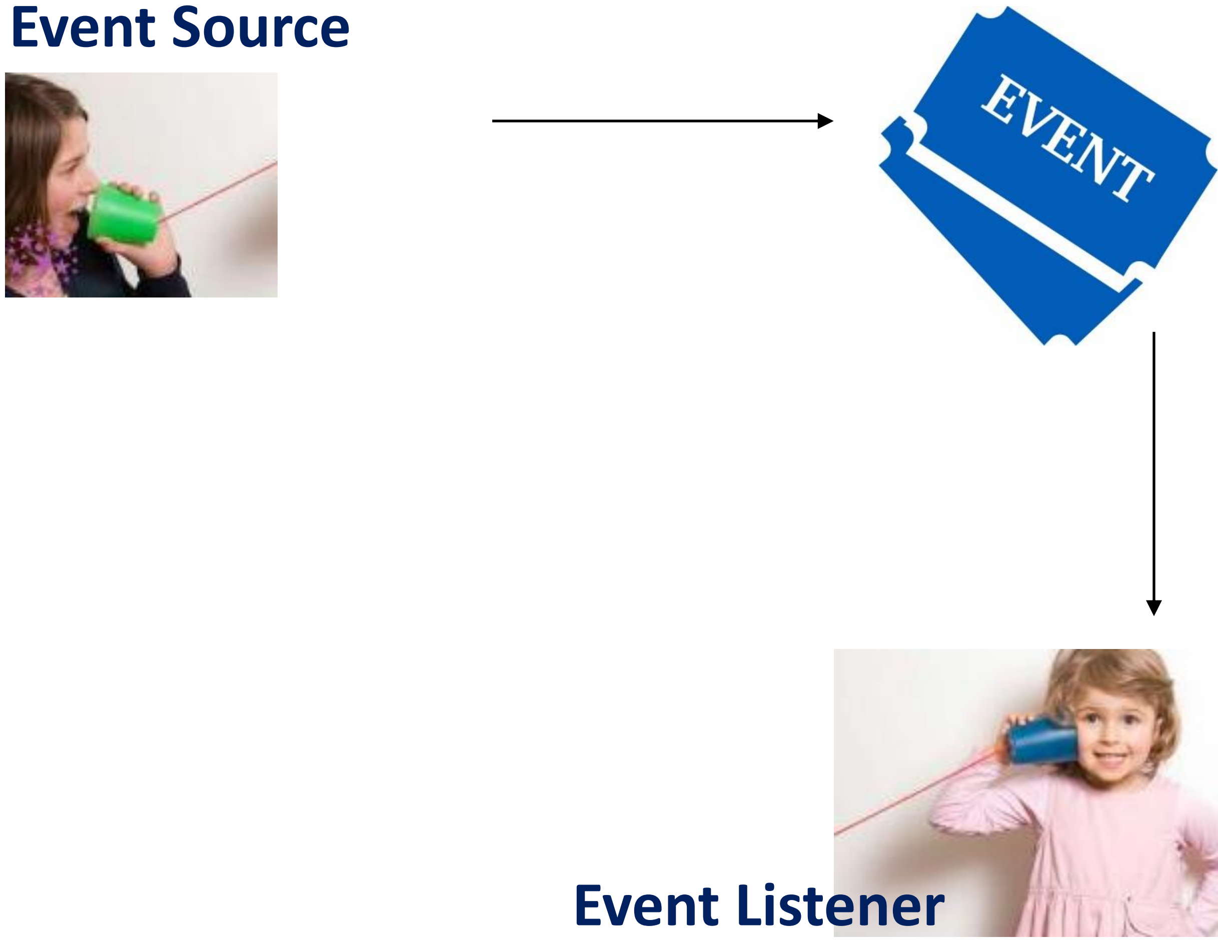
Event-Driven Programming and Animations

Shing-Chi Cheung
Computer Science and Engineering
HKUST

Procedural vs. Event-Driven Programming

- **Procedural programming** is executed in a procedural order
- In **event-driven programming**, code is executed upon activation of events such as button presses, mouse clicks and time events

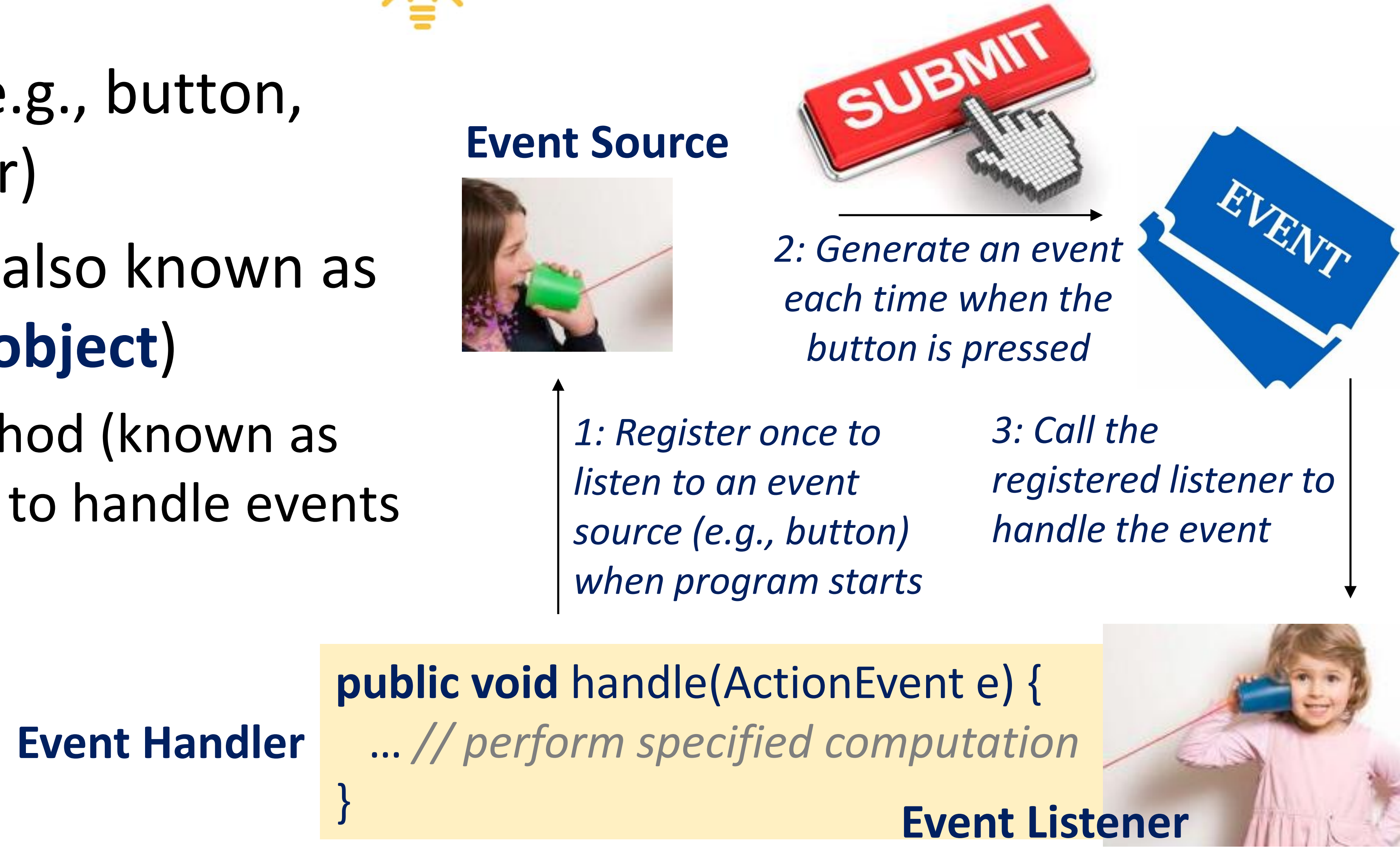


Event Source

Event Listener

Handling GUI Events

- **Event source** (e.g., button, keyboard, timer)
- **Event listener** (also known as **event handler object**)
  - Contains a method (known as **event handler**) to handle events



Event Source

2: Generate an event each time when the button is pressed

1: Register once to listen to an event source (e.g., button) when program starts

3: Call the registered listener to handle the event

Event Handler

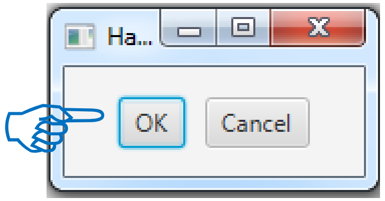
```
public void handle(ActionEvent e) {
  … // perform specified computation
}
```

Event Listener

Event Handling Example

```
public void start(Stage primaryStage) {
  ...
  var btOK = new Button("OK");
  var btCancel = new Button("Cancel");
  var handler1 = new OKHandlerClass();
  btOK.setOnAction(handler1);
  var handler2 = new CancelHandlerClass();
  btCancel.setOnAction(handler2);
  pane.getChildren().addAll(btOK, btCancel);

  var scene = new Scene(pane);
  primaryStage.setTitle("HandleEvent"); // Set the stage title
  primaryStage.setScene(scene); // Place the scene in the stage
  primaryStage.show(); // Display the stage
}
```



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

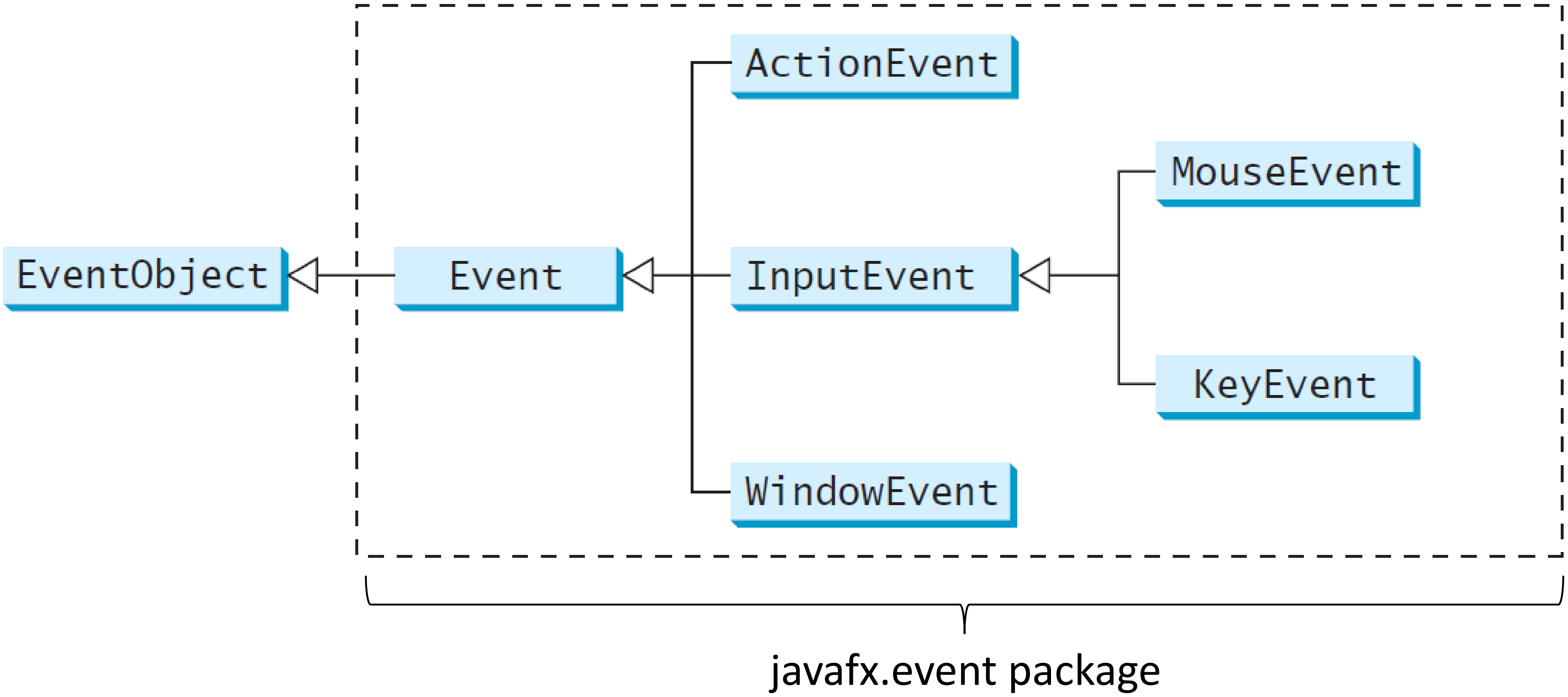
HandleEvent.java

Events

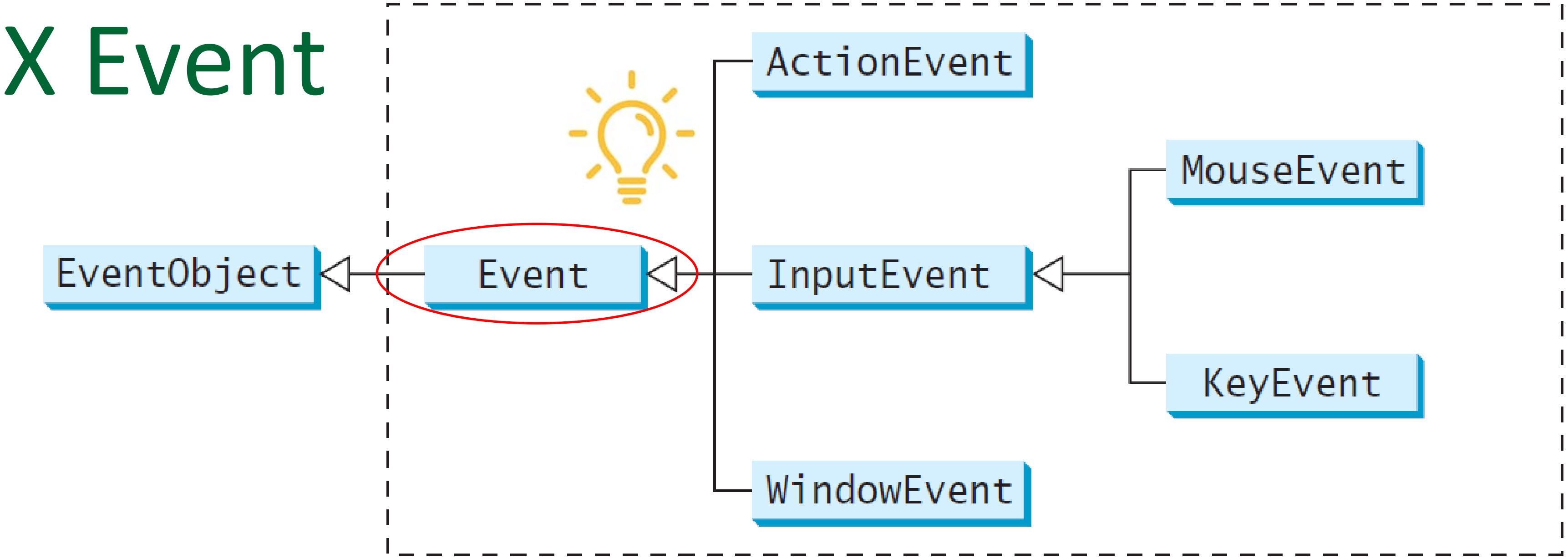
- An *event* can be defined as a type of signal to the program that something has happened.

- The event is generated by external user actions such as button presses, mouse clicks, or keystrokes.

Event Classes



JavaFX Event



- An Event object contains whatever properties are pertinent to the occurred event.
- The subclasses of Event deal with special types of events, such as button actions, window events, component events, mouse movements, and keystrokes.

Commonly-Used 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	Node, Scene	MouseEvent	setOnMouseReleased(EventHandler<MouseEvent>)
Mouse clicked	Node, Scene	MouseEvent	setOnMouseClicked(EventHandler<MouseEvent>)
Mouse dragged	Node, Scene	MouseEvent	setOnMouseDragged(EventHandler<MouseEvent>)
Key pressed	Node, Scene	KeyEvent	setOnKeyPressed(EventHandler<KeyEvent>)
Key released	Node, Scene	KeyEvent	setOnKeyReleased(EventHandler<KeyEvent>)
Key typed	Node, Scene	KeyEvent	setOnKeyTyped(EventHandler<KeyEvent>)

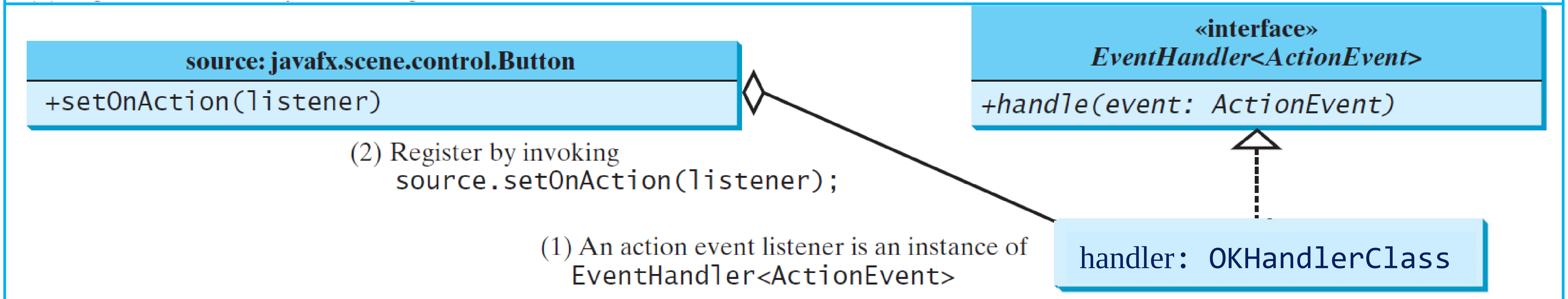
The Delegation Model: Example

```
Button btOK = new Button("OK");

var handler = new OKHandlerClass(); // (1)

btOK.setOnAction(handler); // (2)
```

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



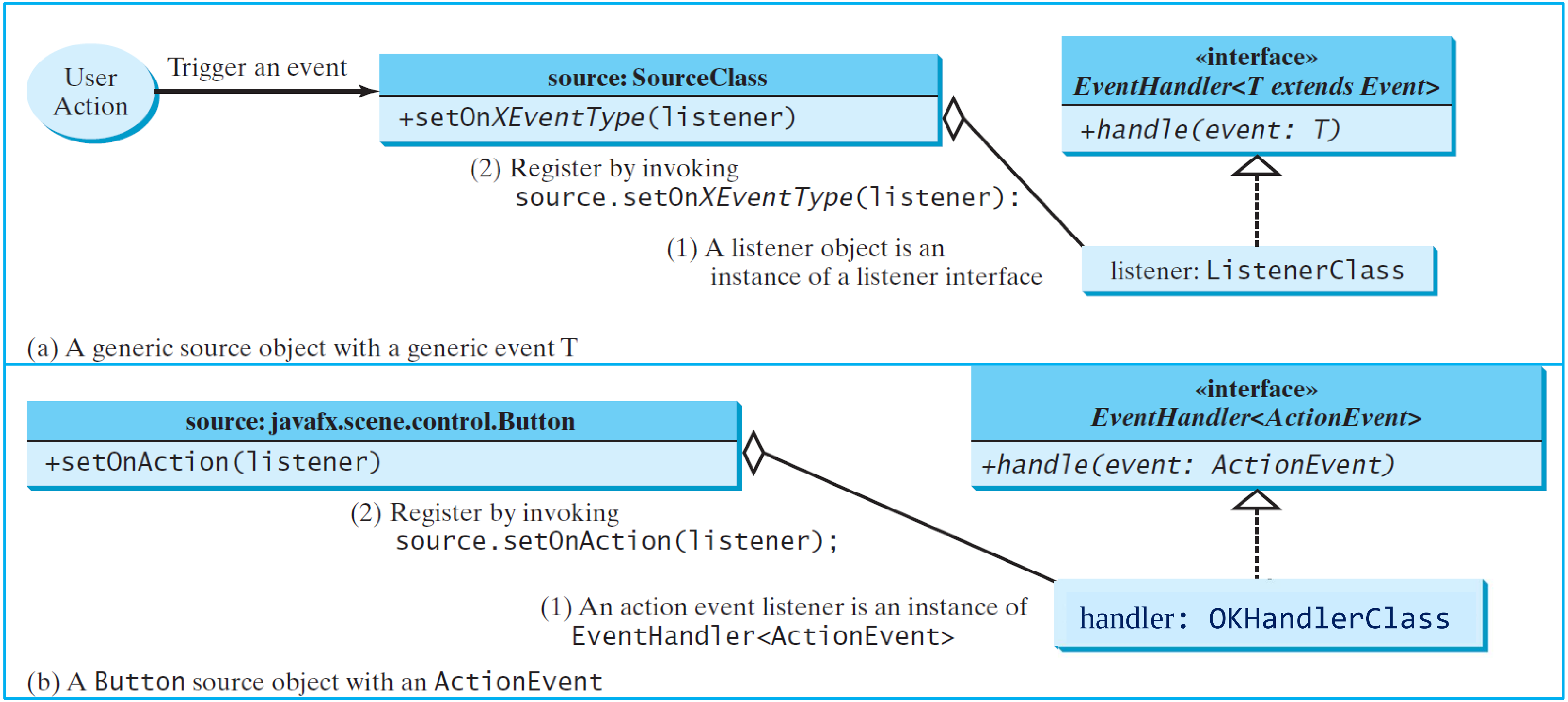
source: javafx.scene.control.Button
+setOnAction(listener)
«interface»
EventHandler<ActionEvent>
+handle(event: ActionEvent)
handler: OKHandlerClass

(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

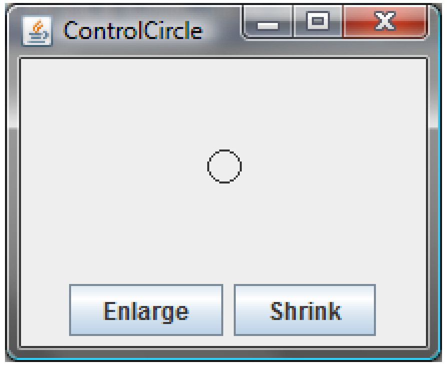

User Action
Trigger an event
source: SourceClass
+setOnXEventType(listener)
(2) Register by invoking
source.setOnXEventType(listener);
«interface»
EventHandler<T extends Event>
+handle(event: T)
(1) A listener object is an instance of a listener interface
listener: ListenerClass
source: javafx.scene.control.Button
+setOnAction(listener)
(2) Register by invoking
source.setOnAction(listener);
«interface»
EventHandler<ActionEvent>
+handle(event: ActionEvent)
(1) An action event listener is an instance of EventHandler<ActionEvent>
handler: OKHandlerClass

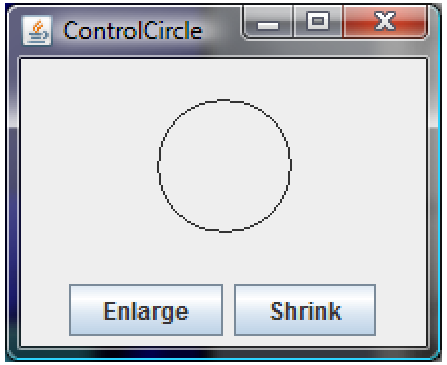
(a) A generic source object with a generic event T

(b) A Button source object with an ActionEvent

Example: ControlCircle (with listener for Enlarge)

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





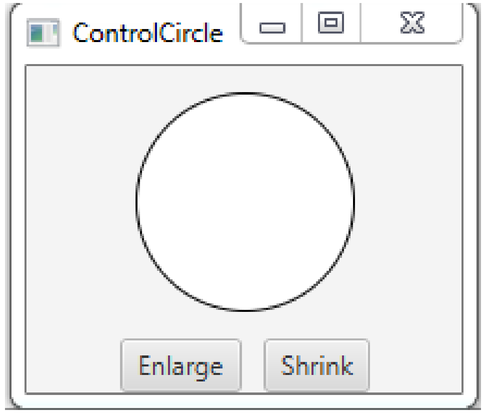
ControlCircle

```
public class ControlCircle extends Application {
  private CirclePane circlePane = new CirclePane();

  public void start(Stage primaryStage) {
    HBox hBox = new HBox(); ...
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    ...
    btEnlarge.setOnAction(new EnlargeHandler());
    btShrink.setOnAction(new ShrinkHandler());
    ...
  }

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

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



```
class CirclePane extends StackPane {
  private Circle circle = new Circle(50);

  public CirclePane() {
   getChildren().add(circle);
   ...
  }
  public void enlarge() {
     circle.setRadius(circle.getRadius() + 2); }
  public void shrink() {
   circle.setRadius(circle.getRadius() > 2 ?
   circle.getRadius() - 2 : circle.getRadius());
  }
}
```

[ControlCircle.java]

Inner Class Listeners

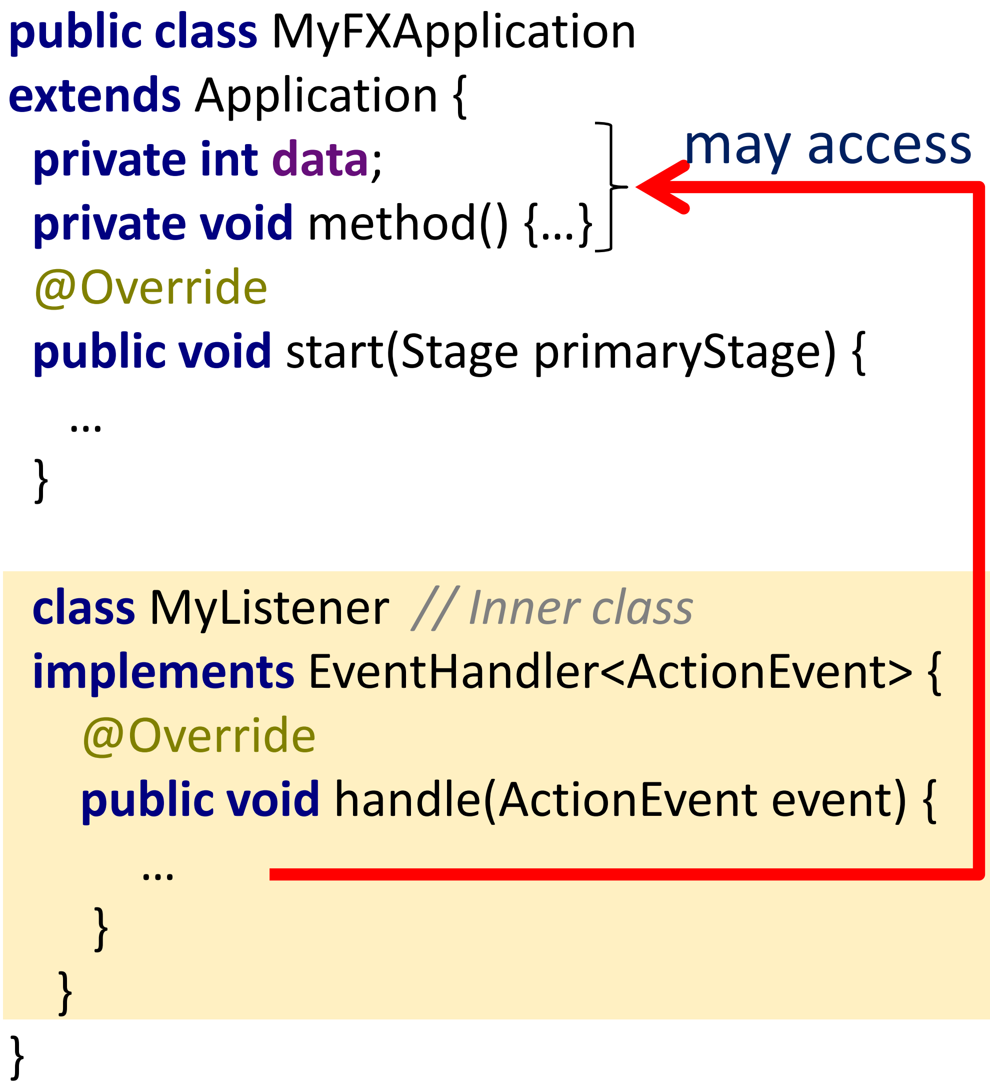
- A listener class (e.g., EnlargeHandler) is often defined specifically to create a listener for a GUI component (e.g., a button) exclusively owned by an application (e.g., ControlCircle)

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

- This listener class will not be used by other applications. It is more appropriate to define the listener class inside the concerned application class (e.g., ControlCircle) as an inner class

Inner Classes

- An inner class is a non-static class defined inside another class.
- Advantages:
  - In some applications, we can use an inner class to make programs simple.
  - An inner class can reference the data and methods defined in its outer class. We do not need to pass the reference of the outer class to the constructor of the inner class.



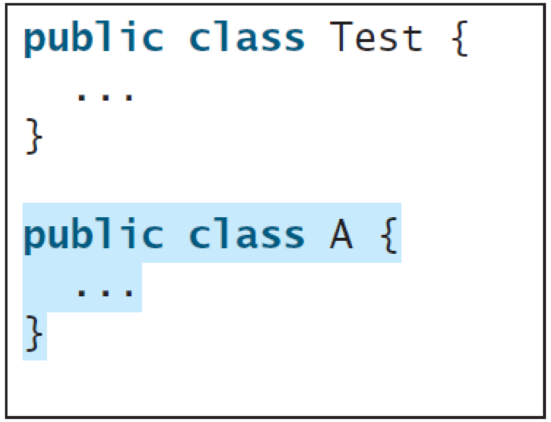
```
public class MyFXApplication
extends Application {
  private int data;
  private void method() {…}
  @Override
  public void start(Stage primaryStage) {
    …
  }

  class MyListener  // Inner class
  implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
      …
    }
  }
}
```

Inner Classes, cont.

Allow another class to access private fields and methods without breaking encapsulation (e.g., using C++ friend).

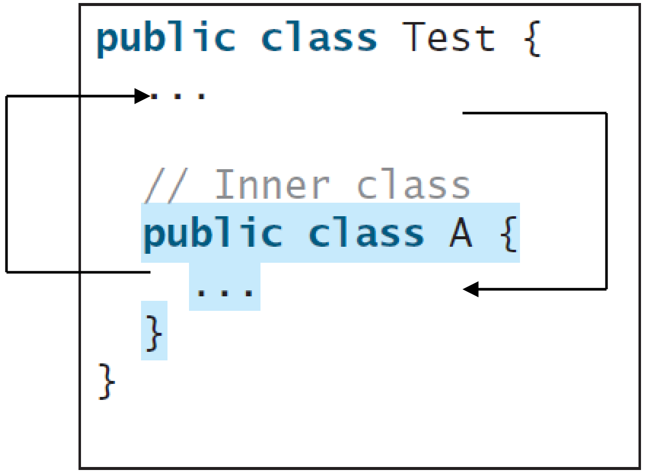
Unlike C++ friend, the access permission is bi-directional.



```
public class Test {
  ...
}

public class A {
  ...
}
```

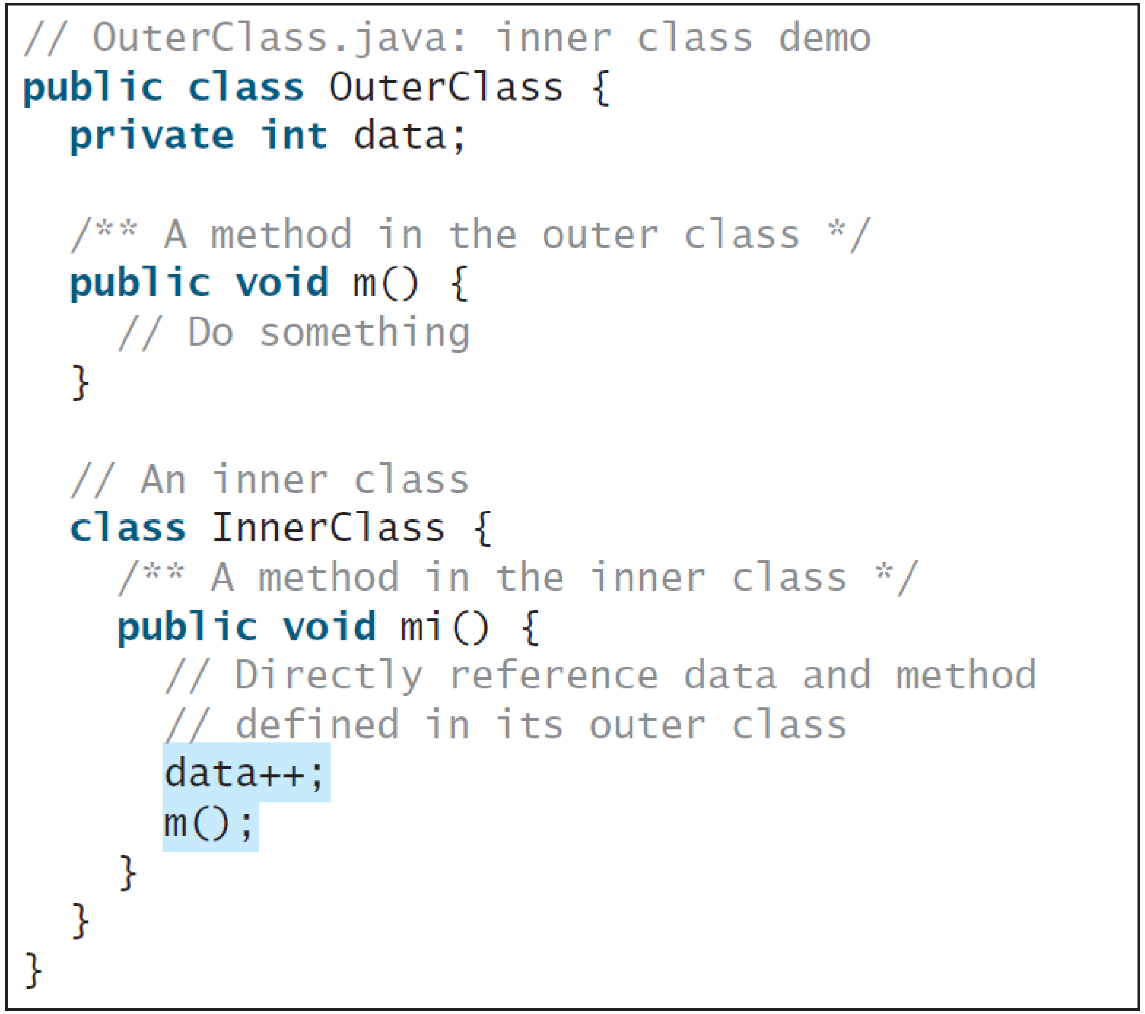
(a)



```
public class Test {
  ...

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

(b)



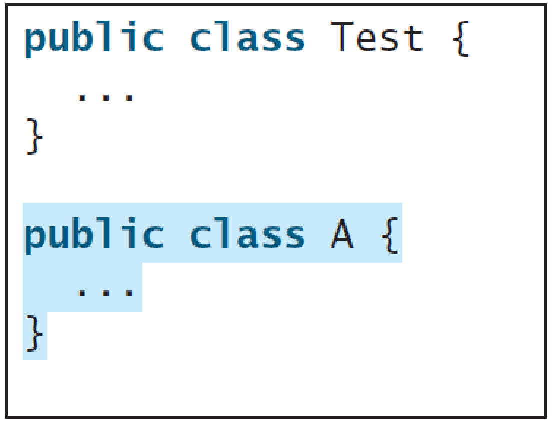
```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** 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.

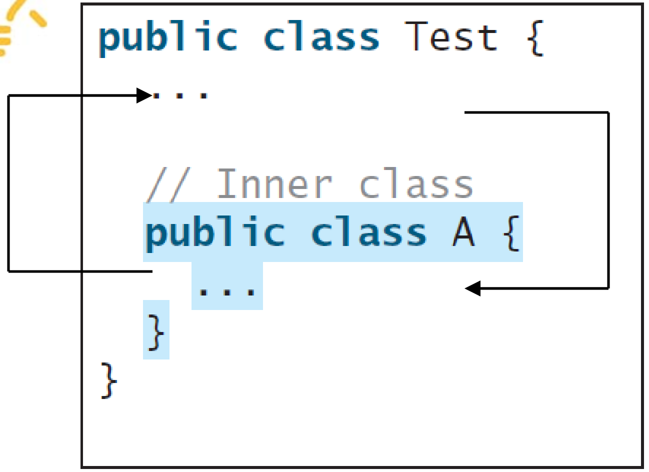


```
public class Test {
  ...
}

public class A {
  ...
}
```

(a)

Allow another class to access private fields and methods without breaking encapsulation (e.g., using friend).



```
public class Test {
  ...

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

(b)

```
public class ShowInnerClass {
 private int data;
 public void m() { // Do something
  InnerClass instance = new InnerClass();
  var i = instance.j; // Reference innerclass
 }

 class InnerClass {
  private int j;
  public void mi() {
   // Reference data and method in its outer class
   data++;
   m();
  }
 }
}
```

Inner class uses the current snapshot of the Outer class

```java
public class Outer {
  int x = 0;

  public Outer() { x++; }

  public void mutate() { x++; }

  public class Inner {
    public int get() { return x; }
  }
```

```java
public static void main(String[] args) {
  // one Outer instance shared by two Inner instances
  Outer o = new Outer();

  Outer.Inner i1 = o.new Inner();
  System.out.println(i1.get());
  o.mutate();  // increment x

  Outer.Inner i2 = o.new Inner();
  System.out.println(i2.get());
  System.out.println(i1.get());
 }
}
```

They don't have a separate copy of x in this example

Output:

```
1
2
2
```

Inner Classes (cont.)

- An inner class is compiled into a class named

  <outer class name>$<inner class name>.class.

  - For example, the class InnerClass in ShowInnerClass is compiled into ShowInnerClass$InnerClass.class.
- An inner class can be declared public, protected, or private
- Inner classes are non-static nested classes
  - Under the recent Java terminology by Oracle

Inner Classes (cont.)

- Inner classes are "instance" inner classes whose accesses must be made through an instance. That is, they are like an instance attribute of the enclosing object
  - new OutClass().new InnerClass();
- There is no point to allow static features in inner classes, for static is meant to work without an instance in the first place
- Inner classes may not declare static initializer blocks, member interfaces. They may not declare static variables, unless they are compile-time constants

No Static Features in Inner Classes

```
public class A {
    class B { static int x; }
}
// Two instances of A
var c = new A().new B();
var d = new A().new B();
```

- Static features cause ambiguity
  - Class B is declared like an instance member of A
  - So, class B is exclusively confined to an instance of A
  - The two B classes are separately confined to instances c and d of A
    - It means each B class has its own static copy of x, which can be modified independently
    - But, doing so violates the ‘static’ semantics - not confined to a particular instance
    - Such ambiguity does not occur if x is a constant
  - Java allows using B as a type
- Interfaces are contracts that must not be confined to an instance
  - They are implicitly static

ShowInnerClass.java

Static Nested Classes and Interfaces

- Static nested classes are static member classes defined in another class
- Static nested classes are often used as types
  - OuterClass.StaticNestedClass var = null; // legal
  - A static inner class can be accessed using the outer class name
- Static nested classes can be used to create instances
  - new OutClass.StaticNestedClass();
- Static nested classes may contain static members
- Nested interfaces are often used as types; they are implicitly static
  - OuterClass.NestedInterface var = null; // legal

ShowInnerClass.java

Why Anonymous Inner Classes?

- Register/Call back
  - enlargeButton.setOnAction($\overbrace{\textbf{new}\ \text{EnlargeHandler()}}^{\textit{instance of a handler class}}$);
- We use inner class listener to simplify programming
- Fancy GUIs often involve many events. So, we need to define many handlers:
  - **EnlargeHlr** implements EventHandler<>, **ShrinkHlr** implements EventHandler<>, ….
  - Tedious to come up with different names for all these handlers
  - Observation: Many event sources don’t share handler classes
- Can we define a handler class on-the-fly anonymously when we register its instance to listen to an event source?
  - We call such a handler class an anonymous inner class
  - It does not have a class name but it can access all fields in the outer class.

Anonymous Inner Classes

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

```
new <interface name> / <super class name>  ( <parameter list > ) {
  // Implement or override methods in interface or superclass
  // Implement other helper methods if necessary
}
```

Anonymous Inner Class

May an anonymous inner class implement multiple interfaces?

```
public class ControlCircle extends Application {
  public void start(Stage primaryStage) {
    …
    Button btEnlarge = new Button("Enlarge");
    btEnlarge.setOnAction(
     new EnlargeHandler()
    );
    …
  }
```

↑ *substitute*

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

```
public class ControlCircle extends Application {
  public void start(Stage primaryStage) {
    …
    Button btEnlarge = new Button("Enlarge");
    btEnlarge.setOnAction(
     new ~~class EnlargeHandler implements~~
       EventHandler<ActionEvent>() {
       public void handle(ActionEvent e) {
          circlePane.enlarge(); }
      }
    );
    …
  }
```

one statement

Anonymous Inner Class Demo

```
// Create and register the handler
btUp.setOnAction(new EventHandler<ActionEvent>() {
 @Override // Override the handle method
 public void handle(ActionEvent e) {
  text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
 }
});

btDown.setOnAction(new EventHandler<ActionEvent>() {
 @Override // Override the handle method
 public void handle(ActionEvent e) {
  text.setY(text.getY() < pane.getHeight() ?
   text.getY() + 5 : pane.getHeight());
 }
});
```

```
btLeft.setOnAction(new EventHandler<ActionEvent>() {
 @Override // Override the handle method
 public void handle(ActionEvent e) {
  text.setX(text.getX() > 0 ? text.getX() - 5 : 0);
 }
});

btRight.setOnAction(new EventHandler<ActionEvent>() {
 @Override // Override the handle method
 public void handle(ActionEvent e) {
  text.setX(text.getX() < pane.getWidth() - 100?
   text.getX() + 5 : pane.getWidth() - 100);
 }
});
```

AnonymousHandlerDemo.java

Anonymous Inner Classes

new <interface name> / <super class name> (<parameter list >) {…}

- An anonymous inner class must always implement an interface or extend a superclass; it does not need an explicit **implements** or **extends** clause
- An anonymous inner class must implement all the abstract methods in the interface or superclass

Q: May an anonymous innerclass have explicitly defined constructors

A: No. Upon object creation, Java executes super(<parameter list>);

Anonymous Inner Classes

new <interface name> / <super class name> (<parameter list >) {…}

- An anonymous inner class always uses the superclass constructor with matching parameters to create an instance. **Why?**
  - Q: Which constructor will be used if an anonymous inner class implements an interface?
  - Q: What if an anonymous inner class needs to perform some initialization?
- 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

TestAnonymousInnerClass.java

Simplifying Event Handing Using Lambda Expressions

- Lambda expression is an **important feature** added to Java 8
- A lambda expression can be viewed as the body of an anonymous method that implements the abstract method in an interface
- For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines

```
btUp.setOnAction(
  new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
      text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
    }
  }
);
```

(a) Anonymous inner class event handler

```
/*
    Much less verbose, specifying exactly only
    the needed information. Loved by developers!
*/
btUp.setOnAction(e -> {
  text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
});
```

(b) Lambda expression event handler

LambdaHandlerDemo.java

Basic Syntax for a Lambda Expression

Syntax:

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

- The parentheses can be omitted if there is only one parameter
- The data type of a parameter can be explicitly declared or **left unspecified** (i.e., $\underbrace{\text{implicitly inferred by the compiler}}_{\text{usual practice}}$)

Type Inference of Lambda Expression

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

1. Java compiler infers that the argument’s type must be EventHandler<ActionEvent>
2. Java compiler infers that the concerned lambda expression must define the abstract method, which is handle(ActionEvent evt) in the interface

```
interface EventHandler<ActionEvent> {
  public abstract void handle(ActionEvent evt);
}
```

3. Java compiler infers that e in the lambda expression must corresponds to evt in handle’s method parameter
4. Java compiler infers that the type of e in the lambda expression is ActionEvent, and {text.setY(text.getY() > 10 ? …);} defines the overriding handle method’s body

Single Abstract Method Interface (SAM)

- Java compiler assumes that all the statements **{…}** in an lambda expression **e -> {…}** is written to define one abstract method

```
@FunctionalInterface
interface EventHandler<ActionEvent> {
  public abstract void handle(ActionEvent evt);
}
```

- So, the interface (e.g., EventHandler<ActionEvent>) implemented by an lambda expression must contain exactly one abstract method
- A functional interface (or Single Abstract Method (SAM) interface) is an interface that contains exactly one abstract method. It can be optionally declared by the @FunctionalInterface annotation

Action.java

More Lambda Expression Example

```
public class LambdaNames {
 public static void main(String [] args) {
  myName( new Names() {
   @Override
   public void sayName(String n) {
    System.out.println("My Name is " + n);
   }
  }, “John”);
 }
 private static void myName(Names nameInstance, String name) {
  nameInstance.sayName(name);
 }
}
```

```
@FunctionalInterface
interface Names {
 void sayName(String name);
}
```

Lambda Expression Example

The lambda expression creates an object that implements the abstract method. Java infers the argument types and return types.

```
myName(n->System.out.println("My Name is "+n)
, “John”);
```

```
public class LambdaNames {
  public static void main(String [] args) {
    myName( new Names() {
      @Override
      public void sayName(String n) {
        System.out.println("My Name is " + n);
      }
    }, “John”);
  }
  private static void myName(Names nameInstance, String name) {
    nameInstance.sayName(name);
  }
}
```

refers to a functional interface

```
@FunctionalInterface
interface Names {
  void sayName(String name);
}
```

LambdaNames.java

Lambda Expression Example

```
public class LambdaNames {
 public static void main(String [] args) {
  myName(n->System.out.println("My Name is "+n), “John”);
 }
```

```
@FunctionalInterface
interface Names {
 void sayName(String name);
}
```

refers to a functional interface

```
 private static void myName(Names nameInstance, String name) {
  nameInstance.sayName(name);
 }
}
```

LambdaNames.java

Lambda Expression Example

Behavior of sayName() is configured on-the-fly when it is called.

```
public class LambdaNames {
 public static void main(String [] args) {
  myName(n->System.out.println("My Name is "+n), “John”);
  myName(n->System.out.println(“Your Name is "+n), “Mary”);
 }
```

```
@FunctionalInterface
interface Names {
 void sayName(String name);
}
```

refers to a functional interface

```
 private static void myName(Names nameInstance, String name) {
  nameInstance.sayName(name);
 }
}
```

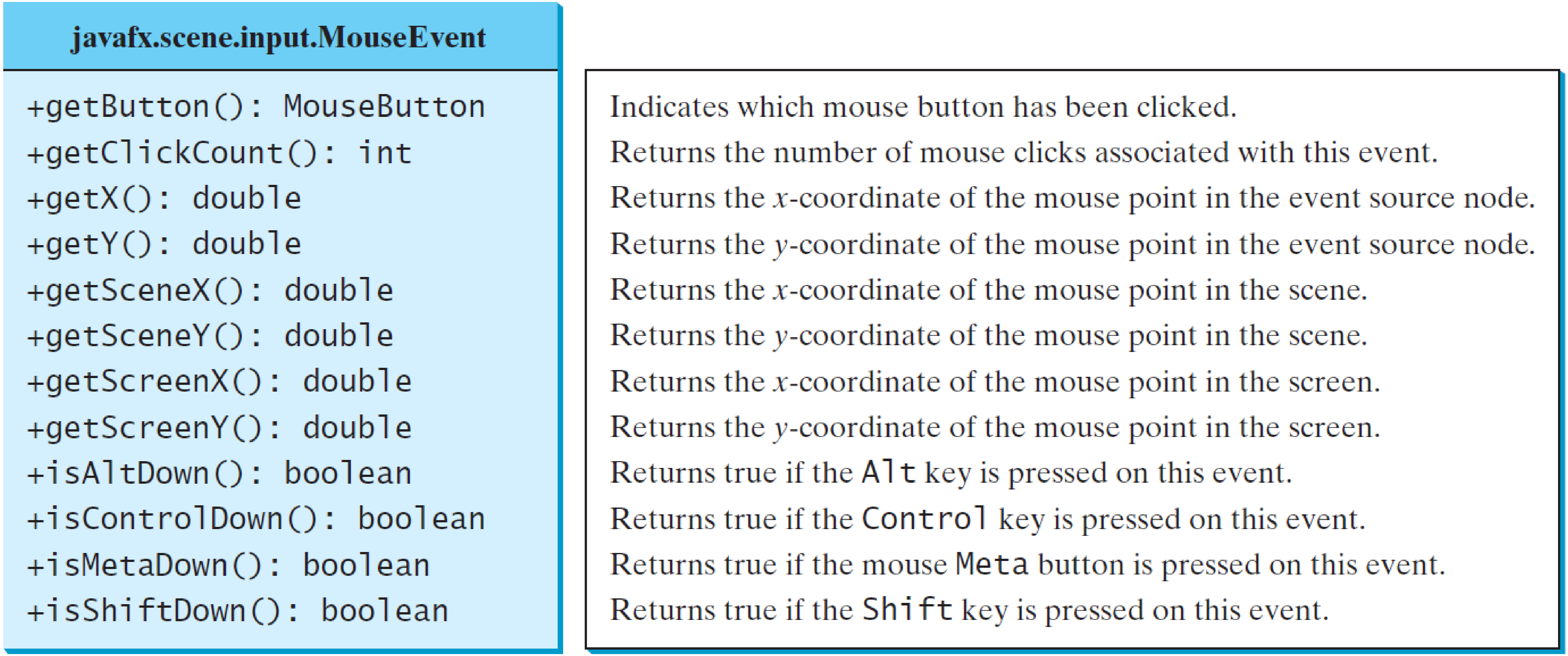
LambdaNames.java

More Lambda Expression Example: Loan Calculator

```
@Override
public void start(Stage primaryStage) {
    …
    btCalculate.setOnAction(e -> calculateLoanPayment());
    …
}
private void calculateLoanPayment() {
    …
}
```

LoanCalculator.java

MouseEvent

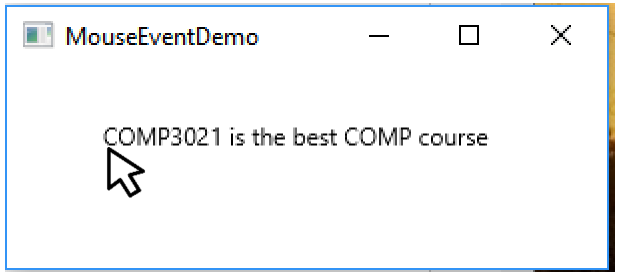


javafx.scene.input.MouseEvent

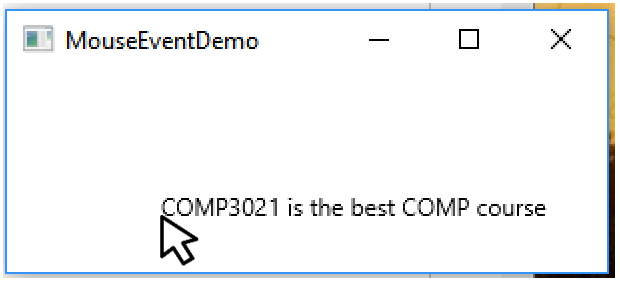
`+getButton(): MouseButton`	Indicates which mouse button has been clicked.
`+getClickCount(): int`	Returns the number of mouse clicks associated with this event.
`+getX(): double`	Returns the *x*-coordinate of the mouse point in the event source node.
`+getY(): double`	Returns the *y*-coordinate of the 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.

MouseEvent

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



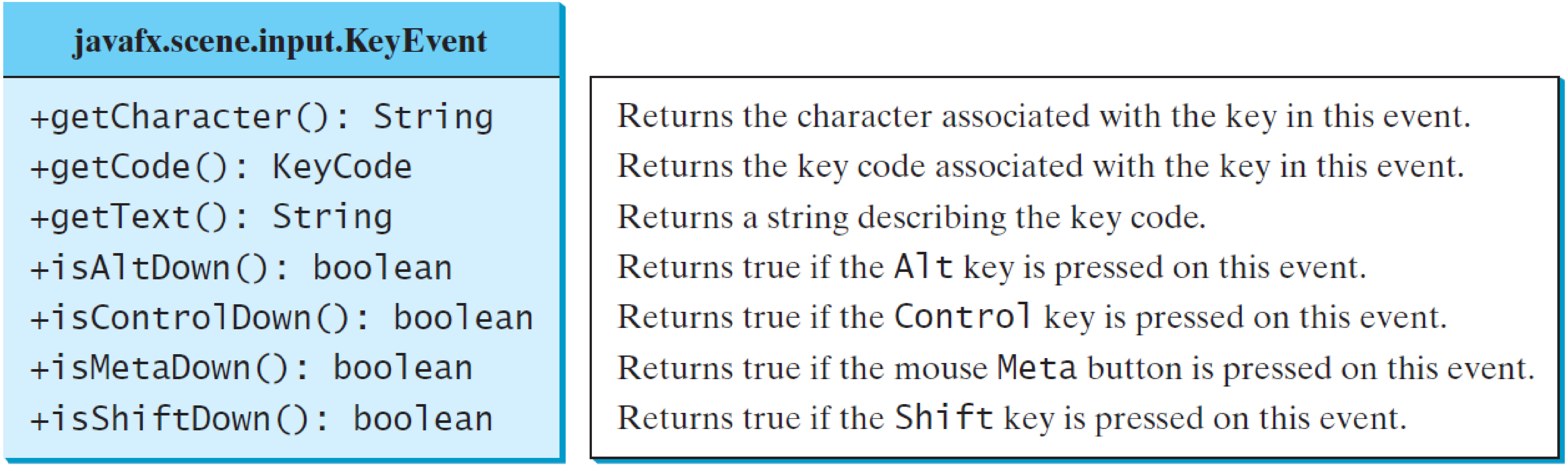
COMP3021 is the best COMP course



COMP3021 is the best COMP course

MouseEventDemo.java

The KeyEvent Class



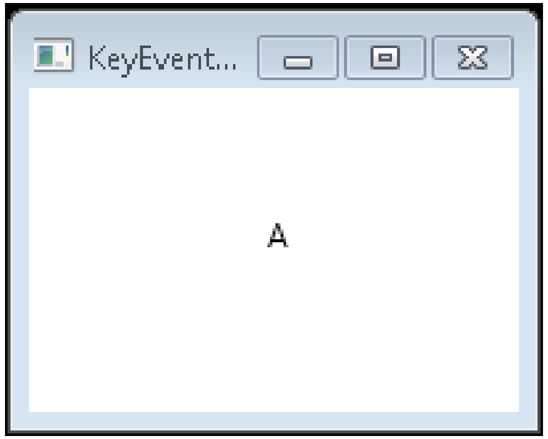
javafx.scene.input.KeyEvent	
`+getCharacter(): String`	Returns the character associated with the key in this event.
`+getCode(): KeyCode`	Returns the key code associated with the key in this event.
`+getText(): String`	Returns a string describing the key code.
`+isAltDown(): boolean`	Returns true if 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.

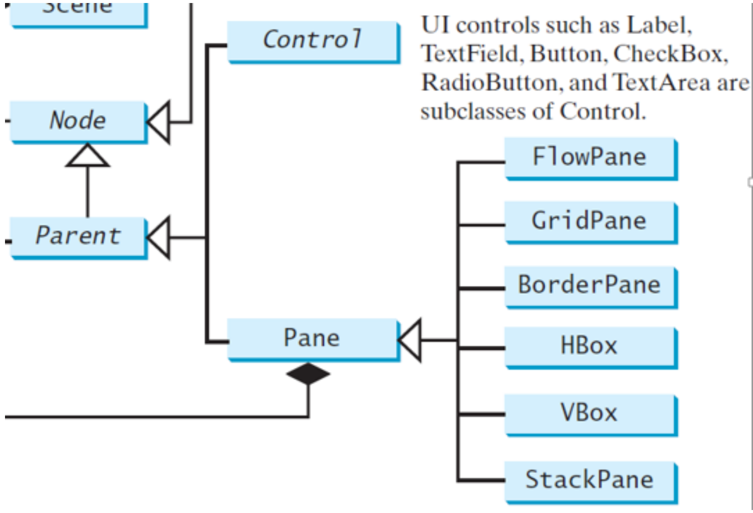
Node.setOnKeyPressed(e -> …)

KeyEventDemo.java

public void javafx.scene.Node.requestFocus()

- Requests that this Node owns the keyboard input focus.
- This Node's top-level ancestor become the focused window (stage).
- To own the focus, the node must be part of a scene, it and all of its ancestors must be visible and active.

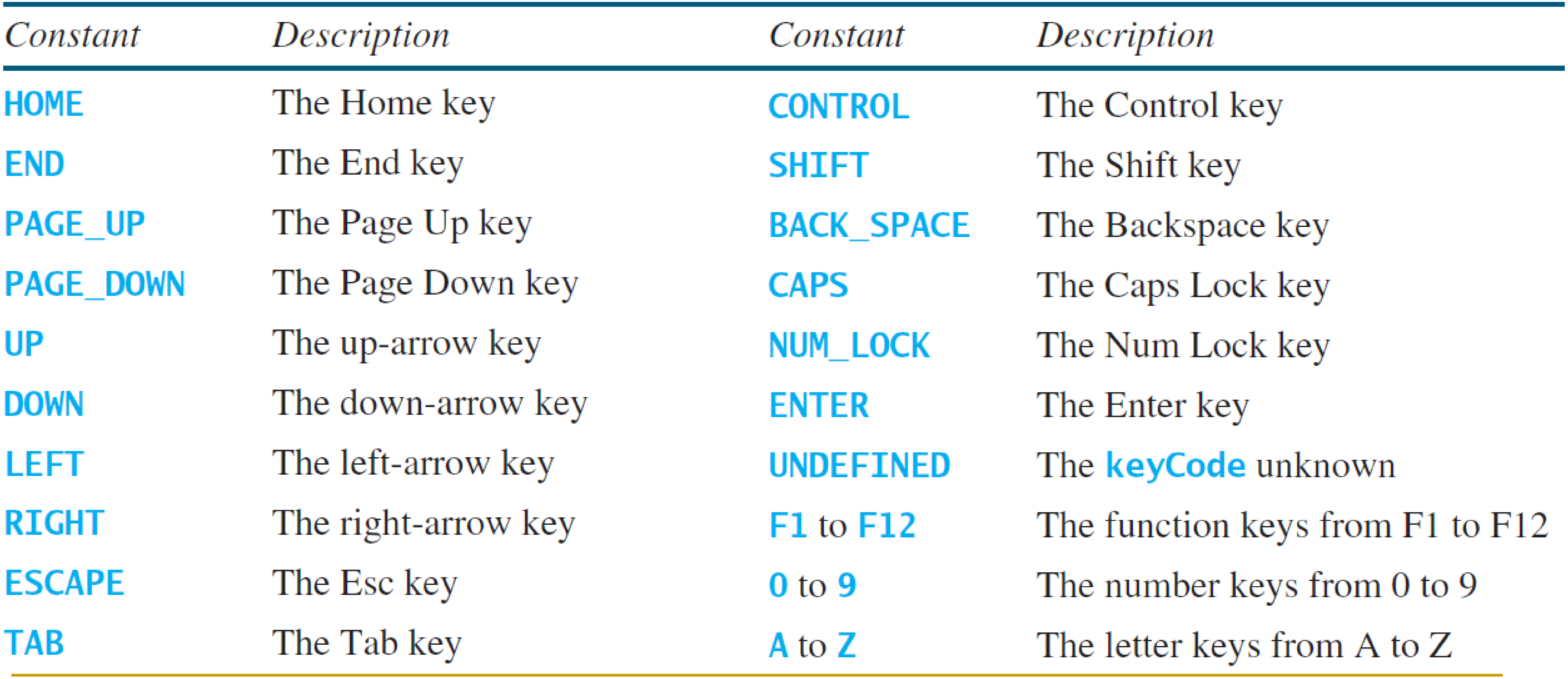




UI controls such as Label, TextField, Button, CheckBox, RadioButton, and TextArea are subclasses of Control.

KeyEventDemo.java

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

Example: Control Circle with Mouse and Key

```
circlePane.setOnMouseClicked(e -> {
 if (e.getButton()==MouseButton.PRIMARY) {
  circlePane.enlarge();
 }
 else if (e.getButton()==MouseButton.SECONDARY) {
  circlePane.shrink();
 }
});
```

```
circlePane.setOnKeyPressed(e -> {
 if (e.getCode()==KeyCode.UP) {
  circlePane.enlarge();
 }
 else if (e.getCode()==KeyCode.DOWN) {
  circlePane.shrink();
 }
 });
```

ControlCircleWithMouseAndKey.java

Summary of Node.setOn*Event*(e -> ...)

- setOnAction, e.g., Button.setOnAction(e -> ...)
- setOnMouseDragged, e.g., Text.setOnMouseDragged(e -> ...)
- setOnMouseClicked, e.g., Pane.setOnMouseClicked(e -> ...)
- setOnKeyPressed, e.g., Pane.setOnKeyPressed(e -> ....)

We will revisit lambda in more detail at a later topic

Listeners for Observable Objects

- We can add a listener to perform a task when there is 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 there is a value change in the object’s property, 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.

Listeners for Observable Objects - Example

```
public class ObservablePropertyDemo {
  public static void main(String[] args) {
    DoubleProperty balance = new SimpleDoubleProperty();
    balance.addListener(new InvalidationListener() {
      public void invalidated(Observable ov) {
        System.out.println("The new value is " +  balance.doubleValue());
      }
    });
    balance.set(4.5);
  }
}
```

Output:

```
The new value is 4.5
```

ObservablePropertyDemo.java

Listeners for Observable Objects – Lambda Example

```
public class ObservablePropertyDemoLambda {
  public static void main(String[] args) {
    DoubleProperty balance = new SimpleDoubleProperty();
    balance.addListener(e -> System.out.println("The new value is " +
      balance.doubleValue()));
    balance.set(4.5);
  }
}
```

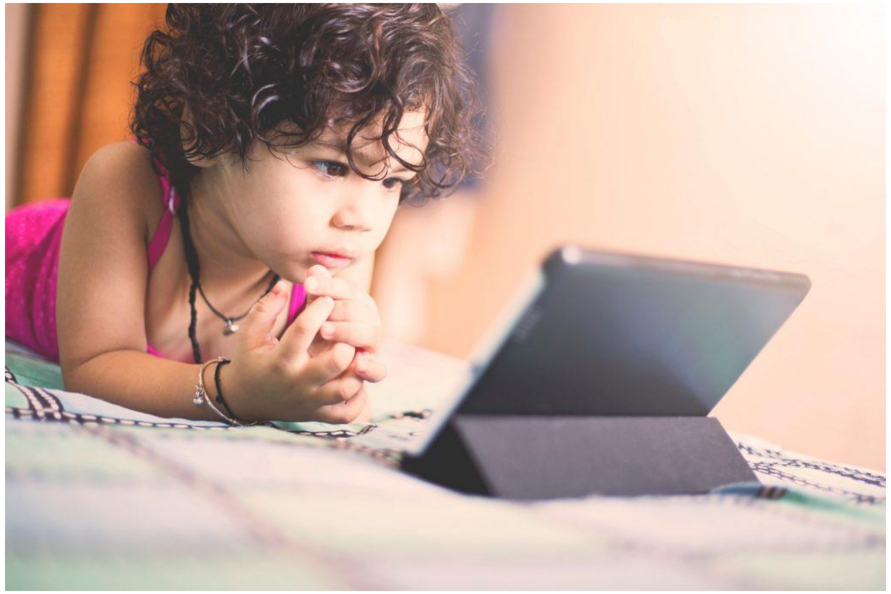
Output:

```
The new value is 4.5
```

Replaced by java.beans.PropertyChangeSupport and java.beans.PropertyChangeListener, which support a richer event model
Example: https://gist.github.com/mtorchiano/e69ac7e309fee81bd17f4f0740b9ffa9

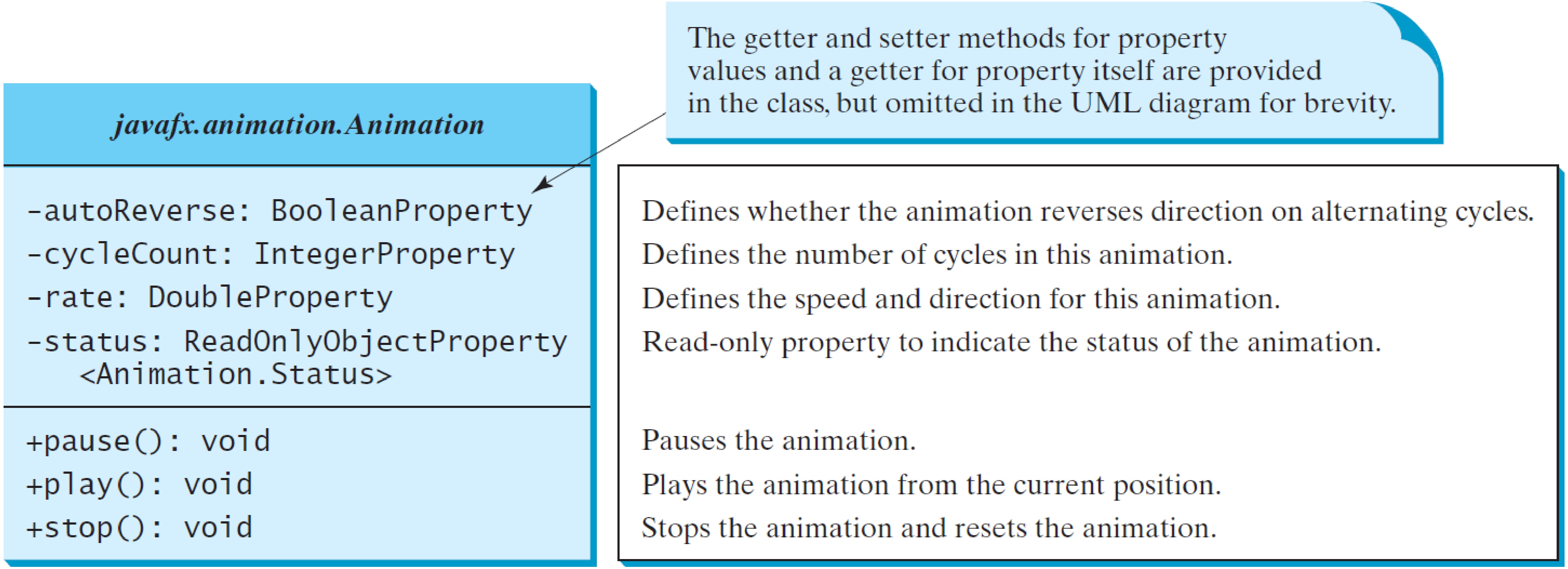
ObservablePropertyDemoLambda.java

Selected Animation Classes and Examples (for self study after class)



Animation

- JavaFX provides the Animation base 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
-cycleCount: IntegerProperty
-rate: DoubleProperty
-status: ReadOnlyObjectProperty
   <Animation.Status>

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

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.

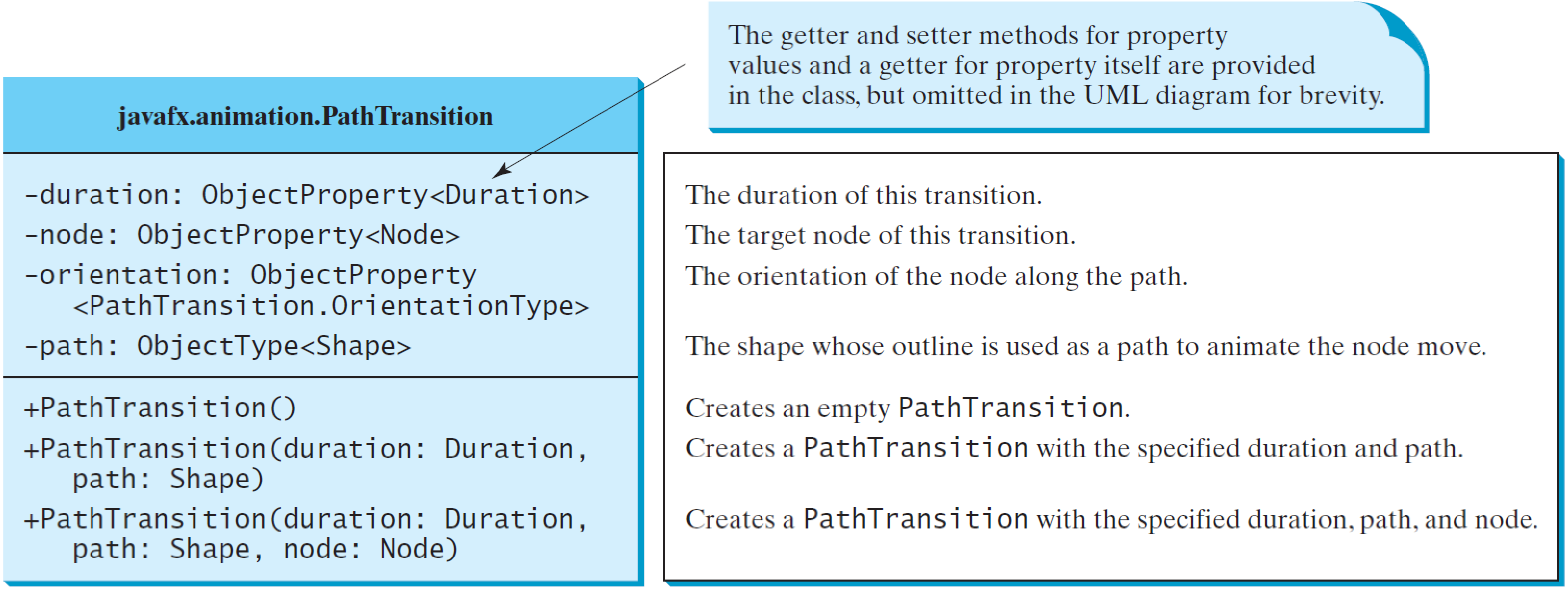
Read-only property to indicate the status of the animation.

Pauses the animation.

Plays the animation from the current position.

Stops the animation and resets the animation.

PathTransition (a subclass of Animation)



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

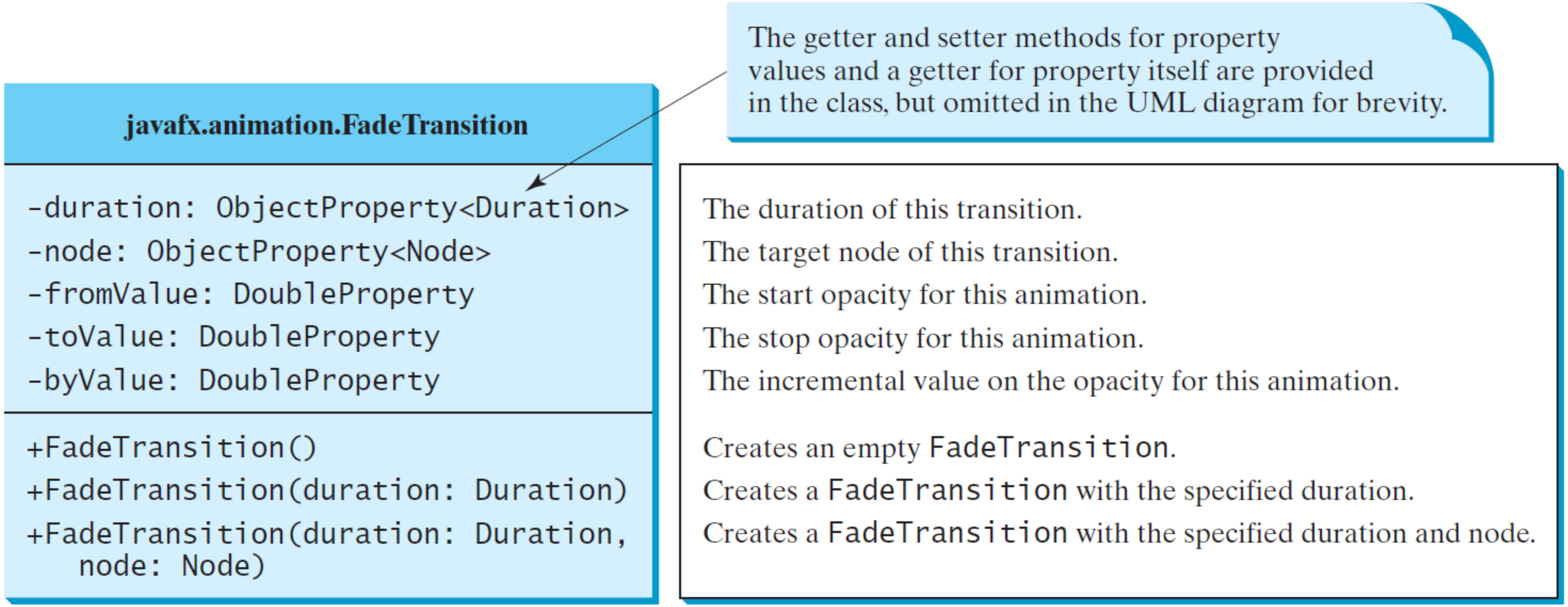
`-duration: ObjectProperty<Duration>`	The duration of this transition.
`-node: ObjectProperty<Node>`	The target node of this transition.
`-orientation: ObjectProperty <PathTransition.OrientationType>`	The orientation of the node along the path.
`-path: ObjectType<Shape>`	The shape whose outline is used as a path to animate the node move.
`+PathTransition()`	Creates an empty `PathTransition`.
`+PathTransition(duration: Duration, path: Shape)`	Creates a `PathTransition` with the specified duration and path.
`+PathTransition(duration: Duration, path: Shape, node: Node)`	Creates a `PathTransition` with the specified duration, path, and node.

PathTransitionDemo.java

FlagRisingAnimation.java

FadeTransition (a subclass of Animation)

- 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.

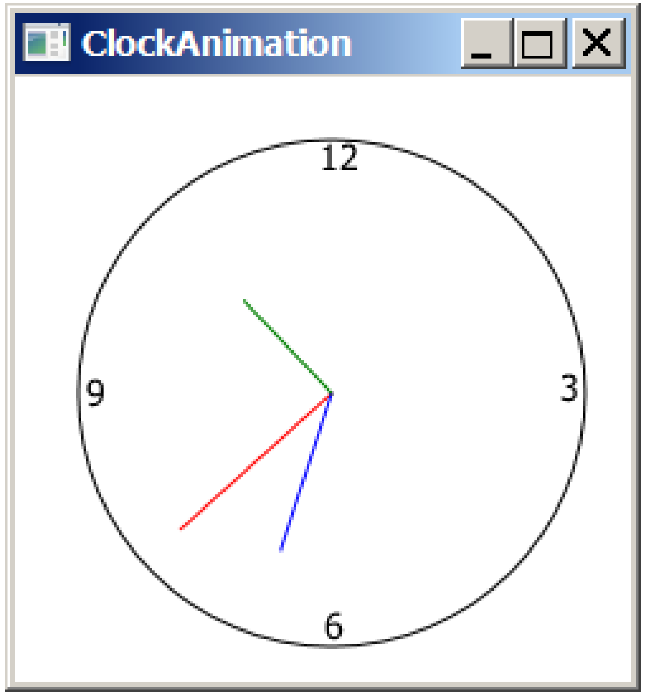
FadeTransitionDemo.java

Timeline (a subclass of Animation)

- **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**.

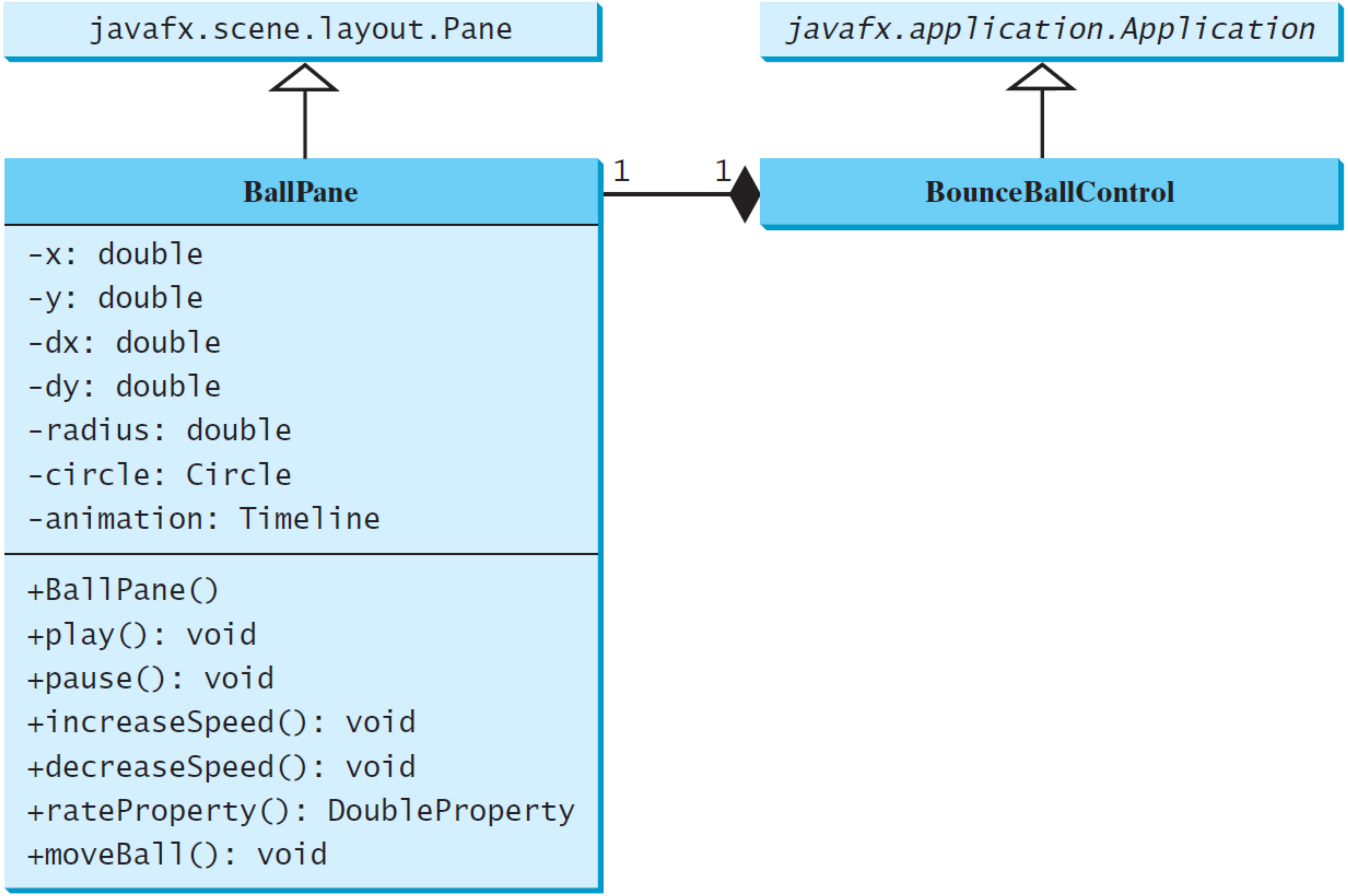
TimeLineDemo.java

Clock Animation



ClockAnimationDemo.java

Case Study: Bouncing Ball



javafx.scene.layout.Pane

javafx.application.Application

1

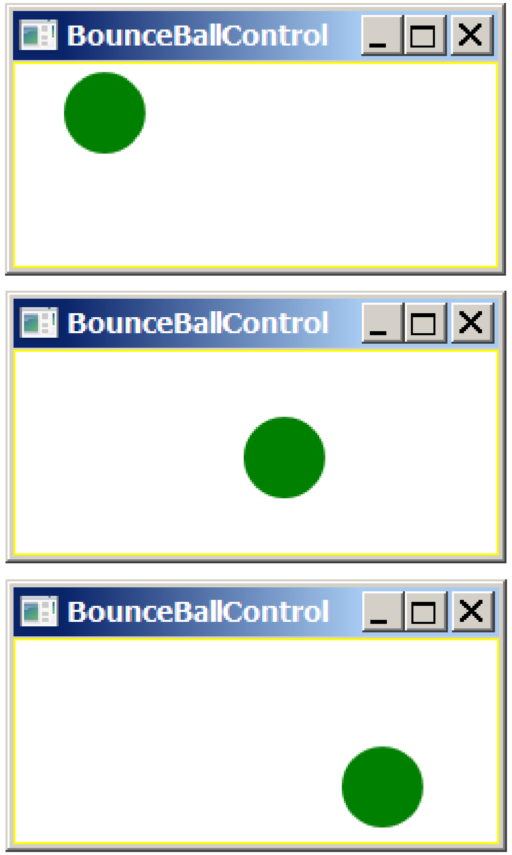
1

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

BounceBallControl



BounceBallControl.java

BallPane.java