoubleProperty rateProperty() {
45     return animation.rateProperty();
46 }
47
48 protected void moveBall() {
49     // Check boundaries
change horizontal direction 50     if (x < radius || x > getWidth() - radius) {
51         dx *= -1; // Change ball move direction
52     }
change vertical direction 53     if (y < radius || y > getHeight() - radius) {
54         dy *= -1; // Change ball move direction
55     }
56
57     // Adjust ball position
set new ball position 58     x += dx;
59     y += dy;
60     circle.setCenterX(x);
61     circle.setCenterY(y);
62 }
63 }

```

**BallPane** extends **Pane** to display a moving ball (line 9). An instance of **Timeline** is created to control animation (lines 21 and 22). This instance contains a **KeyFrame** object that invokes the **moveBall()** method at a fixed rate. The **moveBall()** method moves the ball to simulate animation. The center of the ball is at (**x**, **y**), which changes to (**x + dx**, **y + dy**) on the next move (lines 58–61). When the ball is out of the horizontal boundary, the sign of **dx** is changed (from positive to negative or vice versa) (lines 50–52). This causes the ball to change its horizontal movement direction. When the ball is out of the vertical boundary, the sign of **dy** is changed (from positive to negative or vice versa) (lines 53–55). This causes the ball to change its vertical movement direction. The **pause** and **play** methods (lines 27–33) can be used to pause and resume the animation. The **increaseSpeed()** and **decreaseSpeed()** methods (lines 35–42) can be used to increase and decrease animation speed. The **rateProperty()**



method (lines 44–46) returns a binding property value for rate. This binding property will be useful for binding the rate in future applications in the next chapter.

### LISTING 15.19 BounceBallControl.java

```

1  import javafx.application.Application;
2  import javafx.stage.Stage;
3  import javafx.scene.Scene;
4  import javafx.scene.input.KeyCode;
5
6  public class BounceBallControl extends Application {
7      @Override // Override the start method in the Application class
8      public void start(Stage primaryStage) {
9          BallPane ballPane = new BallPane(); // Create a ball pane           create a ball pane
10
11          // Pause and resume animation
12          ballPane.setOnMousePressed(e -> ballPane.pause());               pause animation
13          ballPane.setOnMouseReleased(e -> ballPane.play());               resume animation
14
15          // Increase and decrease animation
16          ballPane.setOnKeyPressed(e -> {
17              if (e.getCode() == KeyCode.UP) {
18                  ballPane.increaseSpeed();                               increase speed
19              }
20              else if (e.getCode() == KeyCode.DOWN) {
21                  ballPane.decreaseSpeed();                               decrease speed
22              }
23          });
24
25          // Create a scene and place it in the stage
26          Scene scene = new Scene(ballPane, 250, 150);
27          primaryStage.setTitle("BounceBallControl"); // Set the stage title
28          primaryStage.setScene(scene); // Place the scene in the stage
29          primaryStage.show(); // Display the stage
30
31          // Must request focus after the primary stage is displayed
32          ballPane.requestFocus();                                         request focus on pane
33      }
34  }
```

The **BounceBallControl** class is the main JavaFX class that extends **Application** to display the ball pane with control functions. The mouse-pressed and mouse-released handlers are implemented for the ball pane to pause the animation and resume the animation (lines 12 and 13). When the UP arrow key is pressed, the ball pane's **increaseSpeed()** method is invoked to increase the ball's movement (line 18). When the down arrow key is pressed, the ball pane's **decreaseSpeed()** method is invoked to reduce the ball's movement (line 21).

Invoking **ballPane.requestFocus()** in line 32 sets the input focus to **ballPane**.

- 15.12.1** How does the program make the ball appear to be moving?
- 15.12.2** How does the code in Listing 15.17, **BallPane.java**, change the direction of the ball movement?
- 15.12.3** What does the program do when the mouse is pressed on the ball pane? What does the program do when the mouse is released on the ball pane?
- 15.12.4** If line 32 in Listing 15.18, **BounceBall.java**, is not in the program, what would happen when you press the up or the down arrow key?
- 15.12.5** If line 23 is not in Listing 15.17, what would happen?

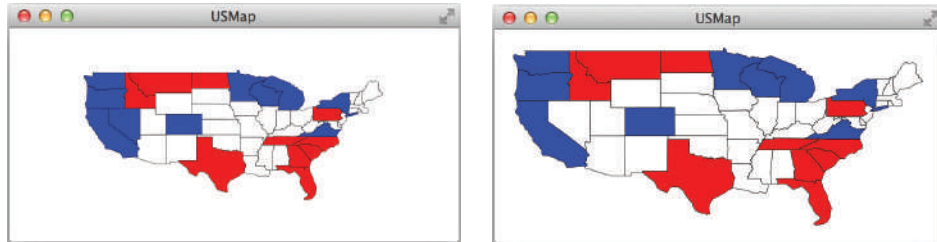


## 15.13 Case Study: US Map



*This section presents a program that draws, colors, and resizes a US map.*

The program reads the GPS coordinates for each state in the 48 continental United States, and draws a polygon to connect the coordinates and displays all the polygons, as shown in Figure 15.25.



**FIGURE 15.25** The program displays, colors, and resizes the US map.

The coordinates are contained in a file at <https://liveexample.pearsoncmg.com/data/usmap.txt>. For each state, the file contains the state name (e.g., Alabama) and all the coordinates (latitude and longitude) for the state. For example, the following is an example for Alabama and Arkansas:

```
Alabama
  35.0041 -88.1955
  34.9918 -85.6068
  ...
  34.9479 -88.1721
  34.9107 -88.1461
Arkansas
  33.0225 -94.0416
  33.0075 -91.2057
  ...
```

A polygon is displayed in red, blue, or white when the primary, secondary, or middle mouse button is clicked in the polygon. The map size is increased when the up arrow key is pressed, and decreased when the down arrow key is pressed. Listing 15.20 gives the code for this program.

### LISTING 15.20 USMap.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.paint.Color;
4  import javafx.stage.Stage;
5  import javafx.scene.shape.Polygon;
6  import javafx.scene.Group;
7  import javafx.scene.layout.BorderPane;
8  import javafx.scene.input.*;
9  import javafx.geometry.Point2D;
10 import java.util.*;
11
12 public class USMap extends Application {
13     @Override // Override the start method in the Application class
14     public void start(Stage primaryStage) {
15         MapPane map = new MapPane();
```

```

16 Scene scene = new Scene(map, 1200, 800);
17 primaryStage.setTitle("USMap"); // Set the stage title
18 primaryStage.setScene(scene); // Place the scene in the stage
19 primaryStage.show(); // Display the stage
20
21 map.setOnKeyPressed(e -> {
22     if (e.getCode() == KeyCode.UP) {
23         map.enlarge(); // Enlarge the map
24     }
25     else if (e.getCode() == KeyCode.DOWN) {
26         map.shrink(); // SHrink the map
27     }
28 });
29 map.requestFocus();
30 }
31
32 class MapPane extends BorderPane {
33     private Group group = new Group();
34
35     MapPane() {
36         // Load coordinates from a file
37         ArrayList<ArrayList<Point2D>> points = getPoints();
38
39         // Add points to the polygon list
40         for (int i = 0; i < points.size(); i++) {
41             Polygon polygon = new Polygon();
42             // Add points to the polygon list
43             for (int j = 0; j < points.get(i).size(); j++)
44                 polygon.getPoints().addAll(points.get(i).get(j).getX(),
45                     -points.get(i).get(j).getY());
46             polygon.setFill(Color.WHITE);
47             polygon.setStroke(Color.BLACK);
48             polygon.setStrokeWidth(1 / 14.0);
49
50             polygon.setOnMouseClicked(e -> {
51                 if (e.getButton() == MouseButton.PRIMARY) {
52                     polygon.setFill(Color.RED);
53                 }
54                 else if (e.getButton() == MouseButton.SECONDARY) {
55                     polygon.setFill(Color.BLUE);
56                 }
57                 else {
58                     polygon.setFill(Color.WHITE);
59                 }
60             });
61
62             group.getChildren().add(polygon);
63         }
64
65         group.setScaleX(14);
66         group.setScaleY(14);
67         this.setCenter(group);
68     }
69
70     public void enlarge() {
71         group.setScaleX(1.1 * group.getScaleX());
72         group.setScaleY(1.1 * group.getScaleY());
73     }
74
75     public void shrink() {

```

listen to key event

enlarge map

shrink map

request focus

extends BorderPane  
create a Group

get coordinates for state

add coordinates

set polygon stroke width

set listener for mouse click  
color polygon

add a polygon to group

scale polygon

center group in the map

enlarge map

shrink map

```

76         group.setScaleX(0.9 * group.getScaleX());
77         group.setScaleY(0.9 * group.getScaleY());
78     }
79
80     private ArrayList<ArrayList<Point2D>> getPoints() {
81         ArrayList<ArrayList<Point2D>> points = new ArrayList<>();
82
83         try (Scanner input = new Scanner(new java.net.URL(
84             "https://liveexample.pearsoncmg.com/data/usmap.txt")
85             .openStream())) {
86             while (input.hasNext()) {
87                 String s = input.nextLine();
88                 if (Character.isAlphabetic(s.charAt(0))) {
89                     points.add(new ArrayList<>()); // For a new state
90                 }
91                 else {
92                     Scanner scanAStrng = new Scanner(s); // Scan one point
93                     double y = scanAStrng.nextDouble();
94                     double x = scanAStrng.nextDouble();
95                     points.get(points.size() - 1).add(new Point2D(x, y));
96                 }
97             }
98         }
99         catch (Exception ex) {
100             ex.printStackTrace();
101         }
102
103         return points;
104     }
105 }
106 }

```

create array list

try-with-resource  
open an Internet resource

read a string  
start a state  
create a state list

read latitude value  
read longitude value  
add a point to list

return list of points

The program defines **MapPane** that extends **BorderPane** to display a map in the center of the border pane (line 32). The program needs to resize the polygons in the map. An instance of the **Group** class is created to hold all the polygons (line 33). Grouping the polygons enables all polygons to be resized in one operation. Resizing the group will cause all polygons in the group to resize accordingly. Resizing can be done by applying the **scaleX** and **scaleY** properties in the group (lines 65 and 66).

The **getPoints()** method is used to return all the coordinates in an array list (line 80). The array list consists of sublists. Each sublist contains the coordinates for a state and is added to the array list (line 89). A **Point2D** object represents the *x*- and *y*-coordinates of the point (line 81). The method creates a **Scanner** object to read data for the map coordinates from a file on the Internet (lines 83–85). The program reads lines from the file. For each line, if the first character is an alphabet, the line is for a new state name (line 88) and a new sublist is created and added to the **points** array list (line 89). Otherwise, the line contains the two coordinates. The latitude becomes the *y*-coordinate for the point (line 93), and the longitude corresponds to the *x*-coordinate of the point (line 94). The program stores the points for a state in a sublist (line 95). **points** is an array list that contains 48 sublists.

The constructor of **MapPane** obtains sublists of the coordinates from the file (line 37). For each sublist of the points, a polygon is created (line 41). The points are added to the polygon (lines 43–45). Since the *y*-coordinates increase upward in the conventional coordinate system, but downward in the Java coordinate system, the program changes the sign for the *y*-coordinates in line 45. The polygon properties are set in lines 46–48. Note the **strokeWidth** is set to **1 / 14.0** (line 48) because all the polygons are scaled up 14 times in lines 65 and 66. If the **strokeWidth** is not set to this value, the stroke width will be very thick. Since polygons are very small, applying the **setScaleX** and **setScaleY** methods on the group causes all the

the Group class

the **scaleX** property

the **scaleY** property

the **scaleX** property

the **scaleY** property

nodes inside the group to be enlarged (lines 65 and 66). **MapPane** is a **BorderPane**. The group is placed in the center of the border pane (line 67).

The **enlarge()** and **shrink()** methods are defined in **MapPane** (lines 70–78). They can be called to enlarge or shrink the group to cause all the polygons in the group to scale up or down.

Each polygon is set to listen to mouse-clicked event (lines 50–60). When clicking the primary/secondary/middle mouse button on a polygon, the polygon is filled red/blue/white.

The program creates an instance of **MapPane** (line 15) and places it in the scene (line 16). The map listens to the key-pressed event to enlarge or shrink the map upon pressing the up and down arrow key (lines 21–28). Since the map is inside the scene, invoking **map.requestFocus()** enables the map to receive key events (line 29).

**15.13.1** What would happen if line 29 in Listing 15.20 is removed?

**15.13.2** What would happen if **map** is replaced by **scene** in line 21 in Listing 15.20?

**15.13.3** What would happen if **map** is replaced by **primaryStage** in line 21 in Listing 15.20?



## KEY TERMS

anonymous inner class	602	functional interface	607
event	596	inner class	599
event-driven programming	596	key code	613
event handler	597	lambda expression	605
event-handler interface	597	observable object	616
event object	596	single abstract method interface	607
event source object	596		

## CHAPTER SUMMARY

1. The root class of the JavaFX event classes is **javafx.event.Event**, which is a subclass of **java.util.EventObject**. The subclasses of **Event** deal with special types of events, such as action events, window events, mouse events, and key events. If a node can fire an event, any subclass of the node can fire the same type of event.
2. The handler object's class must implement the corresponding *event-handler interface*. JavaFX provides a handler interface **EventHandler<T extends Event>** for every event class **T**. The handler interface contains the **handle(T e)** method for handling event **e**.
3. The handler object must be registered by the *source object*. Registration methods depend on the event type. For an action event, the method is **setOnAction**. For a mouse-pressed event, the method is **setOnMousePressed**. For a key-pressed event, the method is **setOnKeyPressed**.
4. An *inner class*, or *nested class*, is defined within the scope of another class. An inner class can reference the data and methods defined in the outer class in which it nests, so you need not pass the reference of the outer class to the constructor of the inner class.
5. An anonymous inner class can be used to shorten the code for event handling. Furthermore, a lambda expression can be used to greatly simplify the event-handling code for functional interface handlers.

6. A *functional interface* is an interface with exactly one abstract method. This is also known as a single abstract method (SAM) interface.
7. A **MouseEvent** is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene. The **getButton()** method can be used to detect which mouse button is pressed for the event.
8. A **KeyEvent** is fired whenever a key is pressed, released, or typed on a node or a scene. The **getCode()** method can be used to return the code value for the key.
9. An instance of **Observable** is known as an observable object, which contains the **addListener(InvalidationListener listener)** method for adding a listener. Once the value is changed in the property, a listener is notified. The listener class should implement the **InvalidationListener** interface, which uses the **invalidated** method to handle the property value change.
10. The abstract **Animation** class provides the core functionalities for animations in JavaFX. **PathTransition**, **FadeTransition**, and **Timeline** are specialized classes for implementing animations.



## Quiz

Answer the quiz for this chapter online at the book Companion Website.

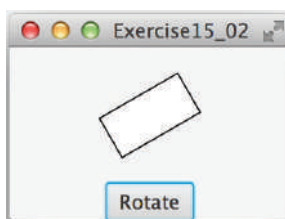
## MyProgrammingLab™ PROGRAMMING EXERCISES

### Sections 15.2–15.7

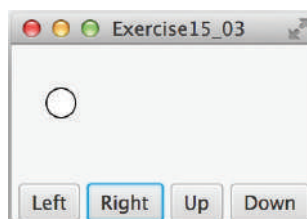
- \*15.1** (*Pick four cards*) Write a program that lets the user click the *Refresh* button to display four cards from a deck of 52 cards, as shown in Figure 15.26a. (See the hint in Programming Exercise 14.3 on how to obtain four random cards.)



(a)



(b)



(c)

**FIGURE 15.26** (a) Exercise 15.1 displays four cards randomly. *Source:* Fotolia. (b) Exercise 15.2 rotates the rectangle. (c) Exercise 15.3 uses the buttons to move the ball.

- 15.2** (*Rotate a rectangle*) Write a program that rotates a rectangle 15 degrees to the right when the *Rotate* button is clicked, as shown in Figure 15.26b.

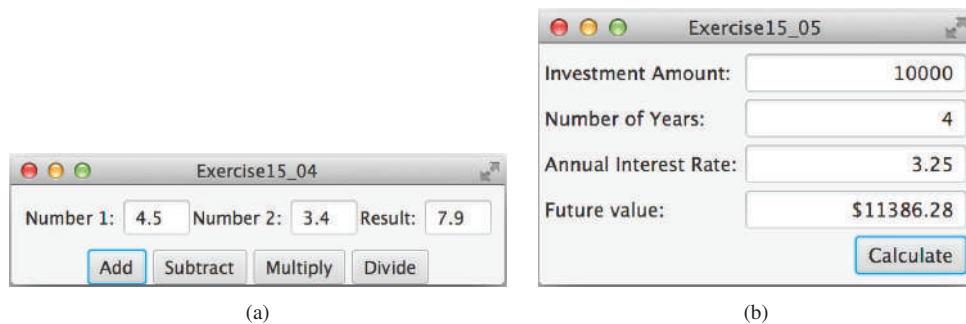
- \*15.3** (*Move the ball*) Write a program that moves the ball in a pane. You should define a pane class for displaying the ball and provide the methods for moving the ball left, right, up, and down, as shown in Figure 15.26c. Check the boundary to prevent the ball from moving out of sight completely.

- \*15.4** (*Create a simple calculator*) Write a program to perform addition, subtraction, multiplication, and division, as shown in Figure 15.27a.



VideoNote

Simple calculator



**FIGURE 15.27** (a) Exercise 15.4 performs addition, subtraction, multiplication, and division on double numbers. (b) The user enters the investment amount, years, and interest rate to compute future value.

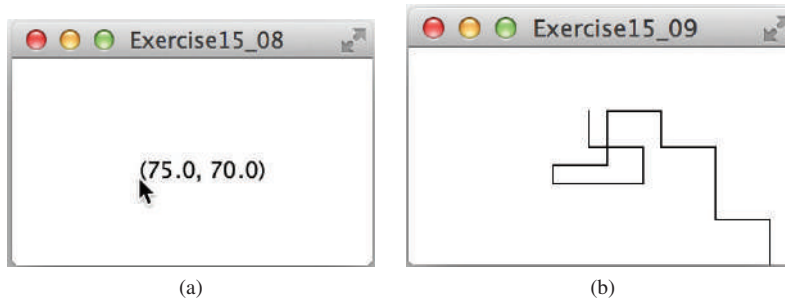
- \*15.5** (*Create an investment-value calculator*) Write a program that calculates the future value of an investment at a given interest rate for a specified number of years. The formula for the calculation is

$$\text{futureValue} = \text{investmentAmount} * (1 + \text{monthlyInterestRate})^{\text{years} * 12}$$

Use text fields for the investment amount, number of years, and annual interest rate. Display the future amount in a text field when the user clicks the *Calculate* button, as shown in Figure 15.27b.

## Sections 15.8 and 15.9

- \*\*15.6** (*Alternate two messages*) Write a program to display the text **Java is fun** and **Java is powerful** alternately with a mouse click.
- \*15.7** (*Change color using a mouse*) Write a program that displays the color of a circle as black when the mouse button is pressed, and as white when the mouse button is released.
- \*15.8** (*Display the mouse position*) Write two programs, such that one displays the mouse position when the mouse button is clicked (see Figure 15.28a), and the other displays the mouse position when the mouse button is pressed and ceases to display it when the mouse button is released.
- \*15.9** (*Draw lines using the arrow keys*) Write a program that draws line segments using the arrow keys. The line starts from (100, 100) in the pane and draws toward east, north, west, or south when the right-arrow key, up-arrow key, left-arrow key, or down-arrow key is pressed, as shown in Figure 15.28b.



**FIGURE 15.28** (a) Exercise 15.8 displays the mouse position. (b) Exercise 15.9 uses the arrow keys to draw the lines.

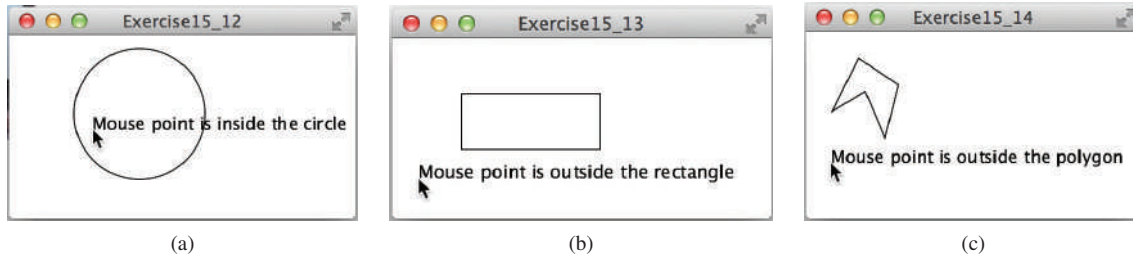




VideoNote

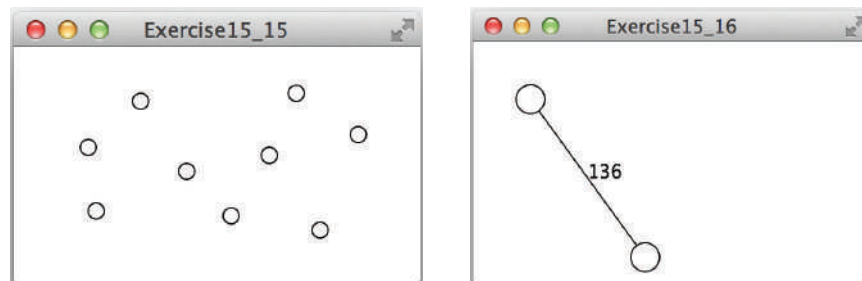
Check mouse-point location

- \*\*15.10** (*Enter and display a string*) Write a program that receives a string from the keyboard and displays it on a pane. The *Enter* key signals the end of a string. Whenever a new string is entered, it is displayed on the pane.
- \*15.11** (*Move a circle using keys*) Write a program that moves a circle up, down, left, or right using the arrow keys.
- \*\*15.12** (*Geometry: inside a circle?*) Write a program that draws a fixed circle centered at (100, 60) with radius 50. Whenever the mouse is moved, display a message indicating whether the mouse point is inside the circle at the mouse point or outside of it, as shown in Figure 15.29a.
- \*\*15.13** (*Geometry: inside a rectangle?*) Write a program that draws a fixed rectangle centered at (100, 60) with width 100 and height 40. Whenever the mouse is moved, display a message indicating whether the mouse point is inside the rectangle at the mouse point or outside of it, as shown in Figure 15.29b. To detect whether a point is inside a polygon, use the `contains` method defined in the `Node` class.



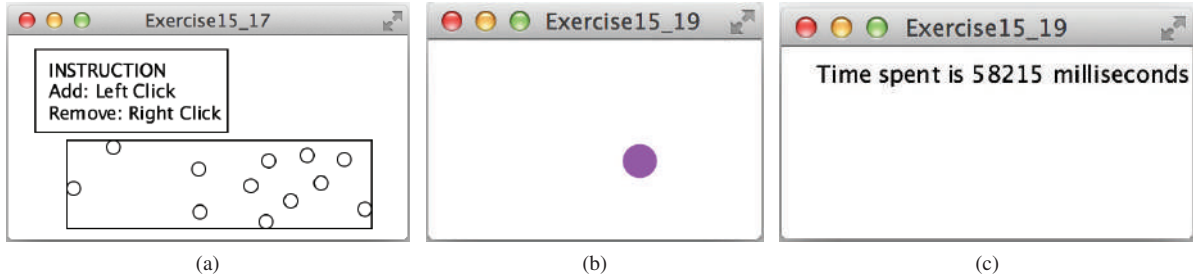
**FIGURE 15.29** Detect whether a point is inside a circle, a rectangle, or a polygon.

- \*\*15.14** (*Geometry: inside a polygon?*) Write a program that draws a fixed polygon with points at (40, 20), (70, 40), (60, 80), (45, 45), and (20, 60). Whenever the mouse is moved, display a message indicating whether the mouse point is inside the polygon at the mouse point or outside of it, as shown in Figure 15.29c. To detect whether a point is inside a polygon, use the `contains` method defined in the `Node` class.
- \*\*15.15** (*Geometry: add and remove points*) Write a program that lets the user click on a pane to dynamically create and remove points (see Figure 15.30a). When the user left-clicks the mouse (primary button), a point is created and displayed at the mouse point. The user can remove a point by pointing to it and right-clicking the mouse (secondary button).



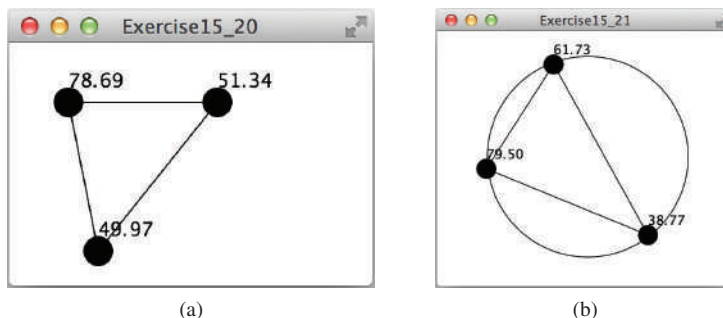
**FIGURE 15.30** (a) Exercise 15.15 allows the user to create/remove points dynamically. (b) Exercise 15.16 displays two vertices and a connecting edge.

- \*15.16** (*Two movable vertices and their distances*) Write a program that displays two circles with radius 10 at location (40, 40) and (120, 150) with a line connecting the two circles, as shown in Figure 15.30b. The distance between the circles is displayed along the line. The user can drag a circle. When that happens, the circle and its line are moved, and the distance between the circles is updated.
- \*\*15.17** (*Geometry: find the bounding rectangle*) Write a program that enables the user to add and remove points in a two-dimensional plane dynamically, as shown in Figure 15.31a. A minimum bounding rectangle is updated as the points are added and removed. Assume the radius of each point is 10 pixels.



**FIGURE 15.31** (a) Exercise 15.17 enables the user to add/remove points dynamically and displays the bounding rectangle. (b) When you click a circle, a new circle is displayed at a random location. (c) After 20 circles are clicked, the time spent is displayed in the pane.

- \*\*15.18** (*Move a rectangle using mouse*) Write a program that displays a rectangle. You can point the mouse inside the rectangle and drag (i.e., move with mouse pressed) the rectangle wherever the mouse goes. The mouse point becomes the center of the rectangle.
- \*\*15.19** (*Game: eye-hand coordination*) Write a program that displays a circle of radius 10 pixels filled with a random color at a random location on a pane, as shown in Figure 15.31b. When you click the circle, it disappears and a new random-color circle is displayed at another random location. After 20 circles are clicked, display the time spent in the pane, as shown in Figure 15.31c.
- \*\*15.20** (*Geometry: display angles*) Write a program that enables the user to drag the vertices of a triangle and displays the angles dynamically as the triangle shape changes, as shown in Figure 15.32a. The formula to compute angles is given in Listing 4.1.



**FIGURE 15.32** (a) Exercise 15.20 enables the user to drag vertices and display the angles dynamically. (b) Exercise 15.21 enables the user to drag vertices and display the angles in the triangle dynamically.

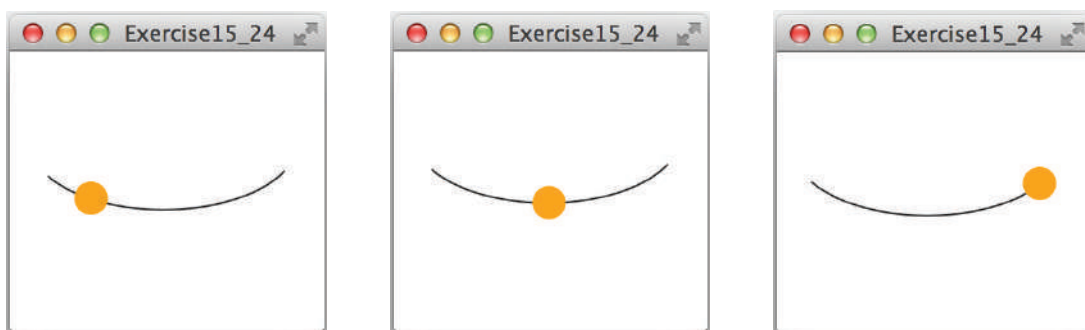
- \*15.21** (*Drag points*) Draw a circle with three random points on the circle. Connect the points to form a triangle. Display the angles in the triangle. Use the mouse to drag a point along the perimeter of the circle. As you drag it, the triangle and angles are redisplayed dynamically, as shown in Figure 15.32b. For computing angles in a triangle, see Listing 4.1.

### Section 15.10

- \*15.22** (*Auto resize cylinder*) Rewrite Programming Exercise 14.10 so the cylinder's width and height are automatically resized when the window is resized.
- \*15.23** (*Auto resize stop sign*) Rewrite Programming Exercise 14.15 so the stop sign's width and height are automatically resized when the window is resized.

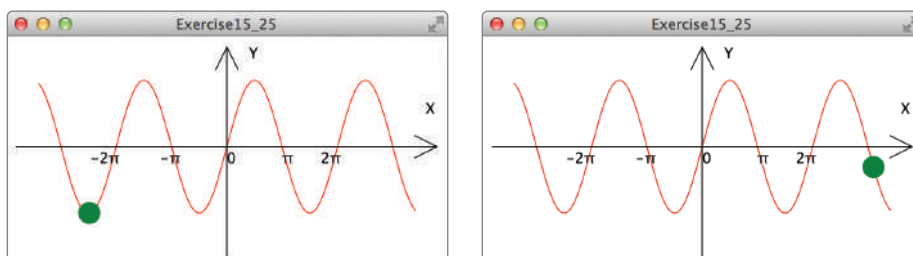
### Section 15.11

- \*\*15.24** (*Animation: pendulum swing*) Write a program that animates a pendulum swing, as shown in Figure 15.33. Press/release the mouse to pause/resume the animation.



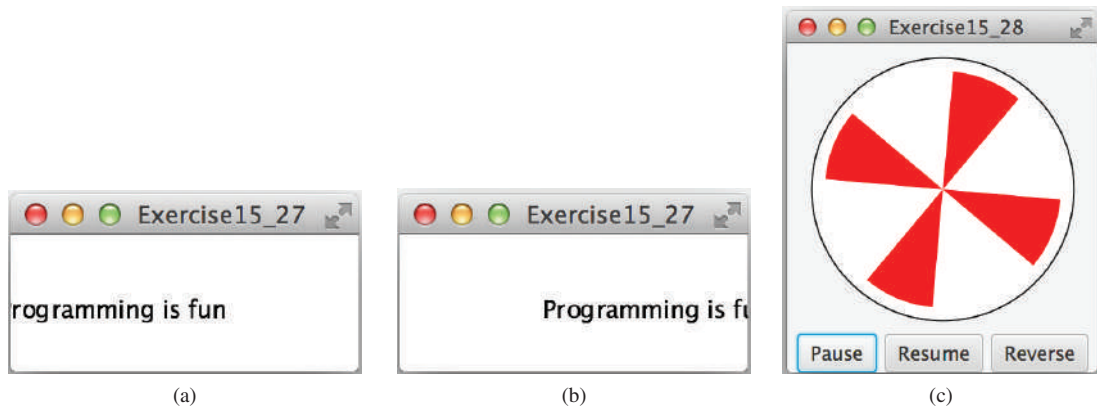
**FIGURE 15.33** The program animates a pendulum swing.

- \*\*15.25** (*Animation: ball on curve*) Write a program that animates a ball moving along a sine curve, as shown in Figure 15.34. When the ball gets to the right border, it starts over from the left. Enable the user to resume/pause the animation with a click on the left/right mouse button.



**FIGURE 15.34** The program animates a ball traveling along a sine curve.

- \*15.26** (*Change opacity*) Rewrite Programming Exercise 15.24 so the ball's opacity is changed as it swings.
- \*15.27** (*Control a moving text*) Write a program that displays a moving text, as shown in Figures 15.35a and b. The text moves from left to right circularly. When it disappears in the right, it reappears from the left. The text freezes when the mouse is pressed, and moves again when the button is released.



**FIGURE 15.35** (a and b) A text is moving from left to right circularly. (c) The program simulates a fan running.

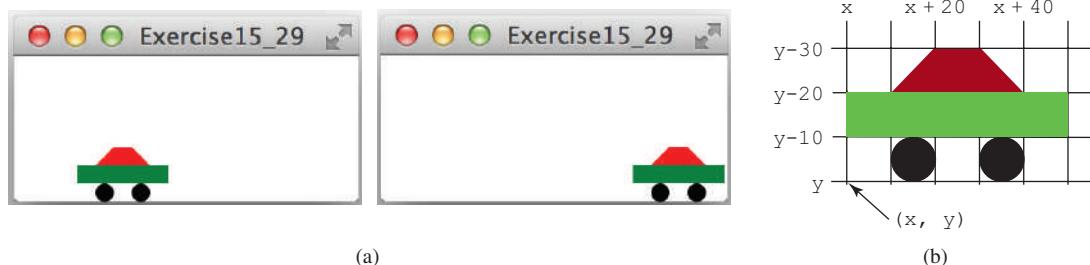
**\*\*15.28** (*Display a running fan*) Write a program that displays a running fan, as shown in Figure 15.35c. Use the *Pause*, *Resume*, and *Reverse* buttons to pause, resume, and reverse fan running.



VideoNote

Display a running fan

**\*\*15.29** (*Racing car*) Write a program that simulates car racing, as shown in Figure 15.36a. The car moves from left to right. When it hits the right end, it restarts from the left and continues the same process. You can use a timer to control animation. Redraw the car with new base coordinates  $(x, y)$ , as shown in Figure 15.36b. Also let the user pause/resume the animation with a button press/release and increase/decrease the car speed by pressing the up and down arrow keys.



**FIGURE 15.36** (a) The program displays a moving car. (b) You can redraw a car with a new base point.

**\*\*15.30** (*Slide show*) Twenty-five slides are stored as image files (**slide0.jpg**, **slide1.jpg**, . . . , **slide24.jpg**) in the **image** directory downloadable along with the source code in the book. The size of each image is  $800 \times 600$ . Write a program that automatically displays the slides repeatedly. Each slide is shown for two seconds. The slides are displayed in order. When the last slide finishes, the first slide is redisplayed, and so on. Click to pause if the animation is currently playing. Click to resume if the animation is currently paused.

**\*\*15.31** (*Geometry: pendulum*) Write a program that animates a pendulum swinging, as shown in Figure 15.37. Press the up arrow key to increase the speed, and the down arrow key to decrease it. Press the *S* key to stop animation of and the *R* key to resume it.

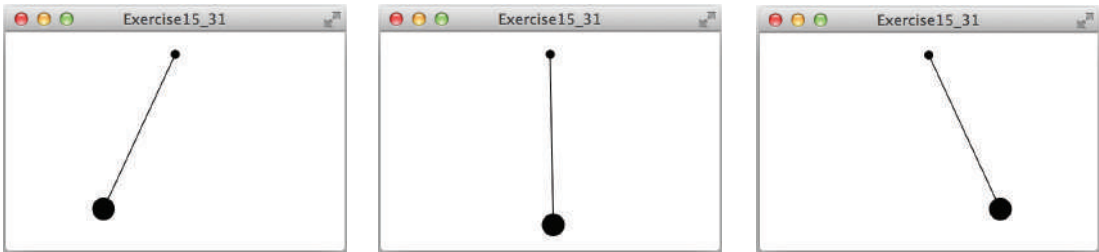


FIGURE 15.37 Exercise 15.31 animates a pendulum swinging.

- \*15.32 (Control a clock) Modify Listing 14.21, ClockPane.java, to add the animation into this class and add two methods `start()` and `stop()` to start and stop the clock, respectively. Write a program that lets the user control the clock with the *Start* and *Stop* buttons, as shown in Figure 15.38a.
- \*\*\*15.33 (Game: bean-machine animation) Write a program that animates the bean machine introduced in Programming Exercise 7.37. The animation terminates after 10 balls are dropped, as shown in Figures 15.38b and c.

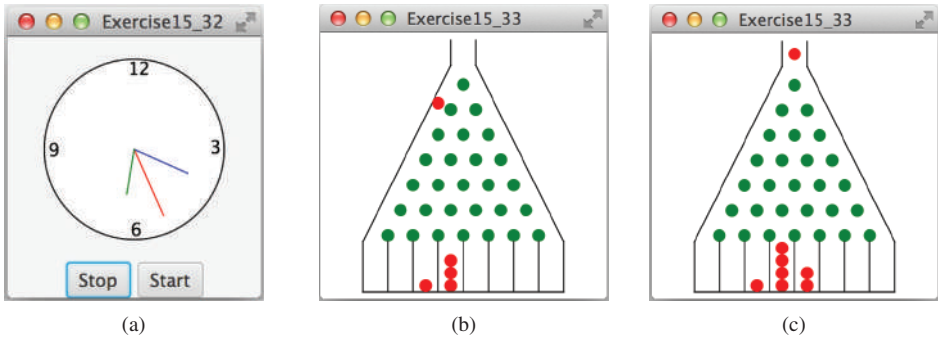


FIGURE 15.38 (a) Exercise 15.32 allows the user to start and stop a clock. (b and c) The balls are dropped into the bean machine.

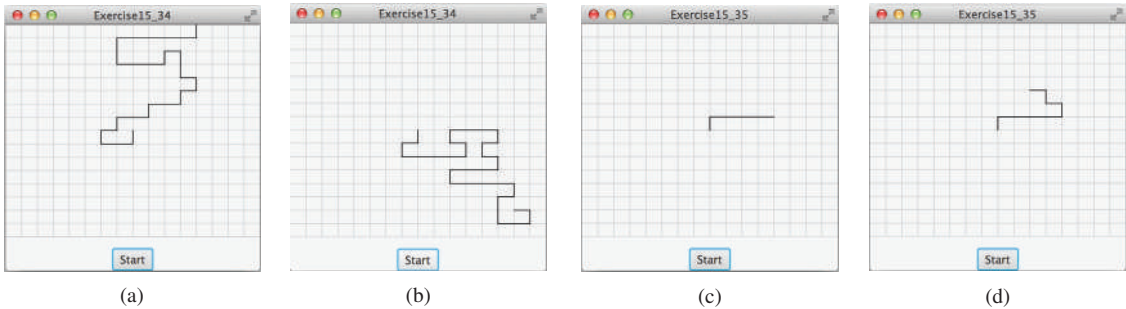


FIGURE 15.39 (a) A path ends at a boundary point. (b) A path ends at dead-end point. (c and d) Animation shows the progress of a path step by step.

- \*\*\*15.34** (*Simulation: self-avoiding random walk*) A self-avoiding walk in a lattice is a path from one point to another that does not visit the same point twice. Self-avoiding walks have applications in physics, chemistry, and mathematics. They can be used to model chain-like entities such as solvents and polymers. Write a program that displays a random path that starts from the center and ends at a point on the boundary, as shown in Figure 15.39a, or ends at a dead-end point (i.e., surrounded by four points that have already been visited), as shown in Figure 15.39b. Assume the size of the lattice is **16** by **16**.
- \*\*\*15.35** (*Animation: self-avoiding random walk*) Revise the preceding exercise to display the walk step by step in an animation, as shown in Figures 15.39c and d.
- \*\*15.36** (*Simulation: self-avoiding random walk*) Write a simulation program to show that the chance of getting dead-end paths increases as the grid size increases. Your program simulates lattices with size from 10 to 80 with increments of 5. For each lattice size, simulate a self-avoiding random walk 10,000 times and display the probability of the dead-end paths, as shown in the following sample output:

```
For a lattice of size 10, the probability of dead-end paths is 10.6%
For a lattice of size 15, the probability of dead-end paths is 14.0%
...
For a lattice of size 80, the probability of dead-end paths is 99.5%
```



