Chapter 15

Event-Driven Programming and Animations
In console applications the program is in charge of the flow of control, but in a GUI application it is the user that dictates what happens next. Execution flow is all dependent on what action the user takes. These actions are manifested as Events, which the application then handles.
Event Handling

Source Object (Button)

Event Object

Target Object (EventHandler)

must implement: `EventHandler<T extends Event>

must register with source: `source.setOnAction(handler)`
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {
        okButton.setOnAction( new OkHandlerClass() );
    }
}

class OkHandlerClass
    implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {...}
}
Inner Handler

Class

Handler
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {
        okButton.setOnAction( new OkHandlerClass() );
    }

    class OkHandlerClass
        implements EventHandler<ActionEvent> {
            @Override
            public void handle(ActionEvent e) {...}
        }
}
Anonymous Inner Handler

Class

( {} )
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {

        okButton.setOnAction(
            new EventHandler<ActionEvent> {
                @Override
                public void handle(ActionEvent e) {...}
            }
        );
    }
}
Lambda Expression

Class

( { } )
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {

        okButton.setOnAction( e -> {...} );
    }
}
This is a sample of the event types.
The widget (source object) the user interacts with on a UI fires an Event object indicating to the application what has happened. In the application a Listener object receives this Event and handles it accordingly.
The OS is the dispatcher of such events, meaning it detects the user’s action and sends the Event to the application.
Your code ties the source object to the listener object by registering the listener with the source object. This registration is a form of delegation. The source object delegates to the listener object the task of handling the event.
### Events by action

<table>
<thead>
<tr>
<th>Source Object</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button, TextField, RadioButton, CheckBox, ComboBox</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Node, Scene</td>
<td>MouseEvent or KeyEvent</td>
</tr>
</tbody>
</table>

13
Event Registration

.setOnAction(...)  
.setOnMouseXXX(...)  
.setOnKeyXXX(...)
A source object raises the event

The user clicks on the Enlarge button, causing an ActionEvent to be raised or fired.
public class Main extends Application {
    // create the circle pane instance
    private final CirclePane circlePane = new CirclePane();

    @Override
    public void start(Stage primaryStage) {
        HBox sizingHB = new HBox(15.0);
        sizingHB.setAlignment(Pos.CENTER);

        Button sizeUpButton = new Button("Increase");
        Button sizeDownButton = new Button("Decrease");
    }
// register the two sizing event handlers
sizeUpButton.setOnAction(new SizeUpHandler());
sizeDownButton.setOnAction(new SizeDownHandler());

sizingHB.getChildren().addAll(sizeUpButton, sizeDownButton);

BorderPane bp = new BorderPane(circlePane);
bp.setPadding(new Insets(10, 10, 10, 10));
bp.setBottom(sizingHB);
primaryStage.setTitle("Circle Size Adjustment");
primaryStage.setScene(new Scene(bp, 200, 200));
primaryStage.show();
} // end start()
/* Note how these two event handler classes
 * are nested inside the Main class. */

private class SizeUpHandler // inner class
    implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        circlePane.sizeUpByOne();
    }
}

private class SizeDownHandler
    implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        circlePane.sizeDownByOne();
    }
}
} // end Main
class CirclePane extends StackPane {
    private final Circle circle =
        new Circle(50, Color.YELLOW);

    CirclePane() { getChildren().add(circle); }

    public void sizeUpByOne() {
        circle.setRadius(circle.getRadius() + 1);
    }

    public void sizeDownByOne() {
        double r = circle.getRadius();
        circle.setRadius(r > 5? r - 1 : r);
    }
}
A class may be defined inside another

Inner classes are intimately related to the classes they reside within.

Think of it this way: The inner class exists only for the purpose of serving the outer class. It is not useful to any other class on its own.

Inner classes are good candidates for listeners for this exact reason.
It is compiled into
OuterClass$InnerClass.class
The inner class can access directly the ivars and methods of its outer class. This includes private and public alike.

The outer class however must abide by the encapsulation rules. In other words, it can only access the public interface of the inner class.
Can have any access modifier
(public, protected, default, private)

This of course will dictate how a client of the outer class will be able to access the inner class.
Can be defined as static
Create inner class object

```java
OuterClass.InnerClass innerObject

= outerObject.new InnerClass();
```

This is permitted only if inner class’s access modifier allows this. For example, it must not be private nor static.
Create Static inner class object

OuterClass.InnerClass innerObject

= new OuterClass.InnerClass();

This is permitted only if inner class’s access modifier allows this. For example, it must not be private nor static.
Anonymous inner classes can replace the 1:1 delegation model, since it saves you the trouble of first defining the named class and then registering an instance of it with the source object.

On the other hand, with a 1:n relationship, this will lead to a lot more code and duplicated effort. In this case you are probably better off sticking with the named class and registering it with all the widgets.
sizeUpButton.setOnAction(new EventHandler<ActionEvent>() {
  @Override
  public void handle(ActionEvent event) {
    circlePane.sizeUpByOne();
  }
});

Anonymous inner classes can replace the 1:1 delegation model, since it saves you the trouble of first defining the named class and then registering an instance of it with the source object.

On the other hand, with a 1:n relationship, this will lead to a lot more code and duplicated effort. In this case you are probably better off sticking with the named class and registering it with all the widgets.
An anonymous inner class must extend a superclass ...
... or implement an interface
It must implement all methods in superclass or interface
Always uses super's no-arg constructor
It is compiled into 
\texttt{OuterClass$^n$.class}

Where $n$ starts out with 1 and increases by 1 for each anonymous class.
Anonymous inner classes can replace the 1:1 delegation model, since it saves you the trouble of first defining the named class and then registering an instance of it with the source object.

On the other hand, with a 1:n relationship, this will lead to a lot more code and duplicated effort. In this case you are probably better off sticking with the named class and registering it with all the widgets.
(type p1, type p2,...) -> statement
(type p1, ...) -> { statements; }
(p1, ...) -> { statements; }

type inferred
p -> { statements; }  

single parameter
Mouse Events

.setOnMousePressed()
.setOnMouseReleased()
.setOnMouseClicked()
.setOnMouseEntered()
.setOnMouseExited()
.setOnMouseMoved()
.setOnMouseDragged()
Text text = new Text(20, 20,
        "Programming is fun");

text.setOnMouseDragged(e -> {
    text.setX( e.getX() );
    text.setY( e.getY() );
});
Key Events

.setOnKeyPressed()
.setOnKeyReleased()
.setOnKeyTyped()
private final CirclePane circlePane =
   new CirclePane();

/* control must have focus for KeyEvent */
circlePane.requestFocus();

circlePane.setOnKeyPressed( e -> {
   if (e.getCode() == KeyCode.I) {
      circlePane.sizeUpBy(RADIUS_INCREMENT);
   } else if (e.getCode() == KeyCode.D) {
      circlePane.sizeDownBy(RADIUS_INCREMENT);
   }
});
Animation

- PathTransition
- FadeTransition
- Timeline
Animation

-autoReverse: BooleanProperty
-cycleCount: IntegerProperty
-rate: DoubleProperty
-status: ReadOnlyObjectProperty<Animation.Status>

+pause(): void
+play(): void
+stop(): void
PathTransition

- duration: ObjectProperty<Duration>
- node: ObjectProperty<Node>
- orientation: ObjectProperty
  <PathTransition.OrientationType>
- path: ObjectType<Shape>

+ PathTransition()
+ PathTransition(duration: Duration, path: Shape)
+ PathTransition(duration: Duration, path: Shape, node: Node)
Circle circlePath = new Circle(100, 100, 50);
Circle orbitingDot = new Circle(100, 50, 5);

// Create the path transition
PathTransition pt = new PathTransition(
    Duration.millis(4000),
    circlePath,
    orbitingDot);

pt.setInterpolator(Interpolator.LINEAR);
pt.setOrientation(
    PathTransition.OrientationType.NONE);
pt.setCycleCount(Timeline.INDEFINITE);
pt.setAutoReverse(false);
pt.play();
**FadeTransition**

- `duration`: `ObjectProperty<Duration>`
- `node`: `ObjectProperty<Node>`
- `fromValue`: `DoubleProperty`
- `toValue`: `DoubleProperty`
- `byValue`: `DoubleProperty`

+ `FadeTransition()`
+ `FadeTransition(duration: Duration)`
+ `PathTransition(duration: Duration, node: Node)`
Timeline

+Timeline()
+Timeline(double targetFramerate)
+Timeline(double targetFramerate, KeyFrame...keyFrames)
+Timeline(KeyFrame...keyFrames)
Text text = new Text(20, 50, "Programming is fun");

Timeline animation = new Timeline(
    new KeyFrame(Duration.millis(500),
        e -> {
            if (text.getText().length() != 0) {
                text.setText('');
            } else {
                text.setText("Programming is fun");
            }
        }
    )
);