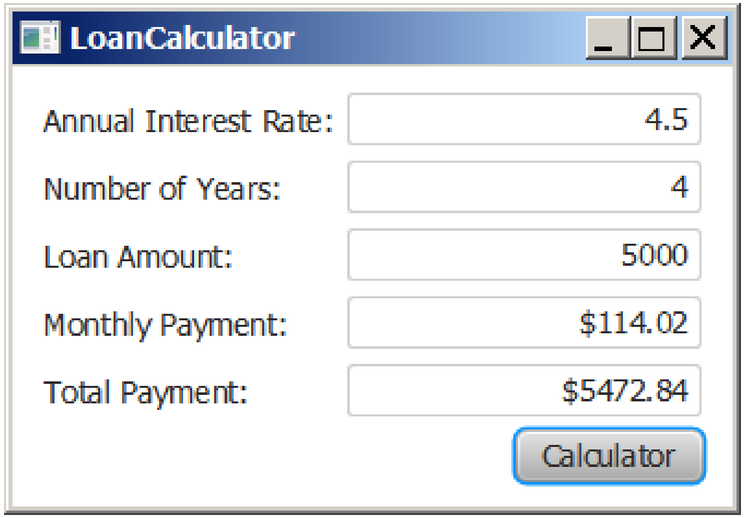
Chapter 15 Event-Driven Programming and Animations

```
// Process events btCalculate.setOnAction(e -> calculateLoanPayment());
```

Motivations

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



Annual Interest Rate:	4.5
Number of Years:	4
Loan Amount:	5000
Monthly Payment:	$114.02
Total Payment:	$5472.84

LoanCalculator

Run

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).

Procedural vs. Event-Driven Programming

- *Procedural programming* is executed in procedural order.

- In event-driven programming, code is executed upon activation of events.

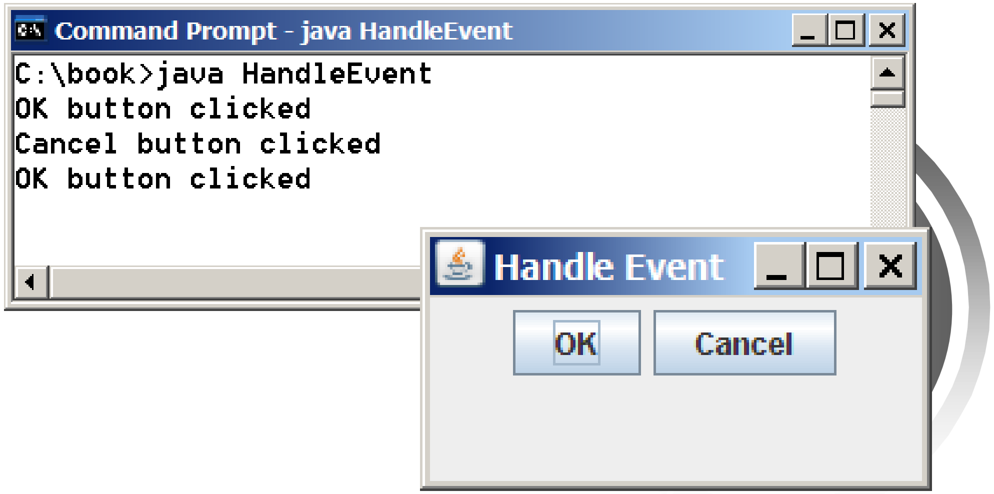
Taste of Event-Driven Programming

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

```
// When btOK  is pushed, OKHandlerClass is called.
OKHandlerClass handler1 = new OKHandlerClass();
btOK.setOnAction(handler1);
```

HandleEvent

Run

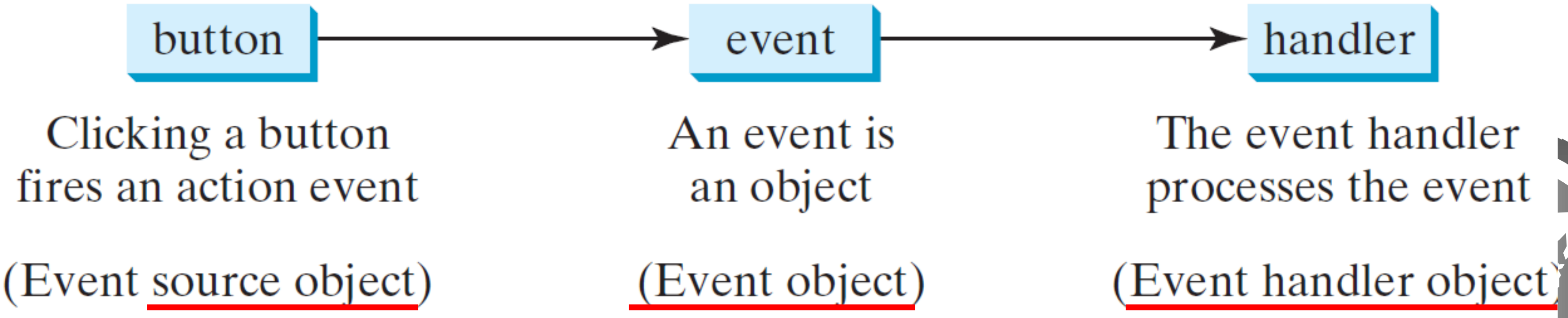


```
C:\book>java HandleEvent
OK button clicked
Cancel button clicked
OK button clicked
```

Handling GUI Events

Source object (e.g., button)

Listener object contains a method for processing the event.

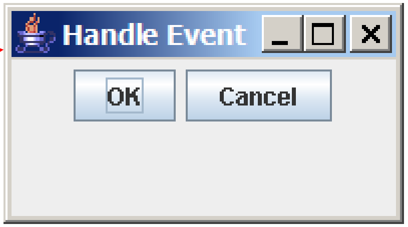


animation

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");
  }
}
```

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

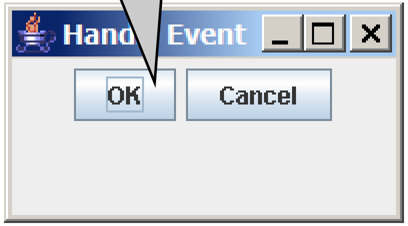


animation

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");
  }
}
```

2. Click OK

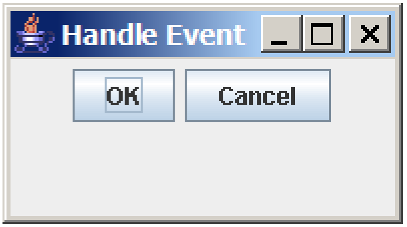


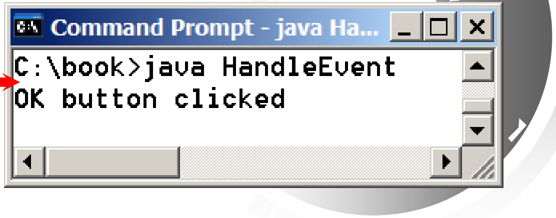
animation

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. Click OK. The JVM invokes the listener's handle method





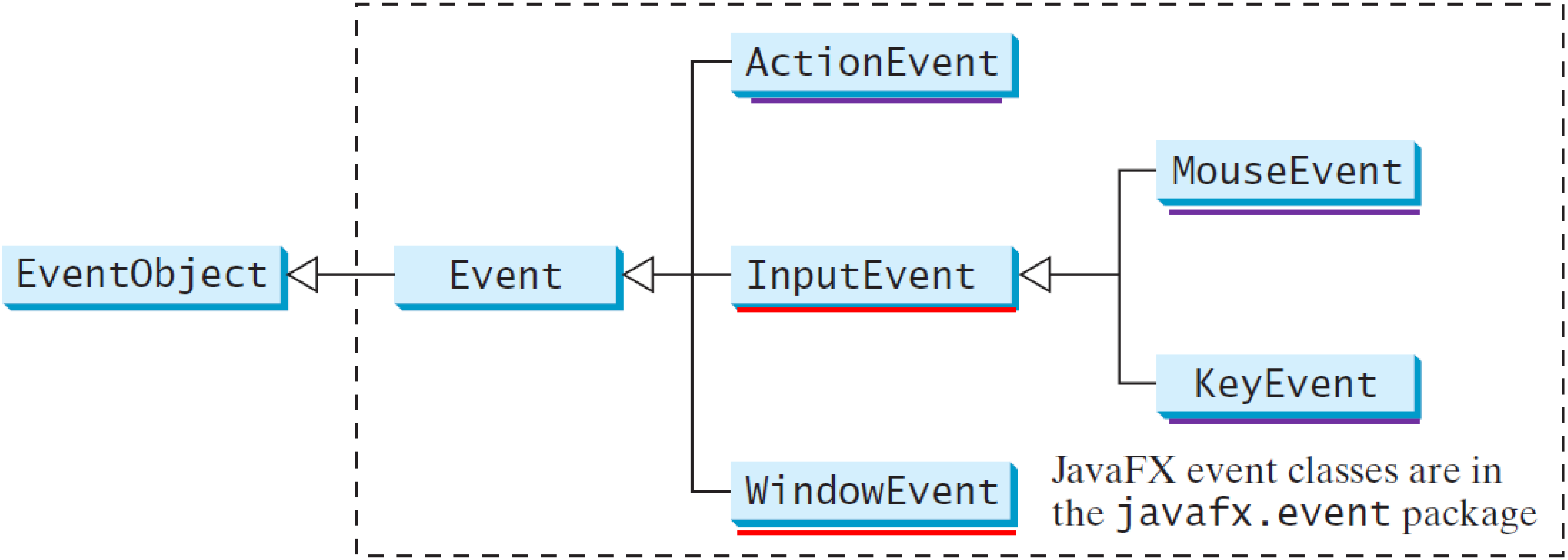
```
C:\book>java HandleEvent
OK button clicked
```

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

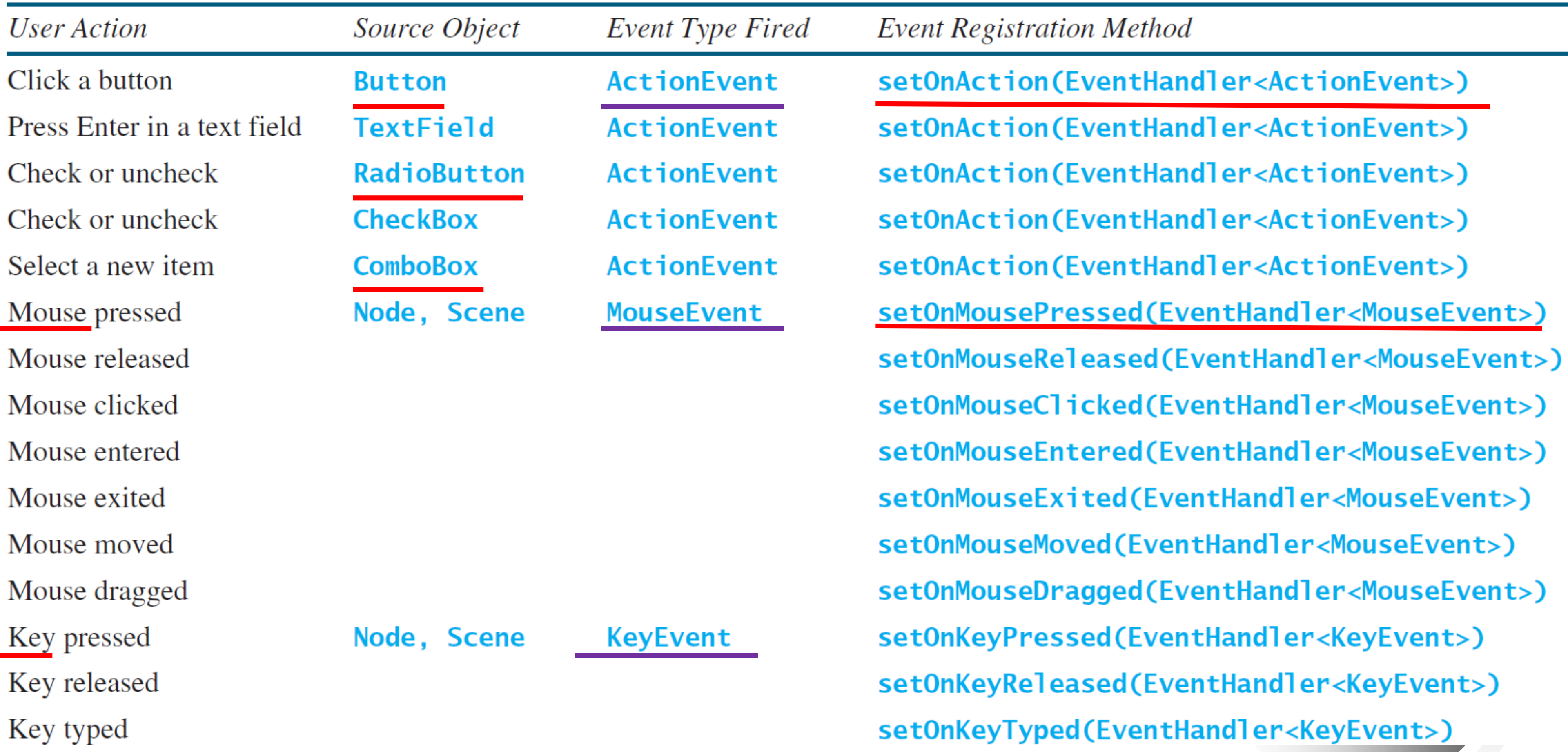


JavaFX event classes are in the `javafx.event` package

Event Information

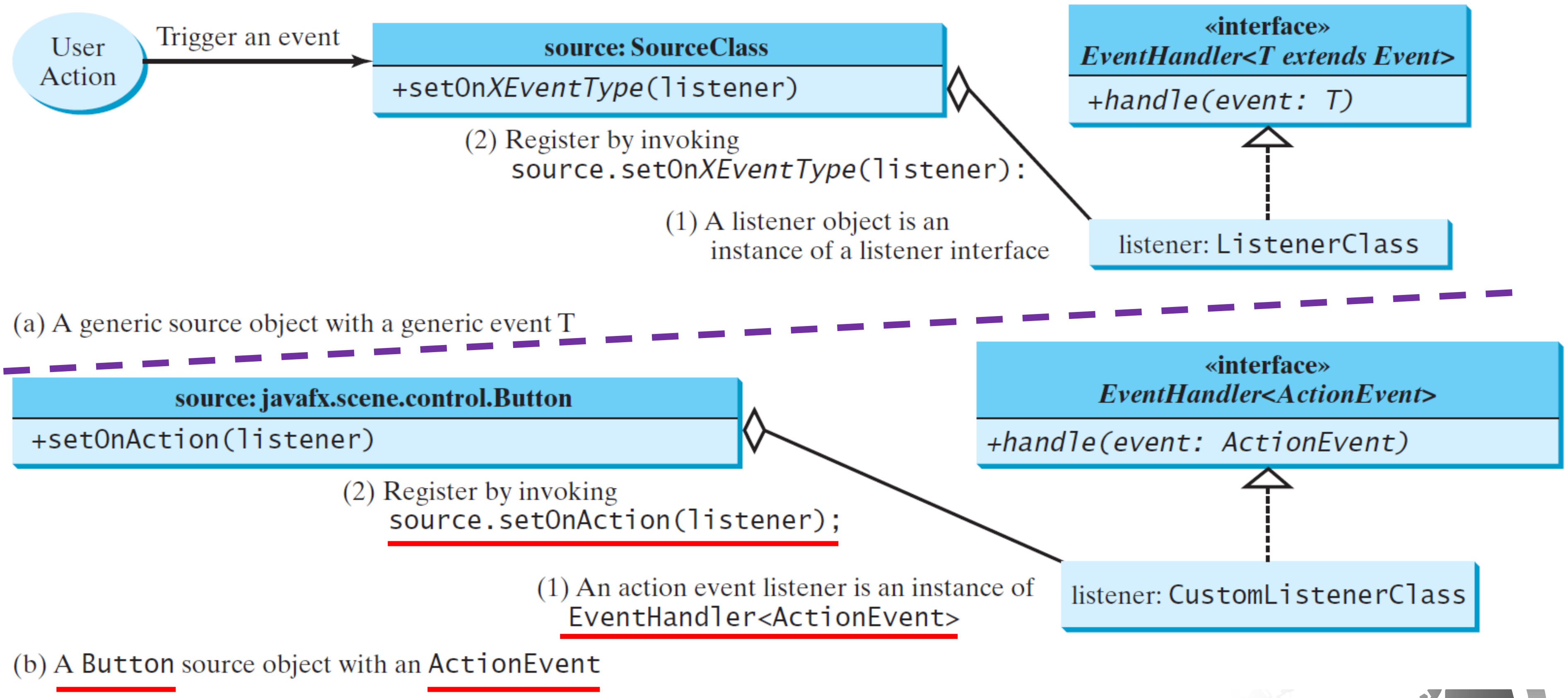
An event object contains whatever properties are pertinent to the event. You can identify the source object of the event using the **getSource()** instance method in the EventObject class. The subclasses of EventObject deal with special types of events, such as button actions, window events, component events, mouse movements, and keystrokes. Table 16.1 lists external user actions, source objects, and event types generated.

Selected User Actions and Handlers



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

The Delegation Model



User Action

Trigger an event

source: SourceClass

+setOnXEventType(listener)

«interface»
EventHandler<T extends Event>

+handle(event: T)

(2) Register by invoking
source.setOnXEventType(listener);

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

listener: ListenerClass

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

source: javafx.scene.control.Button

+setOnAction(listener)

«interface»
EventHandler<ActionEvent>

+handle(event: ActionEvent)

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

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

listener: CustomListenerClass

(b) A Button source object with an ActionEvent

```
CustomListenerClass listener = new CustomListenerClass ();
btOK.setOnAction(listener);
```

The Delegation Model: Example

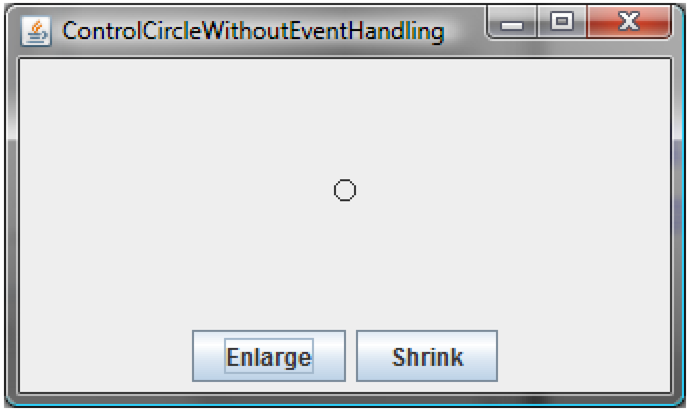
(1)

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

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

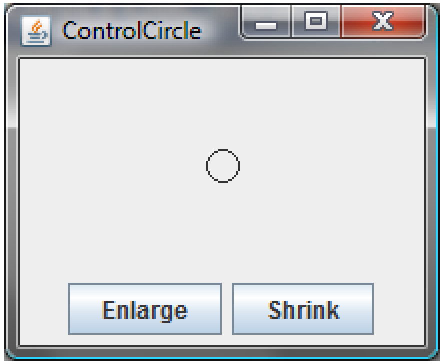


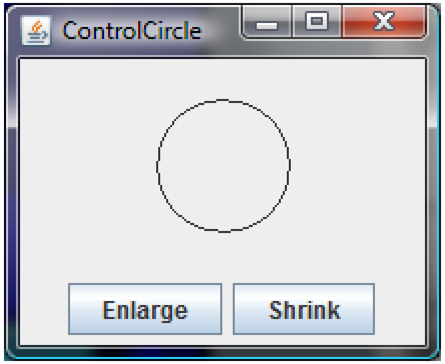
ControlCircleWithoutEventHandling

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.





```
Button btEnlarge = new Button("Enlarge");
btEnlarge.setOnAction(new EnlargeHandler());

class EnlargeHandler implements
EventHandler<ActionEvent> {
@Override // Override the handle method
public void handle(ActionEvent e) {
circlePane.enlarge();
} }
```

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.

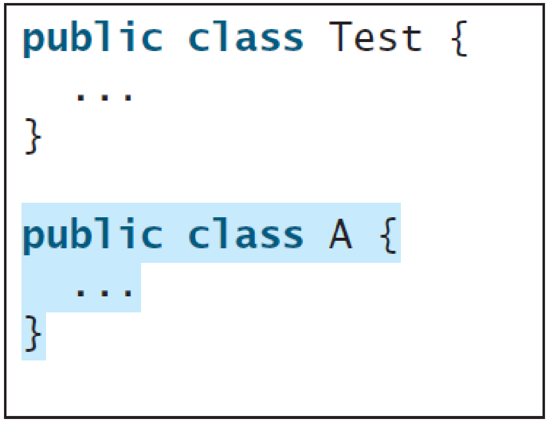
Advantages: In some applications, you can use an inner class to make programs simple.

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.

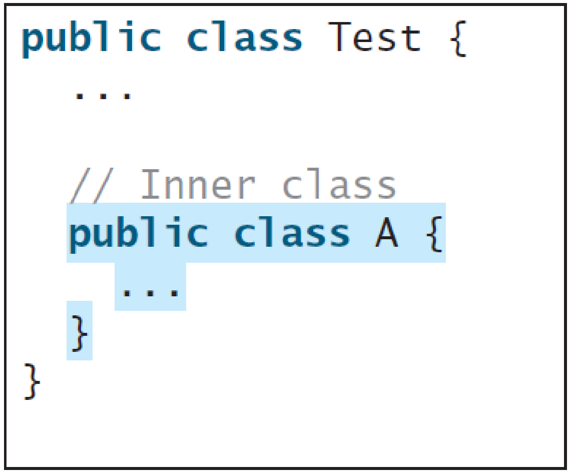
2 classes



```java
public class Test {
  ...
}

public class A {
  ...
}
```

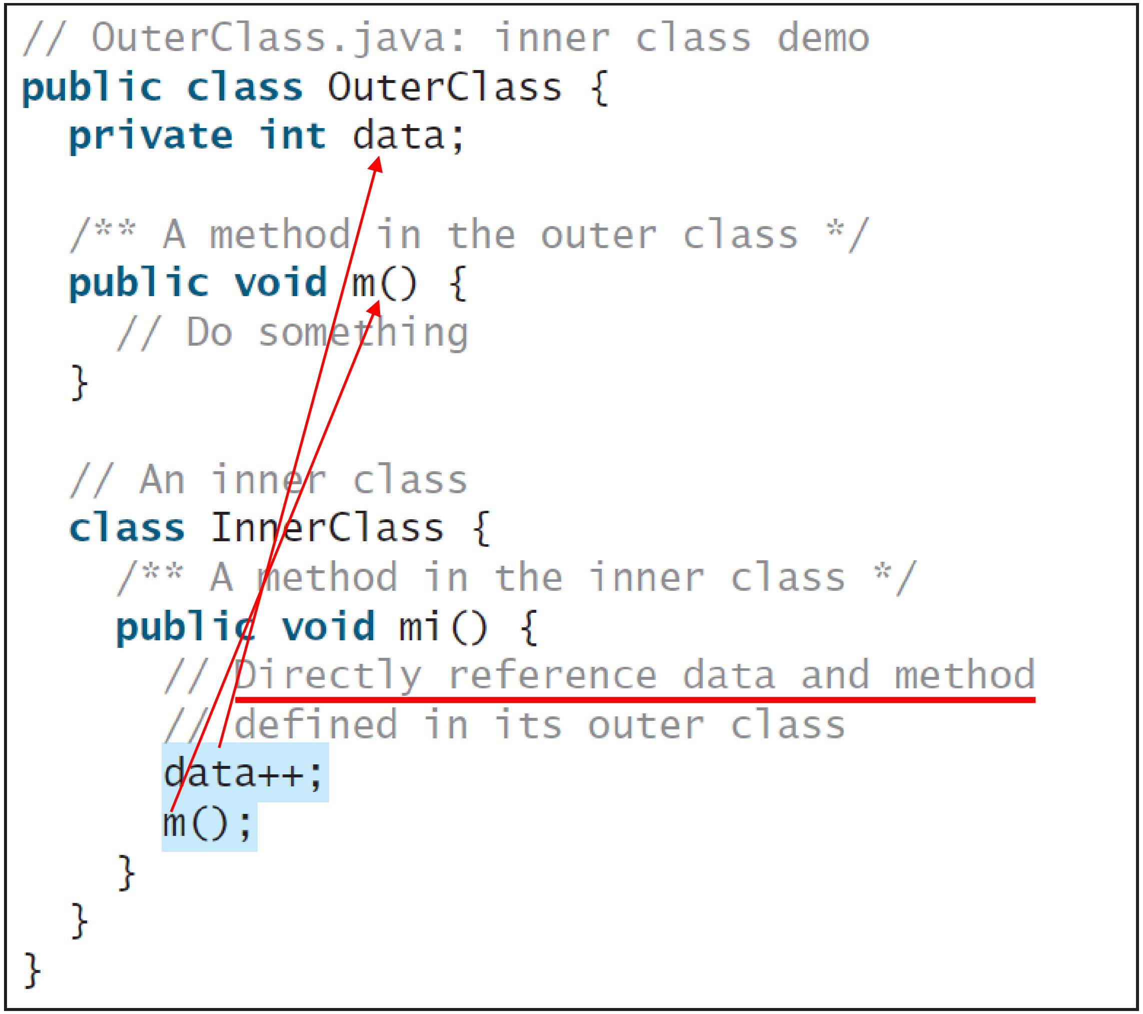
(a)

1 class with 1 inner class



```java
public class Test {
  ...

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

(b)



```java
// 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**.

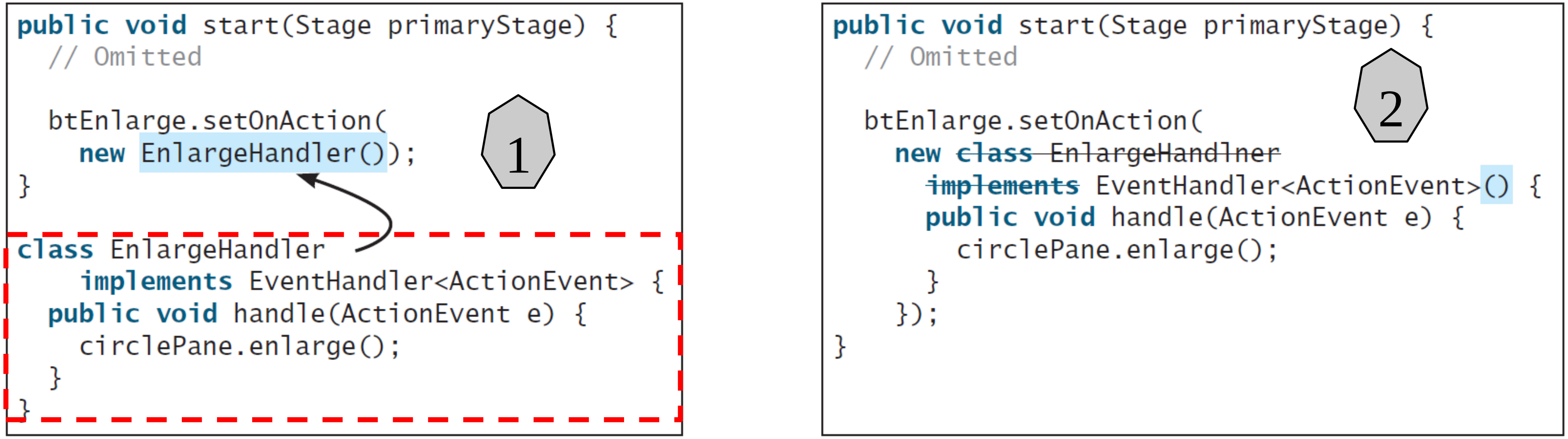
Anonymous Inner Classes (cont.)

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:

```
pane.getChildren().add(new ImageView(new Image("image/us.gif")));
```

```
...(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();
  }
}
```

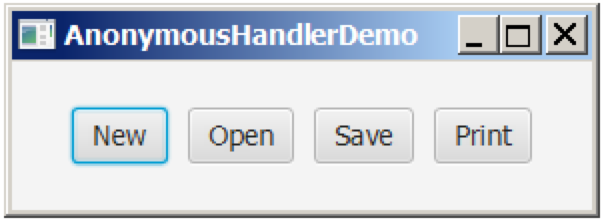
(a) Inner class EnlargeListener

```
public void start(Stage primaryStage) {
  // Omitted

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

(b) Anonymous inner class

```
btNew.setOnAction(new EventHandler<ActionEvent>() {
@Override // Override the handle method
   public void handle(ActionEvent e) {
   System.out.println("Process New");
   }
}
);
```

Anonymous Inner Classes



AnonymousHandlerDemo

Run

Simplifying Event Handing 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

```
btUp.setOnAction((ActionEvent e) -> {
    text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
});
```

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; }
```

or

```
type1 param1-> { 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.

LambdaHandlerDemo

Run

Problem: Loan Calculator

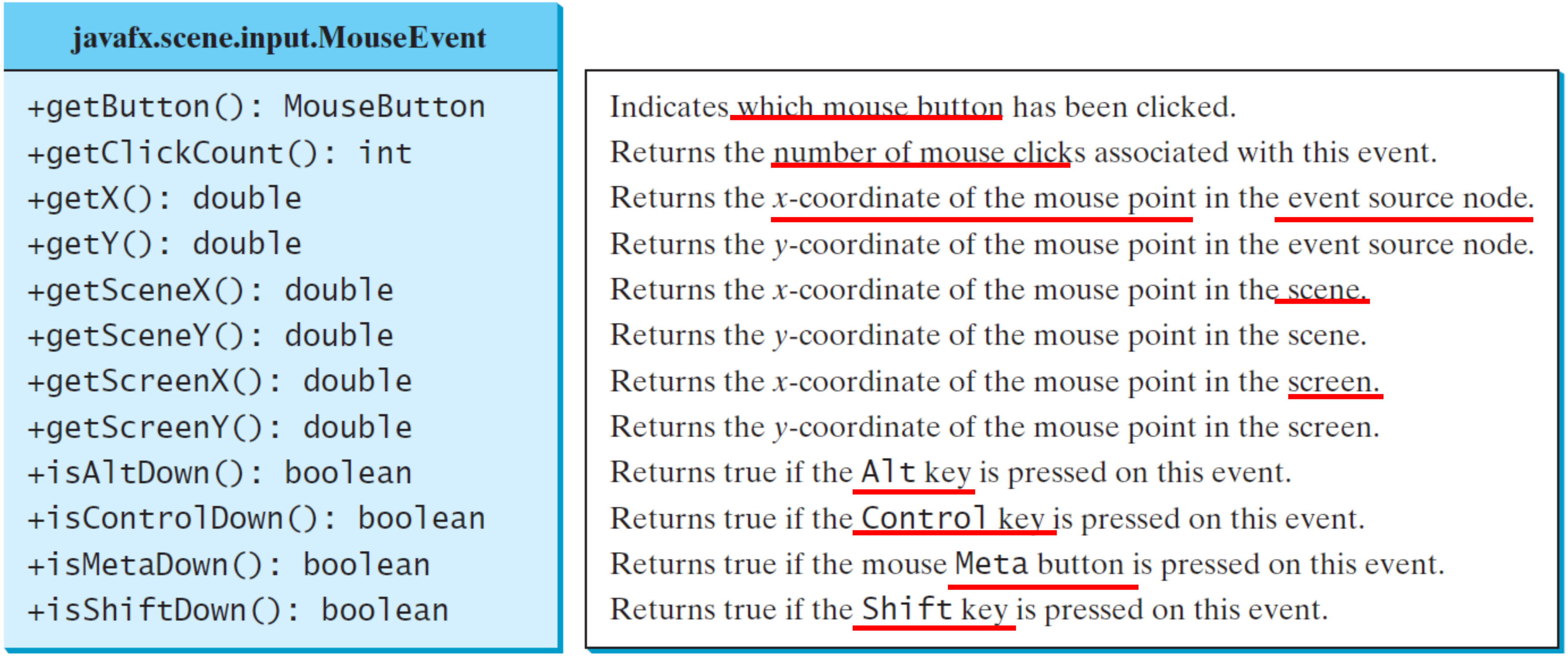
```
private Button btCalculate = new Button("Calculate");
****
// Process events
    btCalculate.setOnAction(e -> calculateLoanPayment());
****
  private void calculateLoanPayment() {
**** /* write your code here */
}
```

LoanCalculator

Run

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

MouseEvent



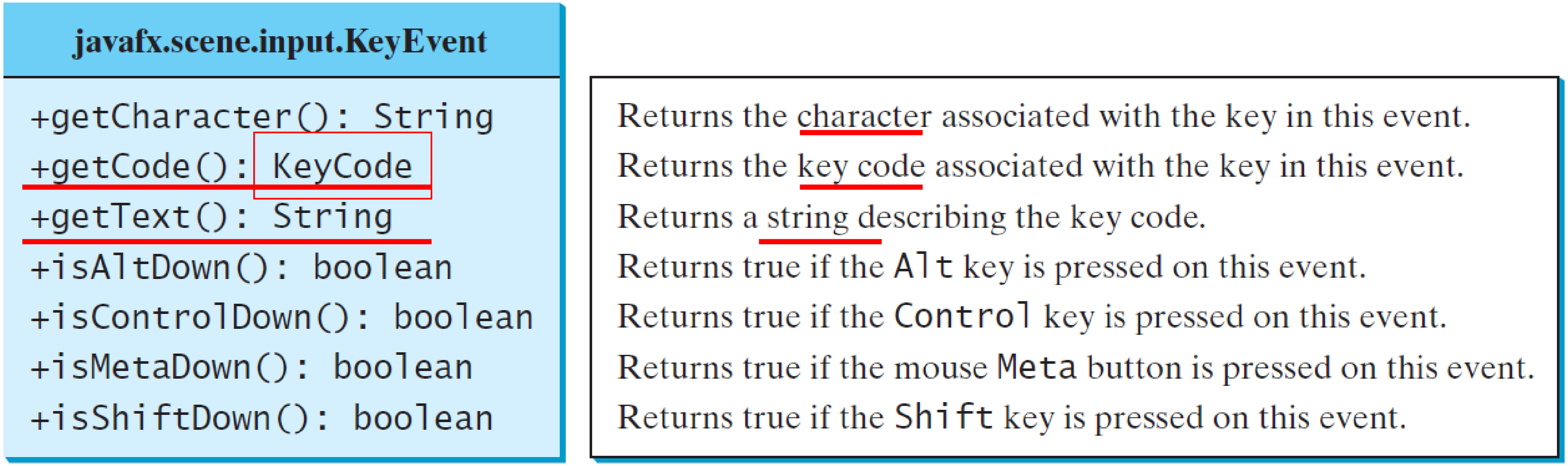
javafx.scene.input.MouseEvent	
`+getButton(): MouseButton`	Indicates which mouse button has been clicked.
`+getClickCount(): int`	Returns the number of mouse clicks associated with this event.
`+getX(): double`	Returns the *x*-coordinate of the mouse point in the event source node.
`+getY(): double`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

MouseEventDemo

Run

The `KeyEvent` Class

```
text.setOnKeyPressed(e -> {
  switch (e.getCode()) {
            case DOWN: text.setY(text.getY() + 10); break;
            case UP: text.setY(text.getY() - 10); break;
            case LEFT: text.setX(text.getX() - 10); break;
            case RIGHT: text.setX(text.getX() + 10); break;
            default: if (e.getText().length() > 0) text.setText(e.getText()); } });
```



javafx.scene.input.KeyEvent

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

KeyEventDemo

Run

The KeyCode Constants

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

Alt

Control

Meta

Shift

Example: Control Circle with Mouse and Key

```
// Create and register the handler
   btEnlarge.setOnAction(e -> circlePane.enlarge());
   btShrink.setOnAction(e -> circlePane.shrink());
****
   circlePane.setOnMouseClicked(e -> {
     if (e.getButton() == MouseButton.PRIMARY) {
       circlePane.enlarge();
           }
     else if (e.getButton() == MouseButton.SECONDARY) {
       circlePane.shrink();
           }
       });
****
   scene.setOnKeyPressed(e -> {
     if (e.getCode() == KeyCode.UP)  {
       circlePane.enlarge();
           }
     else if (e.getCode() == KeyCode.DOWN) {
       circlePane.shrink();
           }
       });
```

Run

ControlCircleWithMouseAndKey

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**.

When widthProperty is changed, clock.setWidth(pane.getWidth() is called

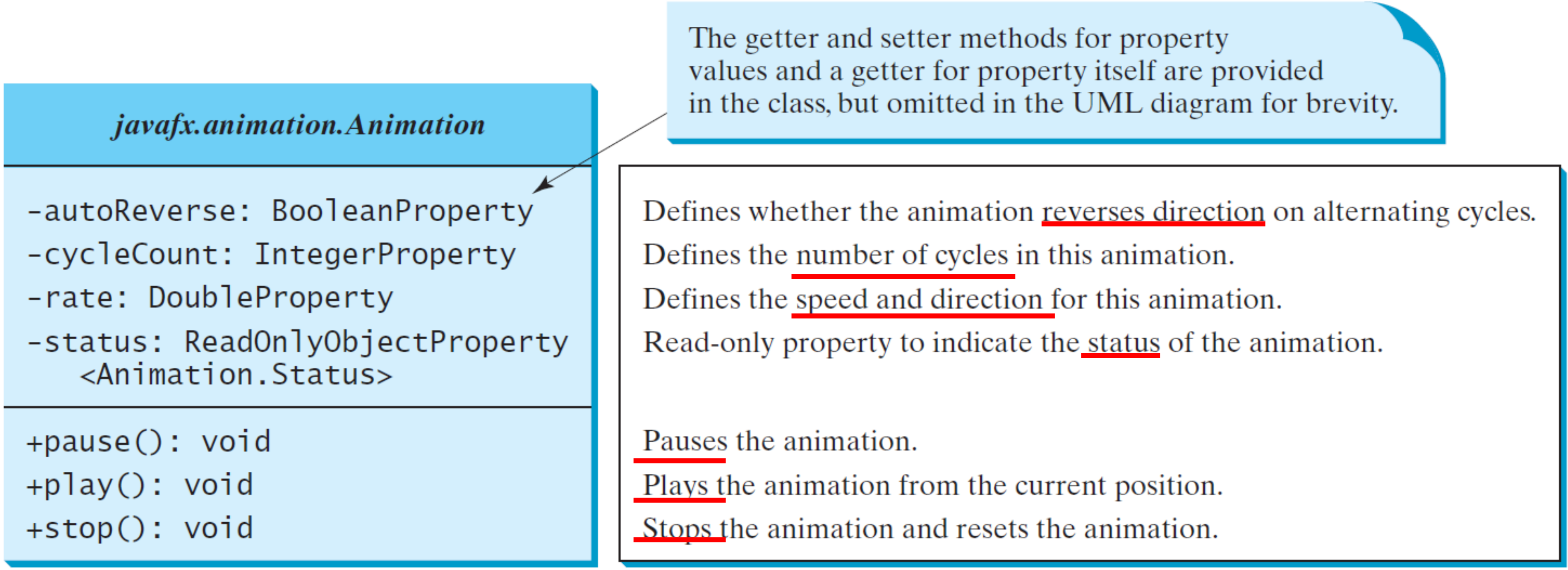
```
pane.widthProperty().addListener(ov ->
    clock.setWidth(pane.getWidth()))
```

ObservablePropertyDemo Run

DisplayResizableClock Run

Animation

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



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.

javafx.animation.Animation

-autoReverse: BooleanProperty — Defines whether the animation reverses direction on alternating cycles.

-cycleCount: IntegerProperty — Defines the number of cycles in this animation.

-rate: DoubleProperty — Defines the speed and direction for this animation.

-status: ReadOnlyObjectProperty <Animation.Status> — Read-only property to indicate the status of the animation.

+pause(): void — Pauses the animation.

+play(): void — Plays the animation from the current position.

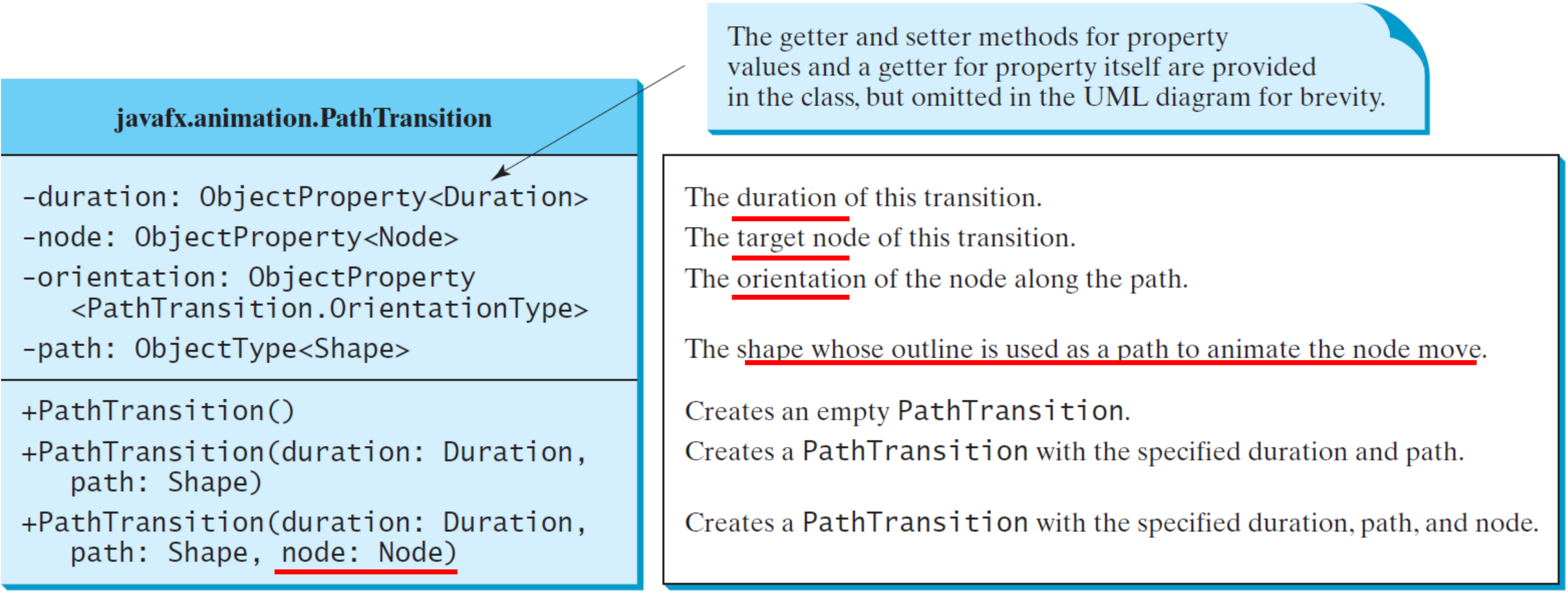
+stop(): void — Stops the animation and resets the animation.

Animation Demo

See AnimationDemo.java (old Java program)

```
// Inner class: Displaying a moving message
 static class MovingMessagePanel extends JPanel {
  private String message = "Welcome to Java";
  private int xCoordinate = 0;
  private int yCoordinate = 20;
  private Timer timer = new Timer(1000, new TimerListener());

  public MovingMessagePanel(String message) {
   this.message = message;

   // Start timer for animation
      timer.start();

   // Control animation speed using mouse buttons
   this.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
     int delay = timer.getDelay();
           if (e.getButton() == MouseEvent.BUTTON1)
              timer.setDelay(delay > 10 ? delay - 10 : 0);
           else if (e.getButton() == MouseEvent.BUTTON3)
            timer.setDelay(delay < 50000 ? delay + 10 : 50000);
          }
      });
    }
```

PathTransition



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.

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 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

Path Transition Demo

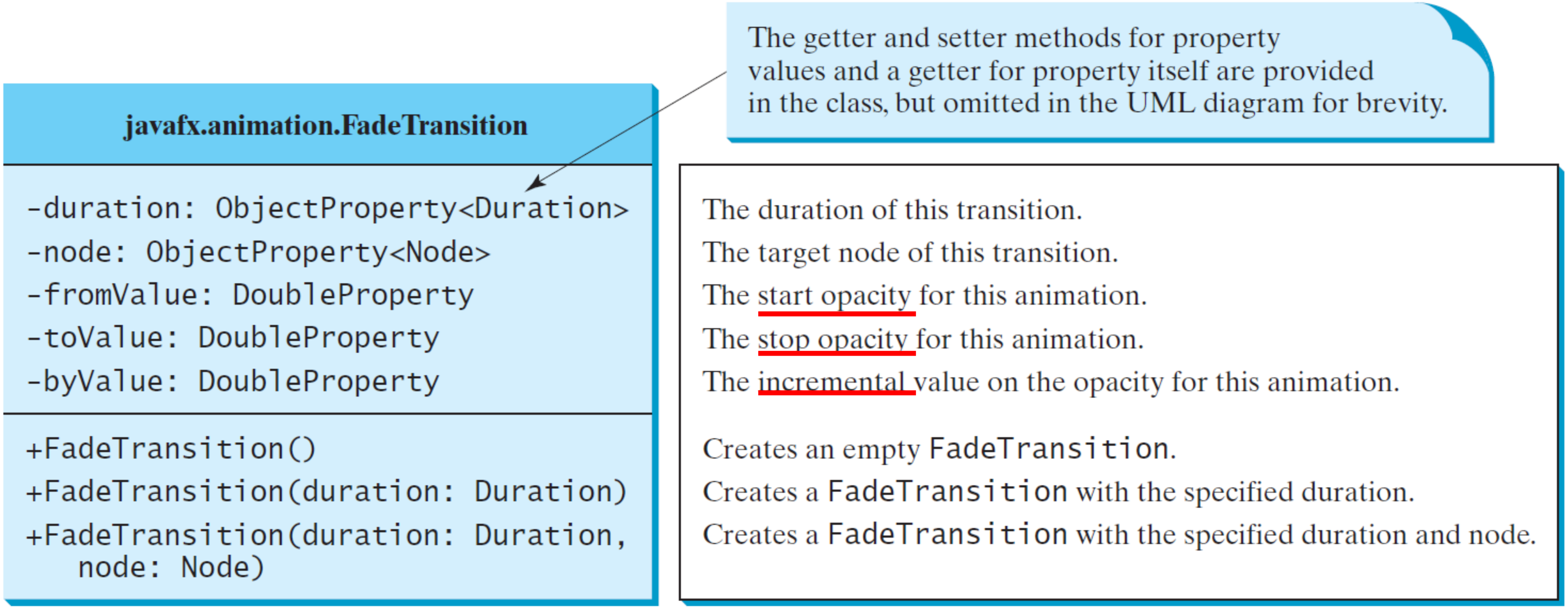
```
// Create a path transition
   PathTransition pt = new PathTransition();

   pt.setDuration(Duration.millis(4000)); // 4 seconds
   pt.setPath(circle);
   pt.setNode(rectangle);
   pt.setOrientation(
     PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
   pt.setCycleCount(Timeline.INDEFINITE);
   pt.setAutoReverse(true);
   pt.play(); // Start animation

   circle.setOnMousePressed(e -> pt.pause());
   circle.setOnMouseReleased(e -> pt.play());
```

Flag Rising Animation

```
 // Create a path transition
    PathTransition pt = new
PathTransition(Duration.millis(10000),
      new Line(100, 200, 100, 0), imageView);
    pt.setCycleCount(5);
    pt.play(); // Start animation
```

FadeTransition

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



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.

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 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

Fade Transition Demo

```
// Apply a fade transition to ellipse
   FadeTransition ft =
     new FadeTransition(Duration.millis(3000), ellipse);
   ft.setFromValue(1.0);
   ft.setToValue(0.1);
   ft.setCycleCount(Timeline.INDEFINITE);
   ft.setAutoReverse(true);
   ft.play(); // Start animation

   // Control animation
   ellipse.setOnMousePressed(e -> ft.pause());
   ellipse.setOnMouseReleased(e -> ft.play());
```

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**.

TimelineDemo

Run

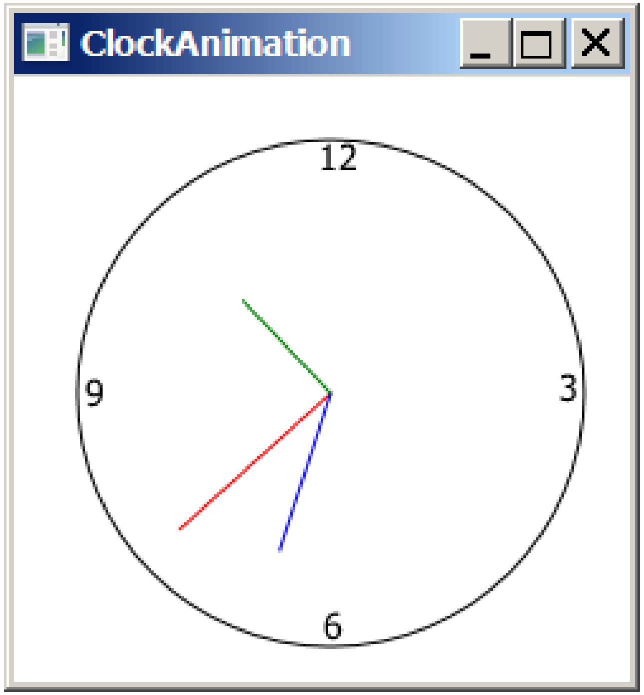
Time Line Demo

```
// Create a handler for changing text
  EventHandler<ActionEvent> eventHandler = e -> {
   if (text.getText().length() != 0) {
    text.setText("");
      }
   else {
    text.setText("Programming is fun");
      }
    };

  // Create an animation for alternating text
  Timeline animation = new Timeline(
   new KeyFrame(Duration.millis(500), eventHandler));
  animation.setCycleCount(Timeline.INDEFINITE);
  animation.play(); // Start animation

  // Pause and resume animation
  text.setOnMouseClicked(e -> {
   if (animation.getStatus() == Animation.Status.PAUSED) {
        animation.play();
      }
   else {
    animation.pause();
      }
    });
```

In every 0.5 second, eventHandler is called

Timeline inherits from **Animation**.

Clock Animation



ClockAnimation

Run

Clock Animation

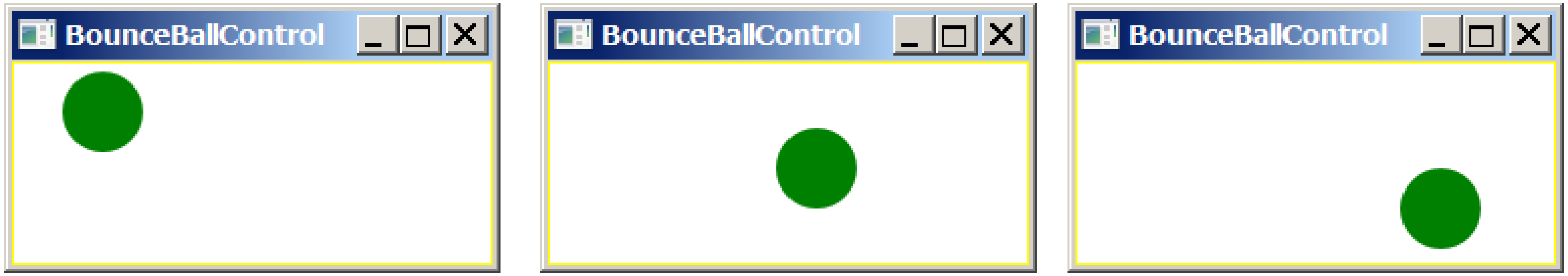
```
// Create a handler for animation
   EventHandler<ActionEvent> eventHandler = e -> {
     clock.setCurrentTime(); // Set a new clock time
       };
```

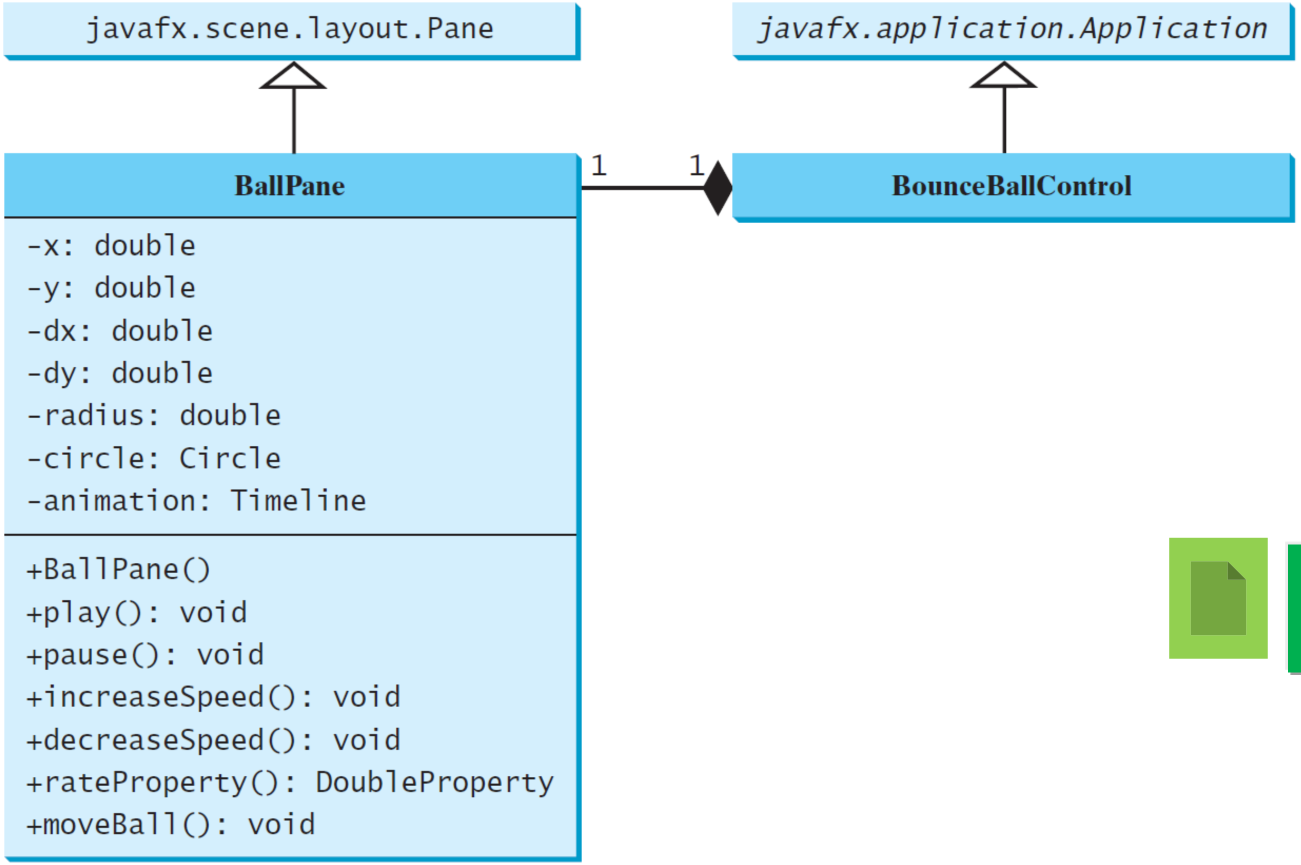
In every 1000/1000 second, eventHandler is called.

```
   // Create an animation for a running clock
   Timeline animation = new Timeline(
     new KeyFrame(Duration.millis(1000), eventHandler));
   animation.setCycleCount(Timeline.INDEFINITE);
   animation.play(); // Start animation
```

In every 1.0 second, eventHandler is called

Case Study: Bouncing Ball





javafx.scene.layout.Pane

javafx.application.Application

BallPane

-x: double
-y: double
-dx: double
-dy: double
-radius: double
-circle: Circle
-animation: Timeline

+BallPane()
+play(): void
+pause(): void
+increaseSpeed(): void
+decreaseSpeed(): void
+rateProperty(): DoubleProperty
+moveBall(): void

1 1

BounceBallControl

BallPane

BounceBallControl

Run

```
public BallPane() {
        circle.setFill(Color.GREEN); // Set ball color
        getChildren().add(circle); // Place a ball into this pane

        // Create an animation for moving the ball
        animation = new Timeline( new KeyFrame(Duration.millis(50),
                                  e -> moveBall()));
        animation.setCycleCount(Timeline.INDEFINITE);
        animation.play(); // Start animation
}
protected void moveBall() {
        // Check boundaries
        if (x < radius || x > getWidth() - radius) { dx *= -1;
        if (y < radius || y > getHeight() - radius) { dy *= -1;

        // Adjust ball position
        x += dx;
        y += dy;
        circle.setCenterX(x);
        circle.setCenterY(y); }
```

```
// Pause and resume animation
ballPane.setOnMousePressed(e -> ballPane.pause());
ballPane.setOnMouseReleased(e -> ballPane.play());

// Increase and decrease animation
ballPane.setOnKeyPressed(e -> {
        if (e.getCode() == KeyCode.UP) {
                ballPane.increaseSpeed(); }
        else if (e.getCode() == KeyCode.DOWN) {
                ballPane.decreaseSpeed(); }

public void increaseSpeed() {
        animation.setRate(animation.getRate() + 0.1); }

public void decreaseSpeed() {
        animation.setRate( animation.getRate() > 0 ?
                        animation.getRate() - 0.1 : 0); }
```