

# Chapter 15 Event-Driven Programming and Animations

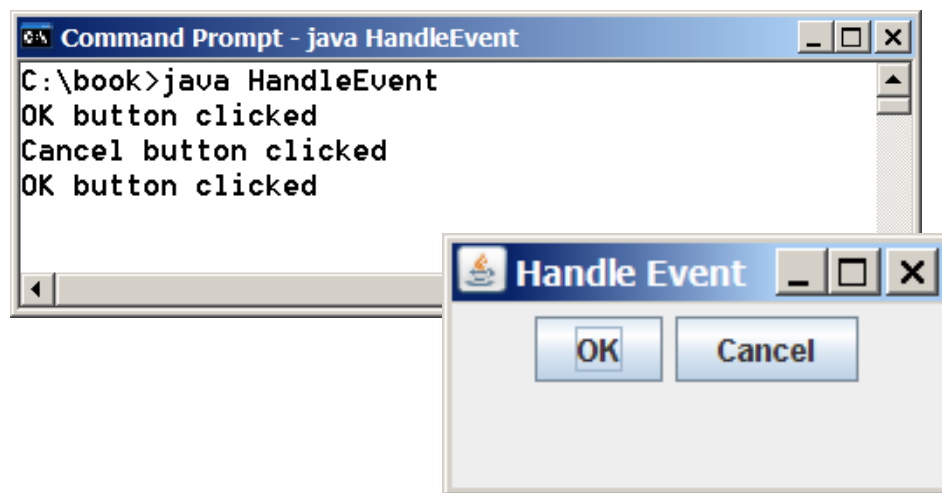


# Procedural vs. Event-Driven Programming

- *Procedural programming* is executed in procedural order.
- In event-driven programming, code is executed upon activation of events.

# Taste of Event-Driven Programming

The example displays a button in the frame. A message is displayed on the console when a button is clicked.



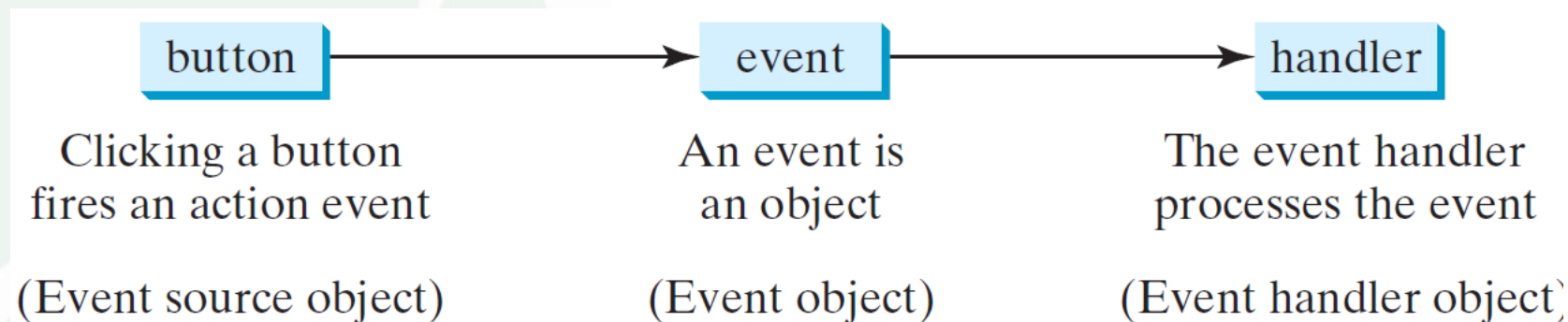
HandleEvent

Run

# Handling GUI Events

Source object (e.g., button)

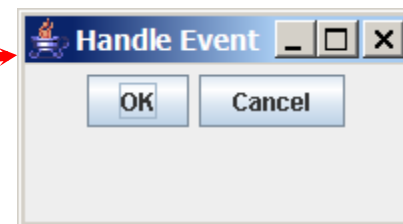
Listener object contains a method for processing the event.



# Trace Execution

```
public class HandleEvent extends Application {  
    public void start(Stage primaryStage) {  
        ...  
        OKHandlerClass handler1 = new OKHandlerClass();  
        btOK.setOnAction(handler1);  
        CancelHandlerClass handler2 = new CancelHandlerClass();  
        btCancel.setOnAction(handler2);  
        ...  
        primaryStage.show(); // Display the stage  
    }  
}
```

1. Start from the main method to create a window and display it

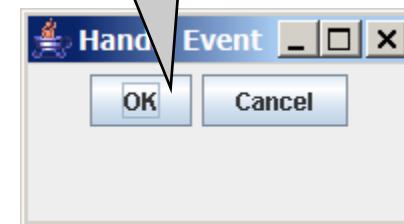


```
class OKHandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

# Trace Execution

```
public class HandleEvent extends Application {  
    public void start(Stage primaryStage) {  
        ...  
        OKHandlerClass handler1 = new OKHandlerClass();  
        btOK.setOnAction(handler1);  
        CancelHandlerClass handler2 = new CancelHandlerClass();  
        btCancel.setOnAction(handler2);  
        ...  
        primaryStage.show(); // Display the stage  
    }  
}
```

2. Click OK



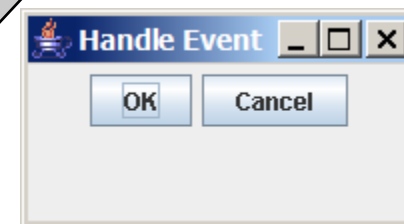
```
class OKHandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

# Trace Execution

```
public class HandleEvent extends Application {  
    public void start(Stage primaryStage) {  
        ...  
        OKHandlerClass handler1 = new OKHandlerClass();  
        btOK.setOnAction(handler1);  
        CancelHandlerClass handler2 = new CancelHandlerClass();  
        btCancel.setOnAction(handler2);  
        ...  
        primaryStage.show(); // Display the stage  
    }  
}
```

```
class OKHandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

3. The JVM invokes the listener's handle method

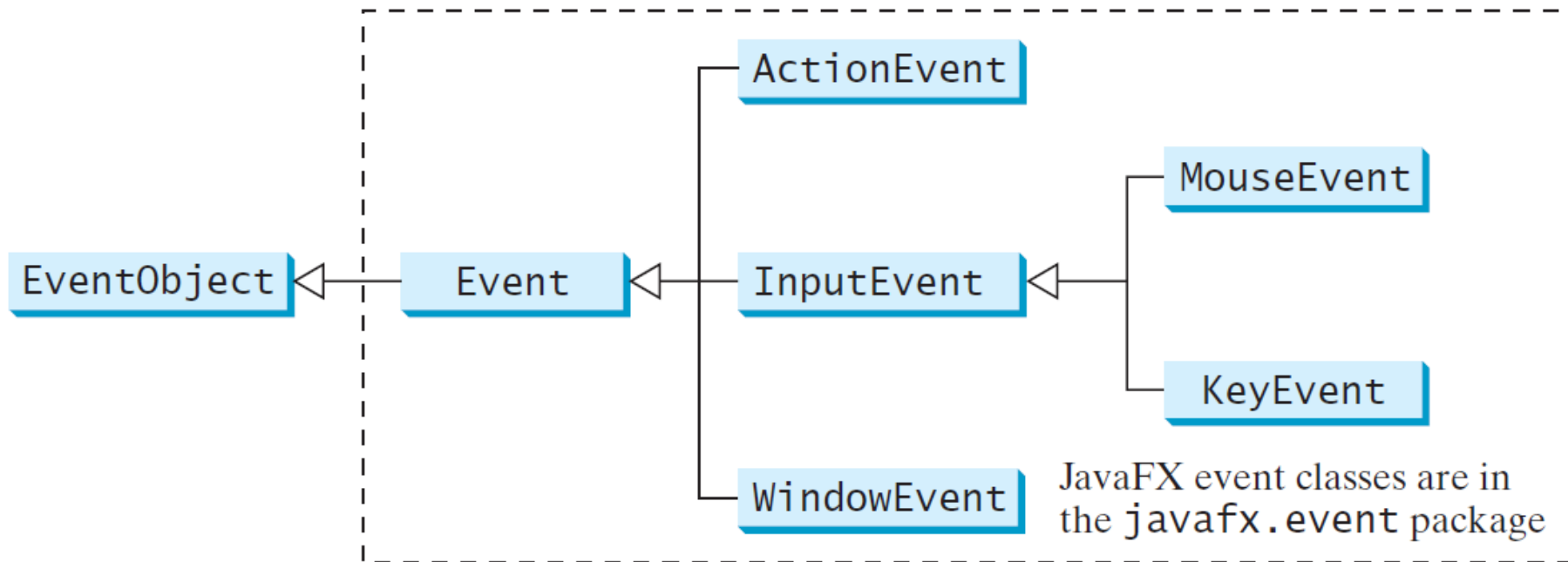


# Events

- ❑ An *event* can be defined as a type of signal to the program that something has happened.
- ❑ The event is generated by external user actions such as mouse movements, mouse clicks, or keystrokes.



# Event Classes



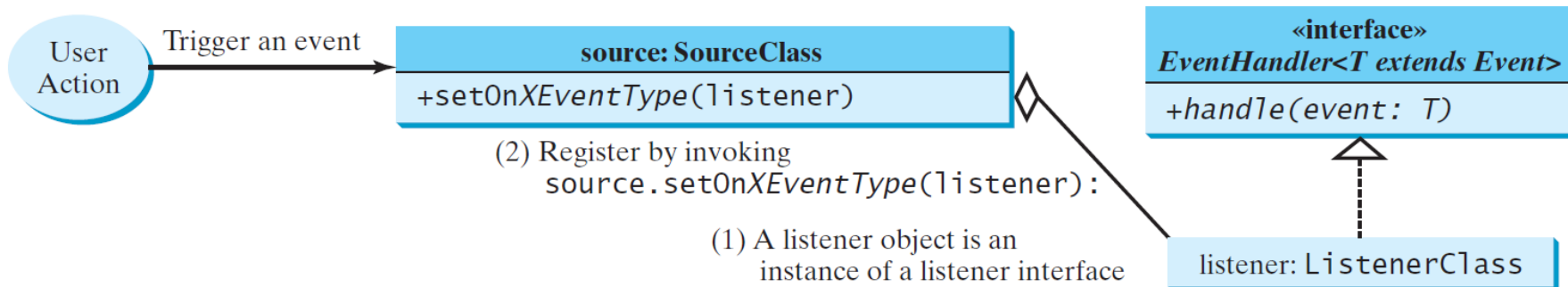
# Event Information

An event object contains whatever properties are pertinent to the event. You can identify the source object of the event using the `getSource()` instance method in the `EventObject` class. The subclasses of `EventObject` deal with special types of events, such as button actions, window events, mouse movements, and keystrokes. Table 15.1 lists external user actions, source objects, and event types generated.

# Selected User Actions and Handlers

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

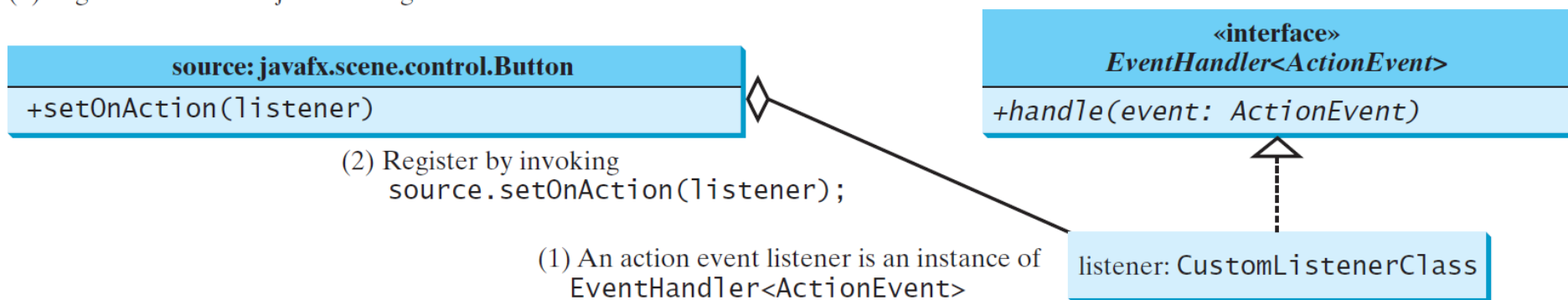
# The Delegation Model



(2) Register by invoking `source.setOnXEventType(listener)`:

(1) A listener object is an instance of a listener interface

(a) A generic source object with a generic event T



(2) Register by invoking `source.setOnAction(listener);`

(1) An action event listener is an instance of `EventHandler<ActionEvent>`

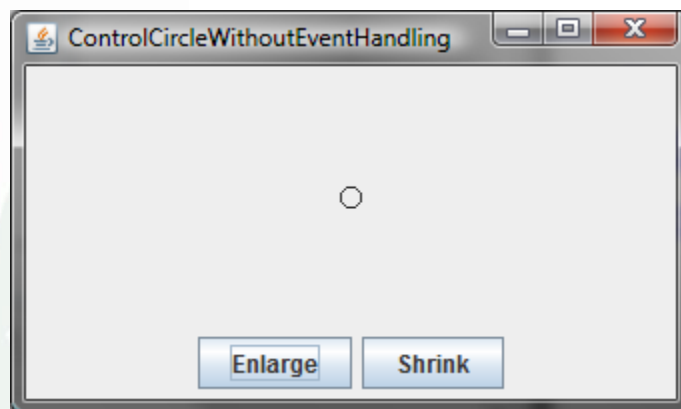
(b) A Button source object with an ActionEvent

# The Delegation Model: Example

```
Button btOK = new Button("OK");  
OKHandlerClass handler = new OKHandlerClass();  
btOK.setAction(handler);
```

# Example: First Version for ControlCircle (no listeners)

Now let us consider to write a program that uses two buttons to control the size of a circle.

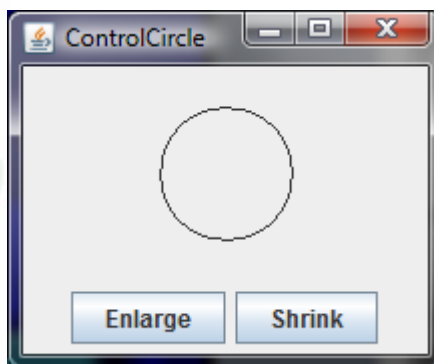
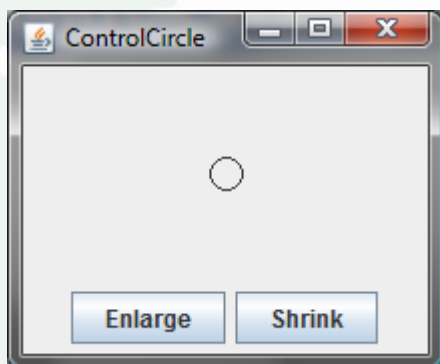


ControlCircleWithoutEventHandling

Run

# Example: Second Version for ControlCircle (with listener for Enlarge)

Now let us consider to write a program that uses two buttons to control the size of a circle.



ControlCircle

Run

# Inner Class Listeners

A listener class is designed specifically to create a listener object for a GUI component (e.g., a button). It will not be shared by other applications. So, it is appropriate to define the listener class inside the frame class as an inner class.



# Inner Classes

Inner class: A class is a member of another class.

Advantages: In some applications, you can use an inner class to make programs simple.

**An inner class can reference the data and methods defined in the outer class in which it nests, so you do not need to pass the reference of the outer class to the constructor of the inner class.**

ShowInnerClass

# Inner Classes, cont.

```
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)

# Inner Classes (cont.)

Inner classes can make programs simple and concise.

An inner class supports the work of its containing outer class and is compiled into a class named

*OuterClassName\$InnerClassName.class*.

For example, the inner class `InnerClass` in `OuterClass` is compiled into

*OuterClass\$InnerClass.class* .

# Inner Classes (cont.)

- ❑ An inner class can be declared public, protected, or private subject to the same visibility rules applied to a member of the class.
- ❑ An inner class can be declared 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

# Anonymous Inner Classes

- ❑ 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, these two classes are compiled into `Test$1.class` and `Test$2.class`.

# Anonymous Inner Classes (cont.)

Inner class listeners can be shortened using anonymous inner classes. An *anonymous inner class* is an inner class without a name. It combines declaring an inner class and creating an instance of the class in one step. An anonymous inner class is declared as follows:

```
new SuperClassName/InterfaceName() {  
    // Implement or override methods in superclass or interface  
    // Other methods if necessary  
}
```

# Anonymous Inner Classes (cont.)

```

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

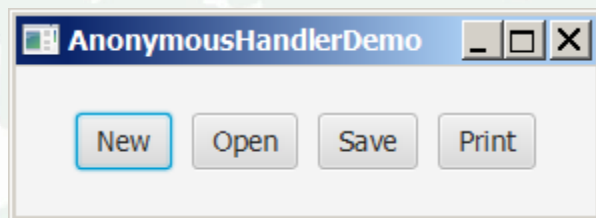
```

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



AnonymousHandlerDemo

Run

# Simplifying Event Handling Using Lambda Expressions

*Lambda expression* is a new feature in Java 8. Lambda expressions can be viewed as an anonymous method with a concise syntax. For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines.

```
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



# Basic Syntax for a Lambda Expression

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.

# Single Abstract Method Interface (SAM)

The statements in the lambda expression is all for that method. If it contains multiple methods, the compiler will not be able to compile the lambda expression. So, for the compiler to understand lambda expressions, the interface must contain exactly one abstract method. Such an interface is known as a *functional interface*, or a *Single Abstract Method (SAM) interface*.

AnonymousHandlerDemo

Run

# The MouseEvent Class

## javafx.scene.input.MouseEvent

```

+getButton(): MouseButton
+getClickCount(): int
+getX(): double
+getY(): double
+getSceneX(): double
+getSceneY(): double
+getScreenX(): double
+getScreenY(): double
+isAltDown(): boolean
+isControlDown(): boolean
+isMetaDown(): boolean
+isShiftDown(): boolean
  
```

Indicates which mouse button has been clicked.

Returns the number of mouse clicks associated with this event.

Returns the *x*-coordinate of the mouse point in the event source node.

Returns the *y*-coordinate of the mouse point in the event source node.

Returns the *x*-coordinate of the mouse point in the scene.

Returns the *y*-coordinate of the mouse point in the scene.

Returns the *x*-coordinate of the mouse point in the screen.

Returns the *y*-coordinate of the mouse point in the screen.

Returns true if the `Alt` key is pressed on this event.

Returns true if the `Control` key is pressed on this event.

Returns true if the mouse `Meta` button is pressed on this event.

Returns true if the `Shift` key is pressed on this event.

MouseEventDemo

Run

# The KeyEvent Class

## `javafx.scene.input.KeyEvent`

```
+getCharacter(): String  
+getCode(): KeyCode  
+getText(): String  
+isAltDown(): boolean  
+isControlDown(): boolean  
+isMetaDown(): boolean  
+isShiftDown(): boolean
```

Returns the character associated with the key in this event.  
Returns the key code associated with the key in this event.  
Returns a string describing the key code.  
Returns true if the `Alt` key is pressed on this event.  
Returns true if the `Control` key is pressed on this event.  
Returns true if the mouse `Meta` button is pressed on this event.  
Returns true if the `Shift` key is pressed on this event.

KeyEventDemo

Run

# The KeyCode Constants

| <i>Constant</i>  | <i>Description</i>  | <i>Constant</i>   | <i>Description</i>               |
|------------------|---------------------|-------------------|----------------------------------|
| <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      |

# Example: Control Circle with Mouse and Key



ControlCircleWithMouseAndKey

Run

# 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. Once the value is changed in the property, a listener is notified. The listener class should implement the **InvalidationListener** interface, which uses the **invalidated(Observable o)** method to handle the property value change. Every binding property is an instance of **Observable**.

ObservablePropertyDemo

Run

DisplayResizableClock

Run

# Animation

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

## *javafx.animation.Animation*

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

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

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Defines whether the animation reverses direction on alternating cycles.  
 Defines the number of cycles in this animation.  
 Defines the speed and direction for this animation.  
 Read-only property to indicate the status of the animation.

Pauses the animation.  
 Plays the animation from the current position.  
 Stops the animation and resets the animation.



# PathTransition

## javafx.animation.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)

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The duration of this transition.

The target node of this transition.

The orientation of the node along the path.

The shape whose outline is used as a path to animate the node move.

Creates an empty PathTransition.

Creates a PathTransition with the specified duration and path.

Creates a PathTransition with the specified duration, path, and node.

PathTransitionDemo

Run

FlagRisingAnimation

Run

# FadeTransition

The **FadeTransition** class animates the change of the opacity in a node over a given time.

## javafx.animation.FadeTransition

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

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

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The duration of this transition.

The target node of this transition.

The start opacity for this animation.

The stop opacity for this animation.

The incremental value on the opacity for this animation.

Creates an empty `FadeTransition`.

Creates a `FadeTransition` with the specified duration.

Creates a `FadeTransition` with the specified duration and node.

FadeTransitionDemo

Run

# Timeline

**PathTransition** and **FadeTransition** define specialized animations. The **Timeline** class can be used to program any animation using one or more **KeyFrames**. Each **KeyFrame** is executed sequentially at a specified time interval. **Timeline** inherits from **Animation**.

A green rectangular button with the text "TimelineDemo" in white.

TimelineDemo

A blue rectangular button with the text "Run" in white.

Run

# Case Study: Bouncing Ball

