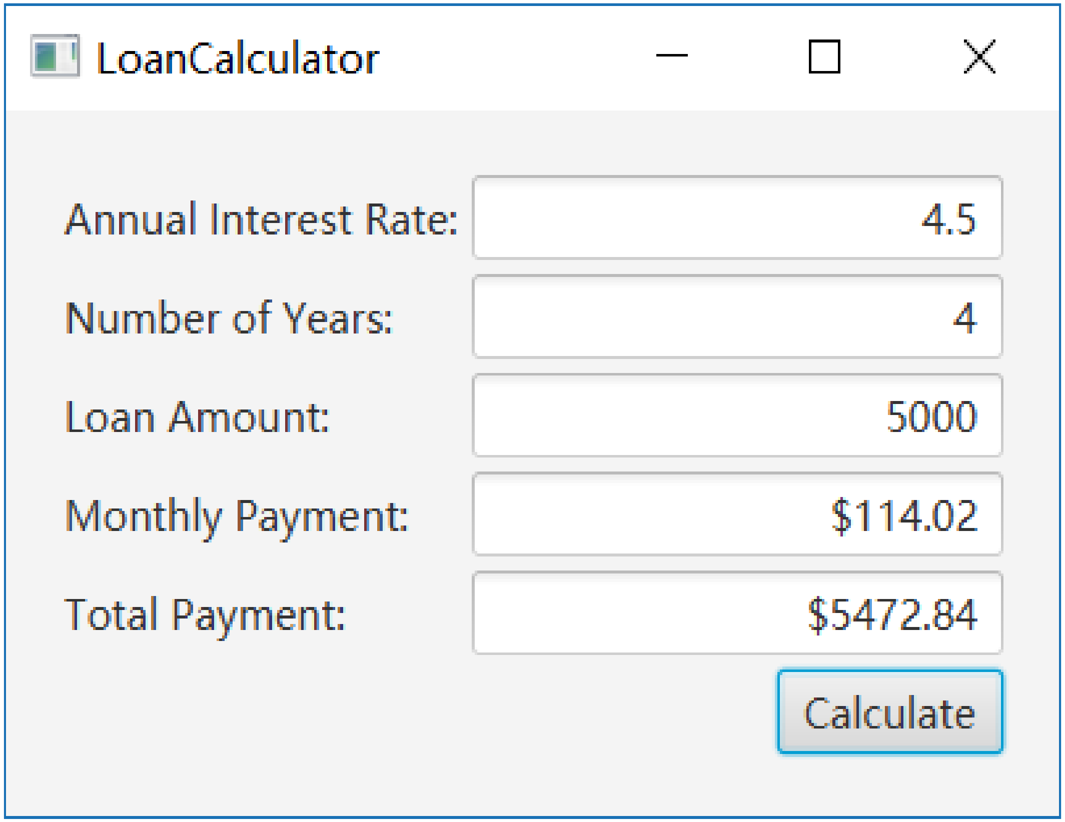
Chapter 12 Event-Driven Programming

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.



LoanCalculator

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

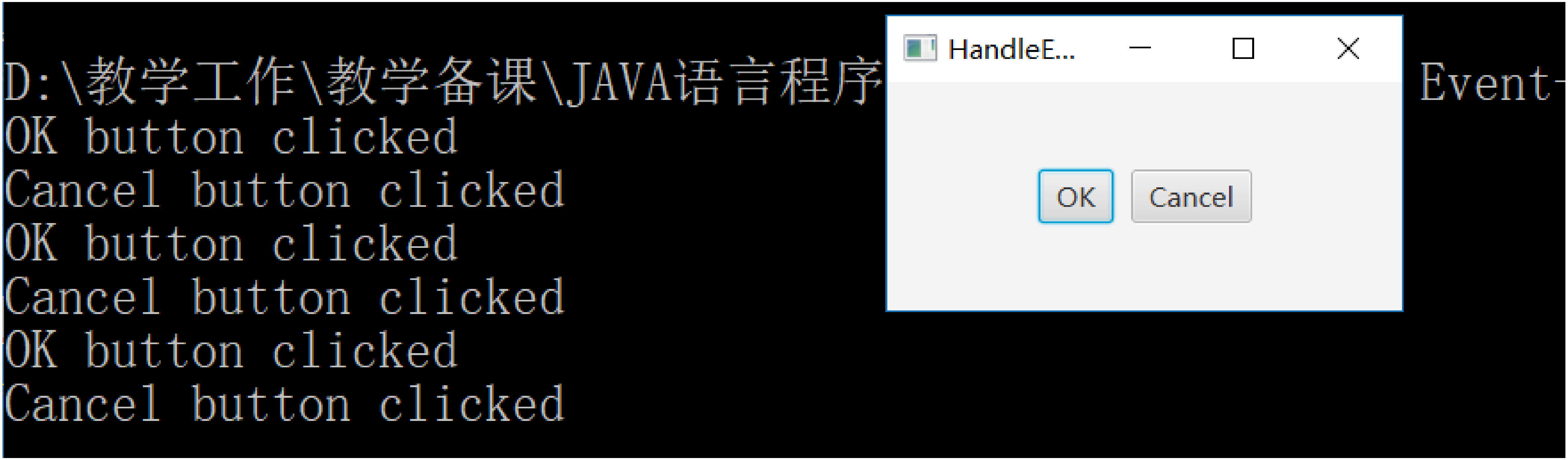
Procedural vs. Event-Driven Programming

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.

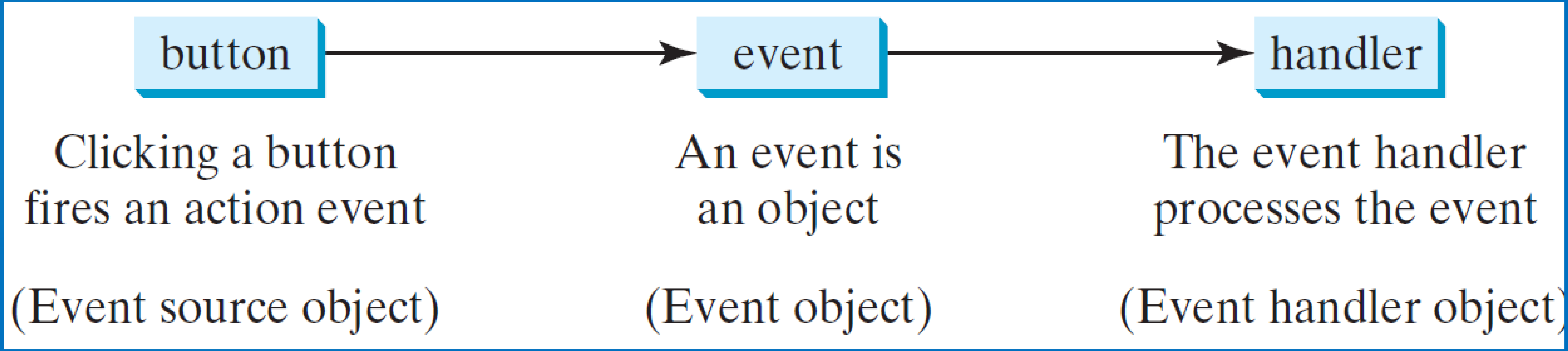


```
D:\教学工作\教学备课\JAVA语言程序                              Event-
OK button clicked
Cancel button clicked
OK button clicked
Cancel button clicked
OK button clicked
Cancel button clicked
```

Event-Source, Event and Event-Handler

Source object (e.g., button)

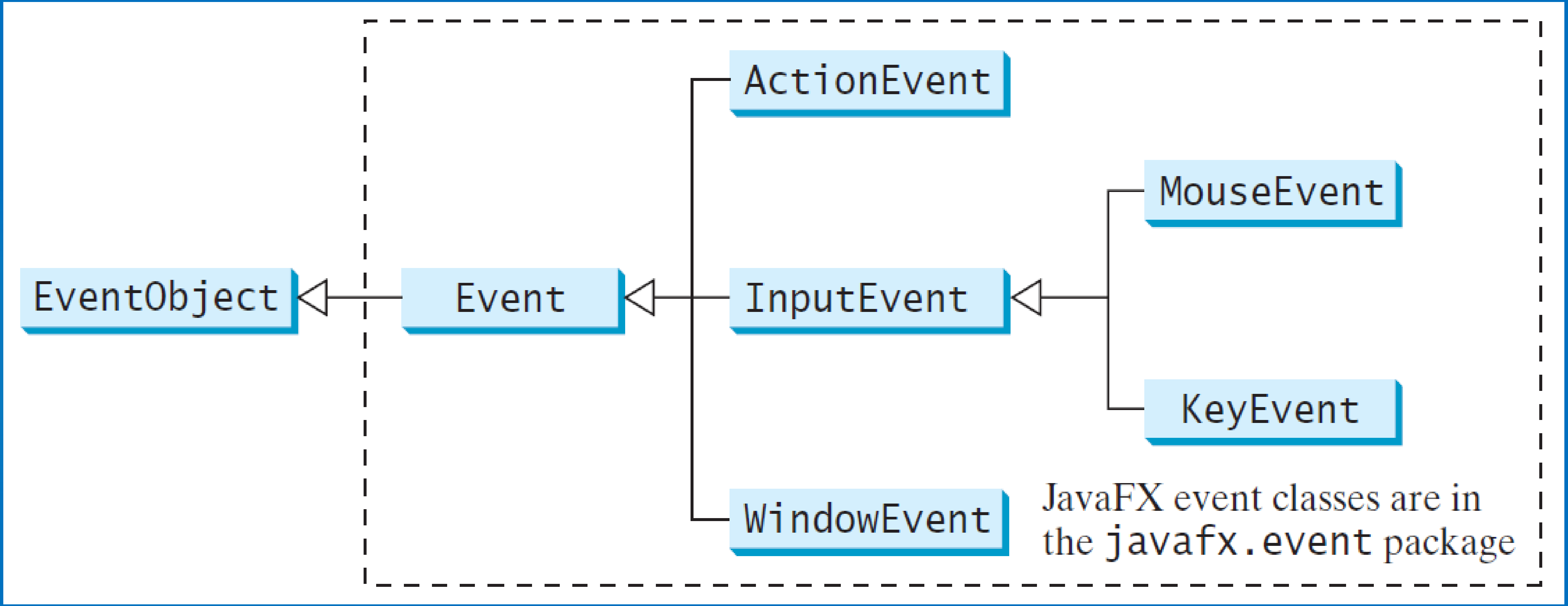
Listener object contains a method for processing the event.



Event

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

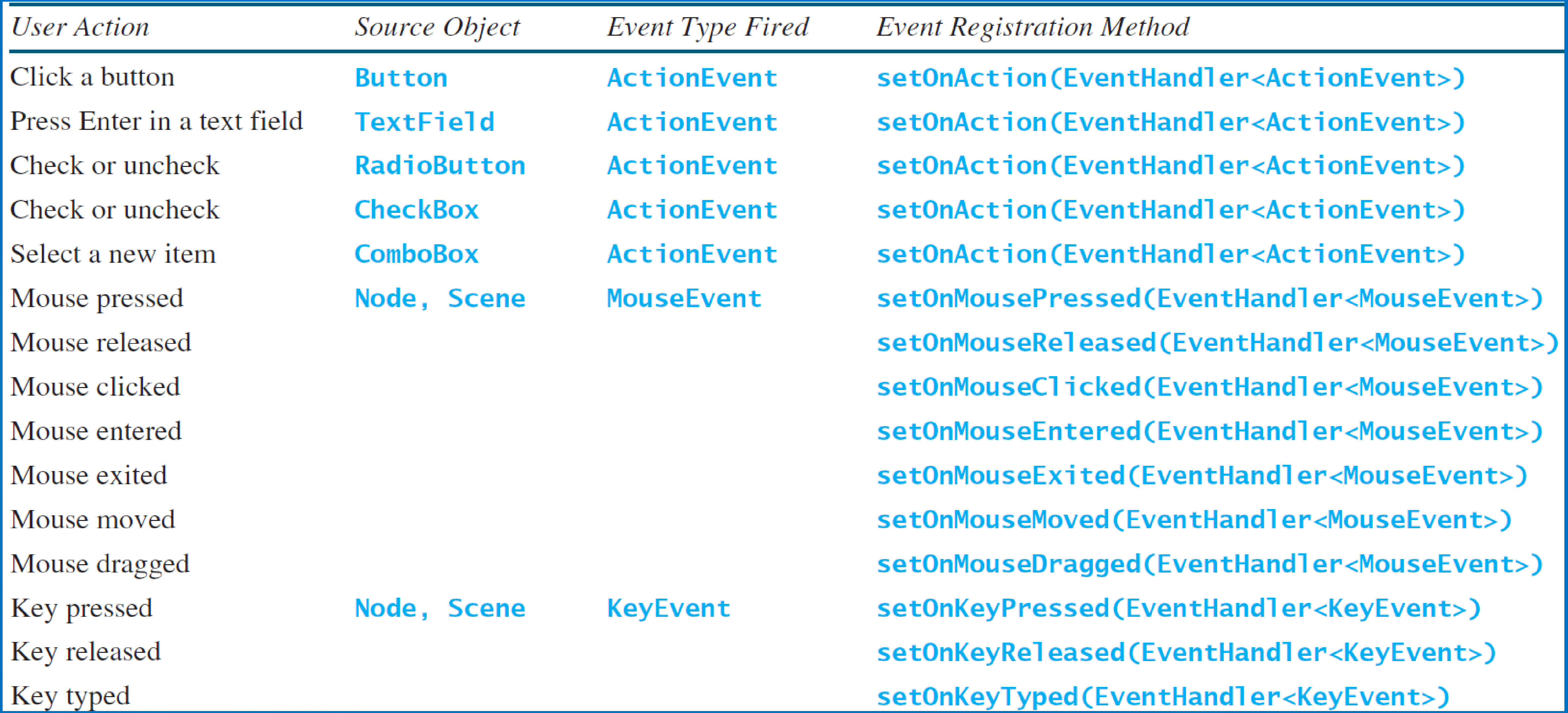


JavaFX event classes are in the javafx.event package

Event Information

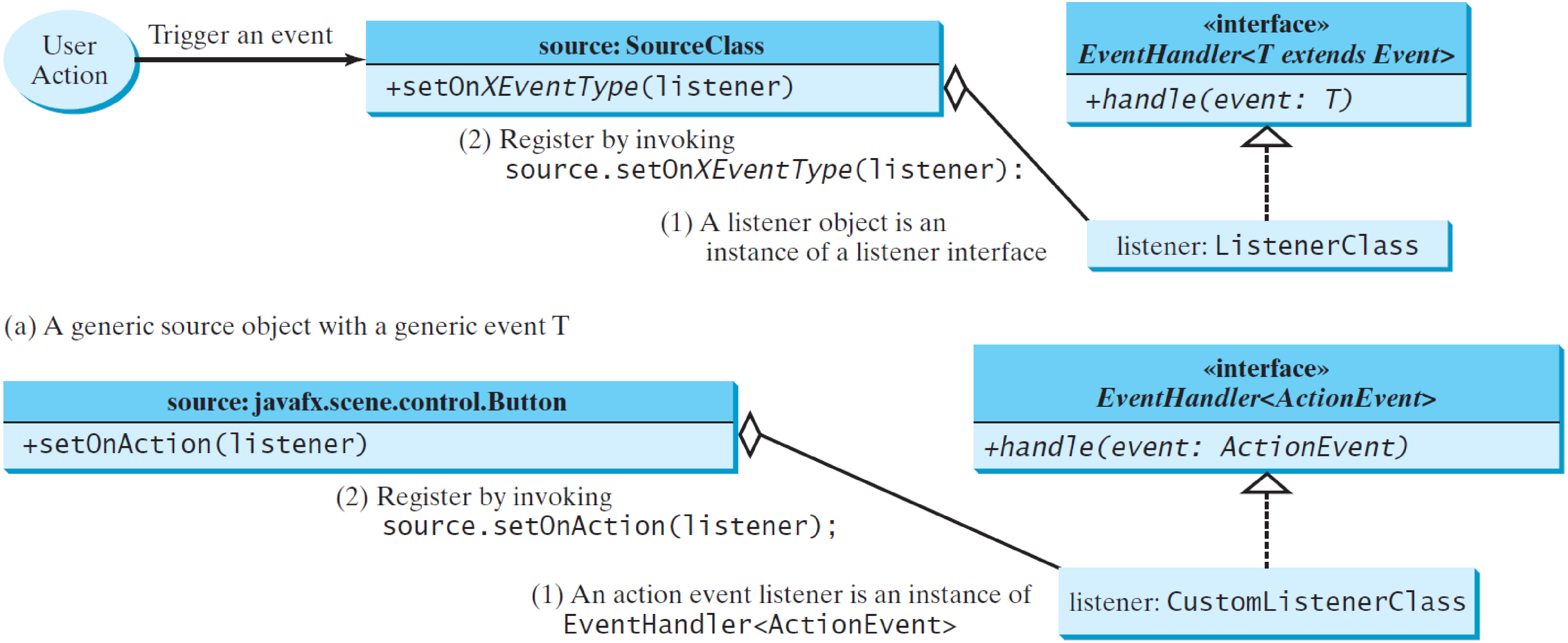
An event object contains whatever properties are pertinent to the event. You can *identify the source object* of the event using the `getSource()` instance method in the EventObject class. The subclasses of EventObject deal with special types of events, such as *button actions*, *window events*, *component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

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 Event Handling Model



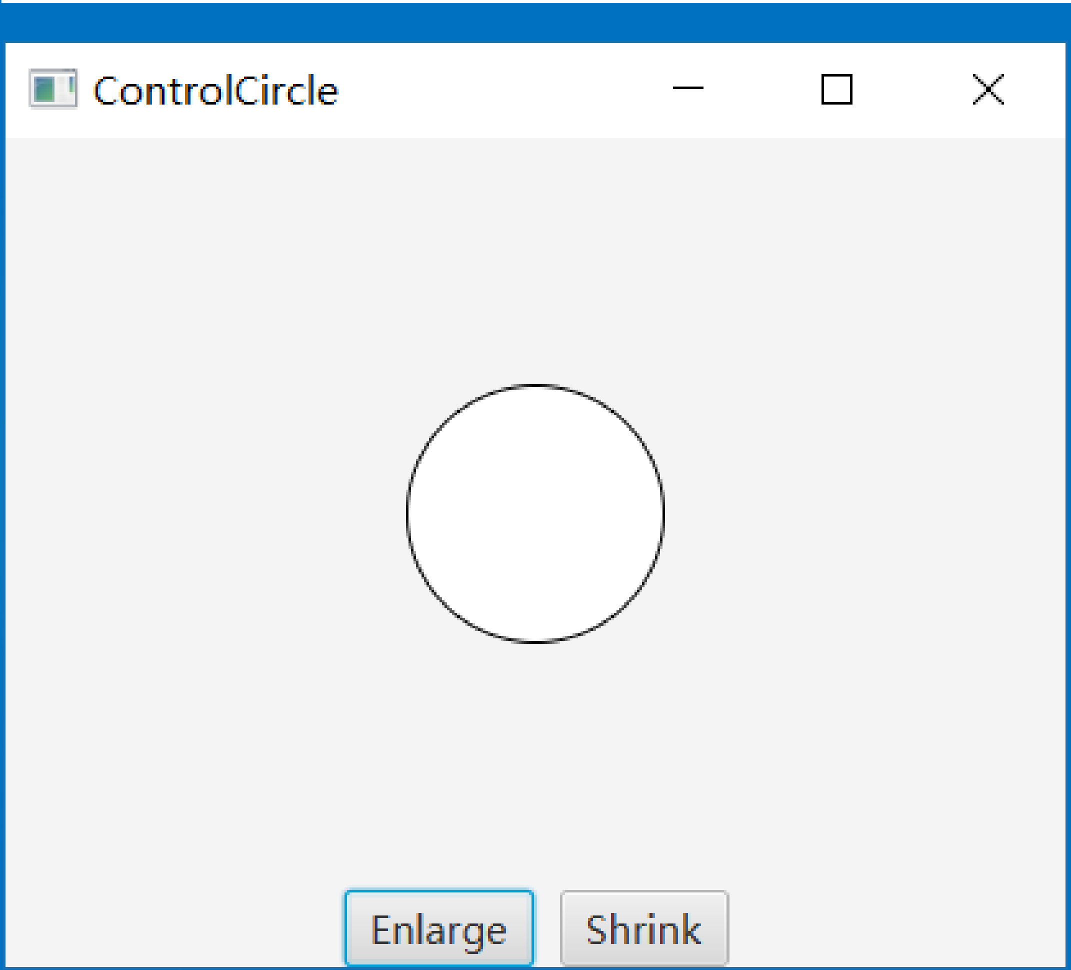
User Action
Trigger an event
source: SourceClass
+setOnXEventType(listener)
«interface»
EventHandler<T extends Event>
+handle(event: T)
(2) Register by invoking source.setOnXEventType(listener);
(1) A listener object is an instance of a listener interface
listener: ListenerClass
source: javafx.scene.control.Button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

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

(b) A Button source object with an ActionEvent

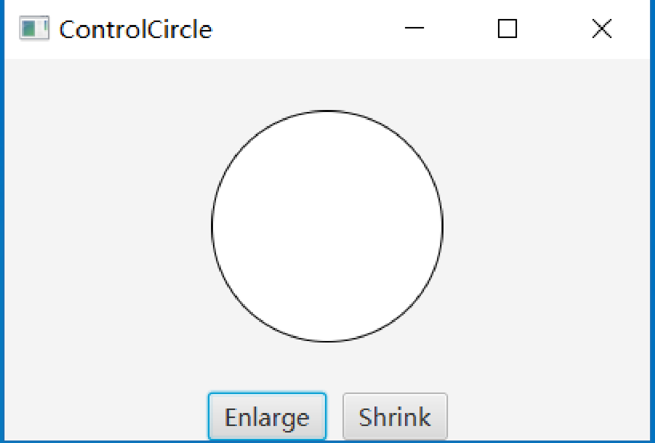
Example: First Version for ControlCircle (no listeners)

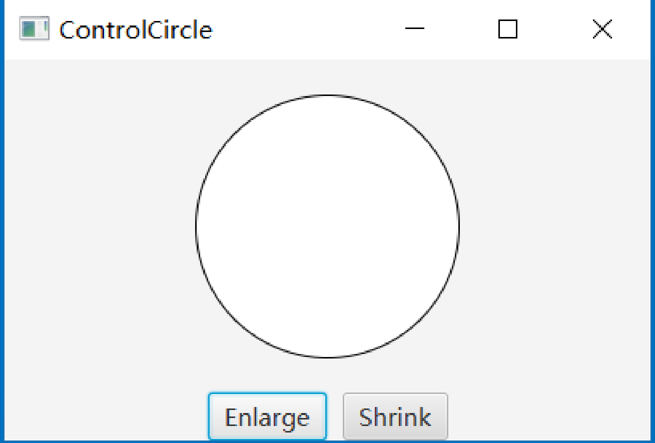
Now let us consider to write a program that uses two buttons to control the size of a circle.



Example: Second Version for ControlCircle (with listener)

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





Inner Class Listeners

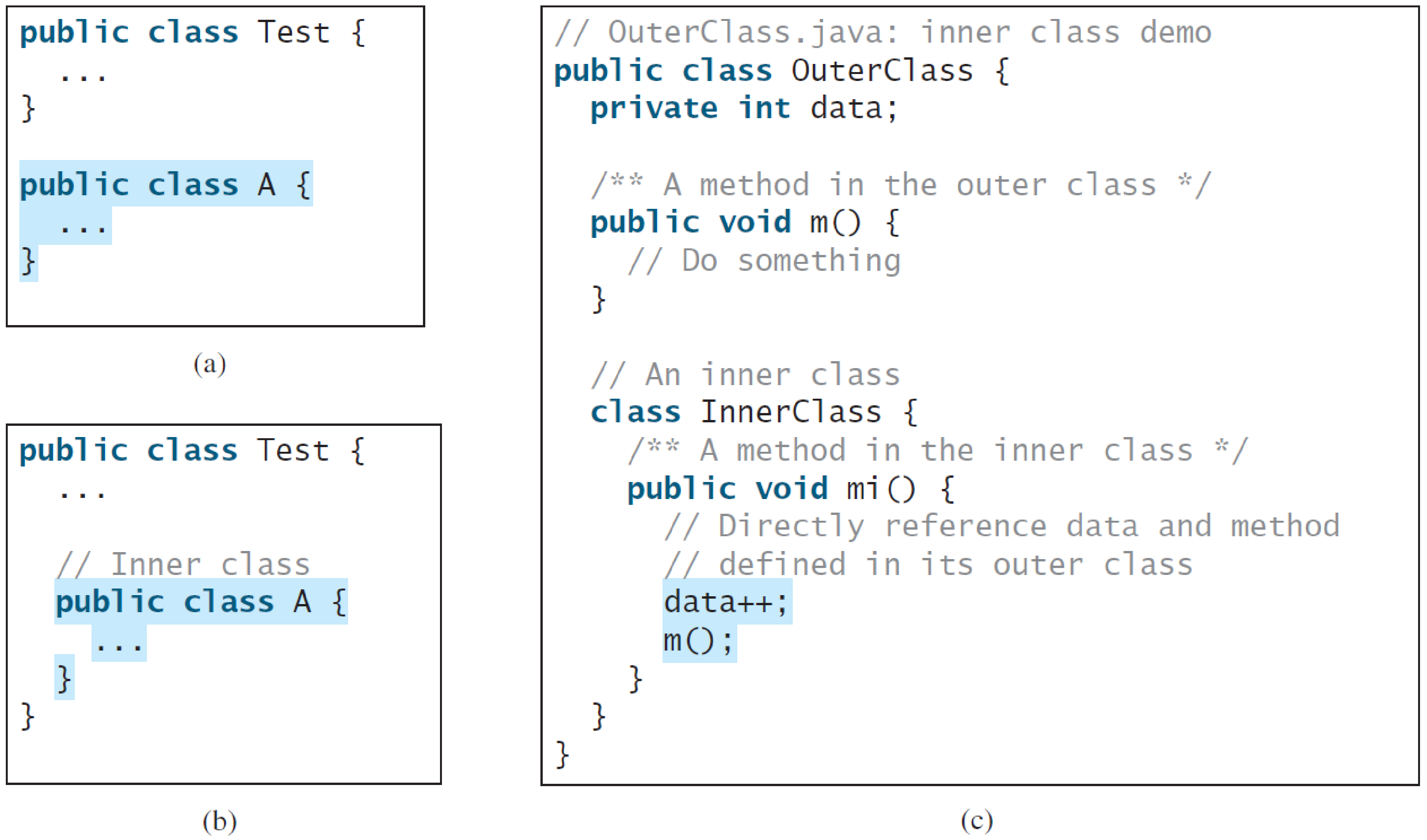
A listener class is designed specifically to create a listener object for a GUI component (e.g., a button). It will not be shared by other applications. So, it is appropriate to define the listener class inside the frame class as an inner class.

Inner 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*.

Inner Class



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

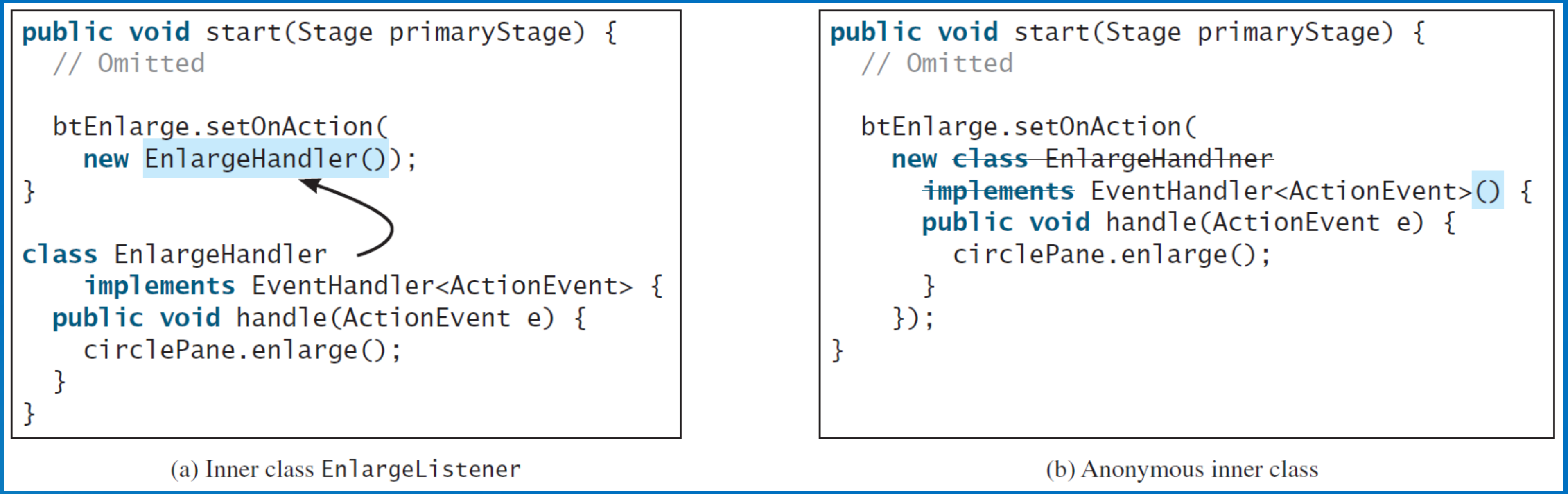
(c)

Anonymous Inner Classes

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



```
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

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

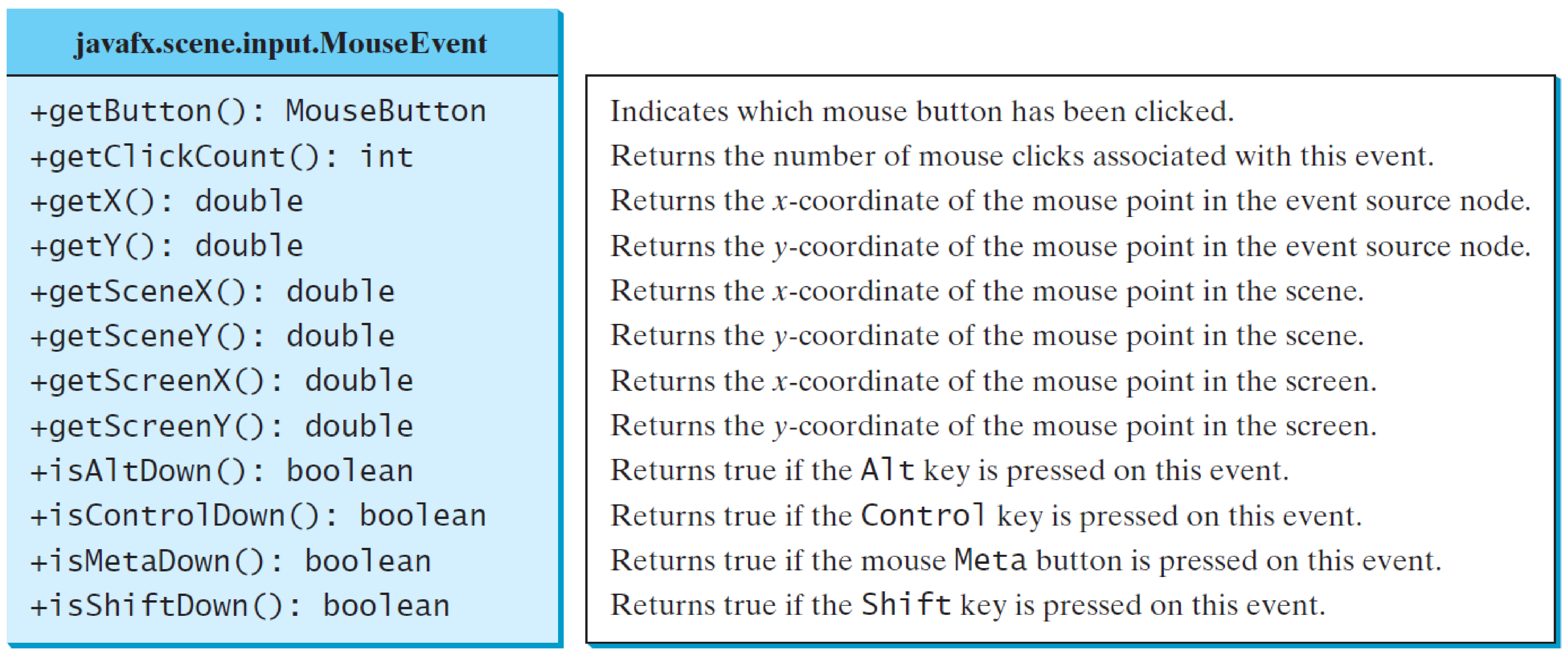
```
btEnlarge.setOnAction(e -> {
  // Code for processing event e
});
```

(b) Lambda expression event handler

Single Abstract Method Interface (SAM)

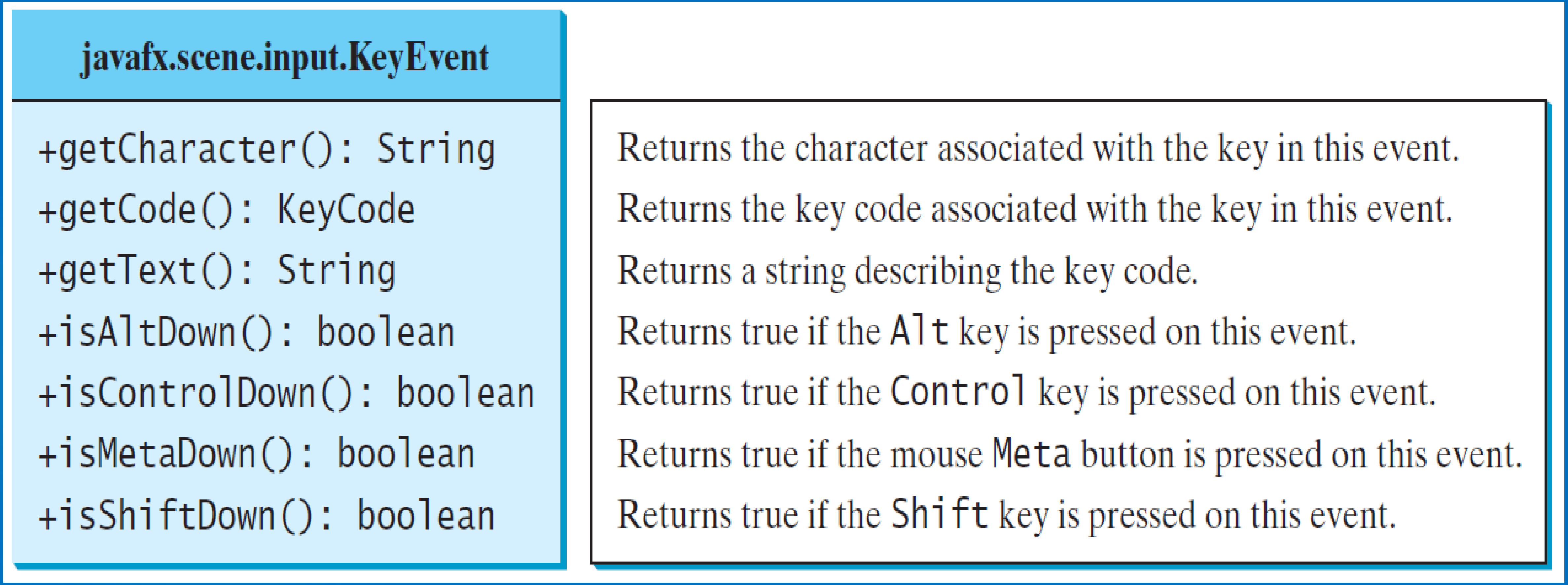
The statements in the lambda expression is all for that method. If it contains multiple methods, the compiler will not be able to compile the lambda expression. So, for the compiler to understand lambda expressions, *the interface must contain exactly one abstract method*. Such an interface is known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

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.

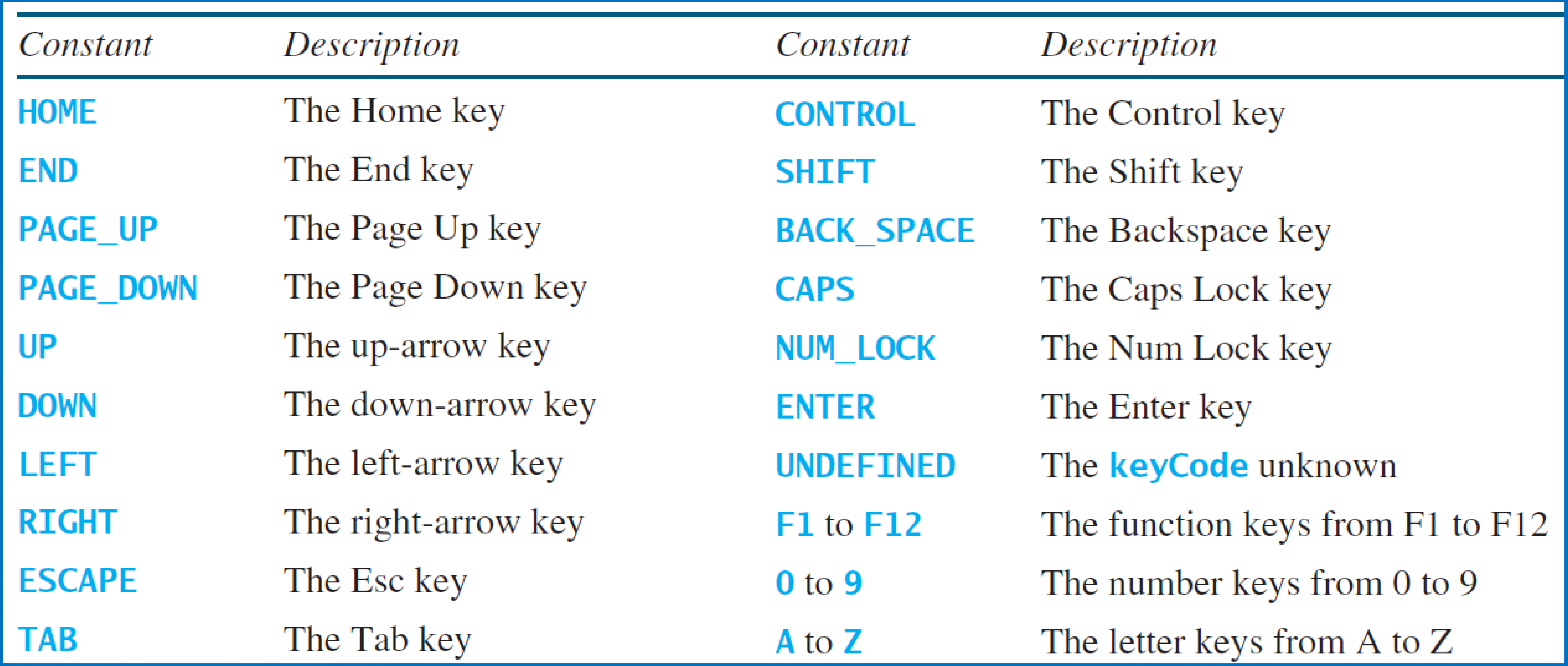
The KeyEvent Class



javafx.scene.input.KeyEvent

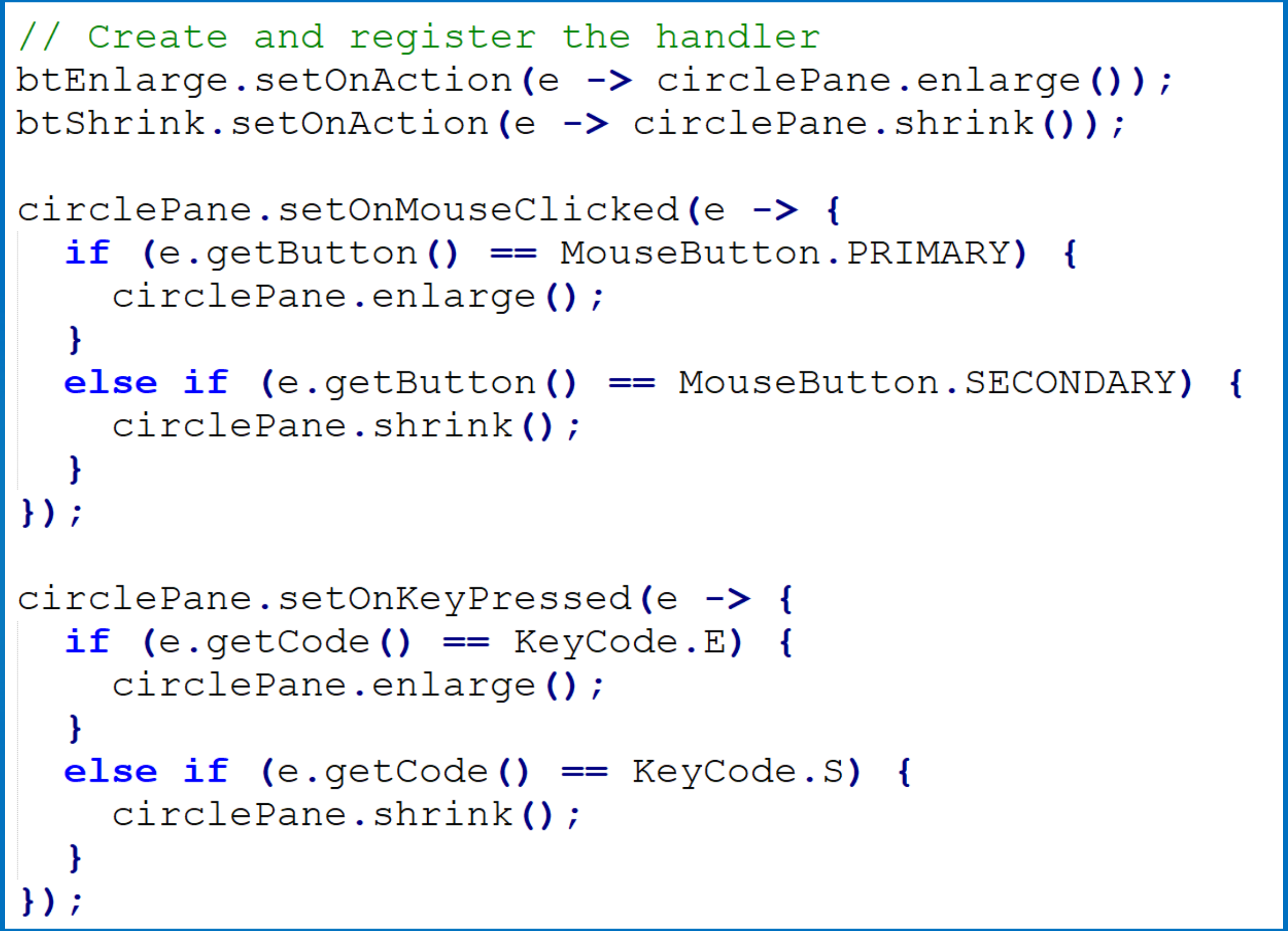
`+getCharacter(): String`	Returns the character associated with the key in this event.
`+getCode(): KeyCode`	Returns the key code associated with the key in this event.
`+getText(): String`	Returns a string describing the key code.
`+isAltDown(): boolean`	Returns true if the `Alt` key is pressed on this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

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

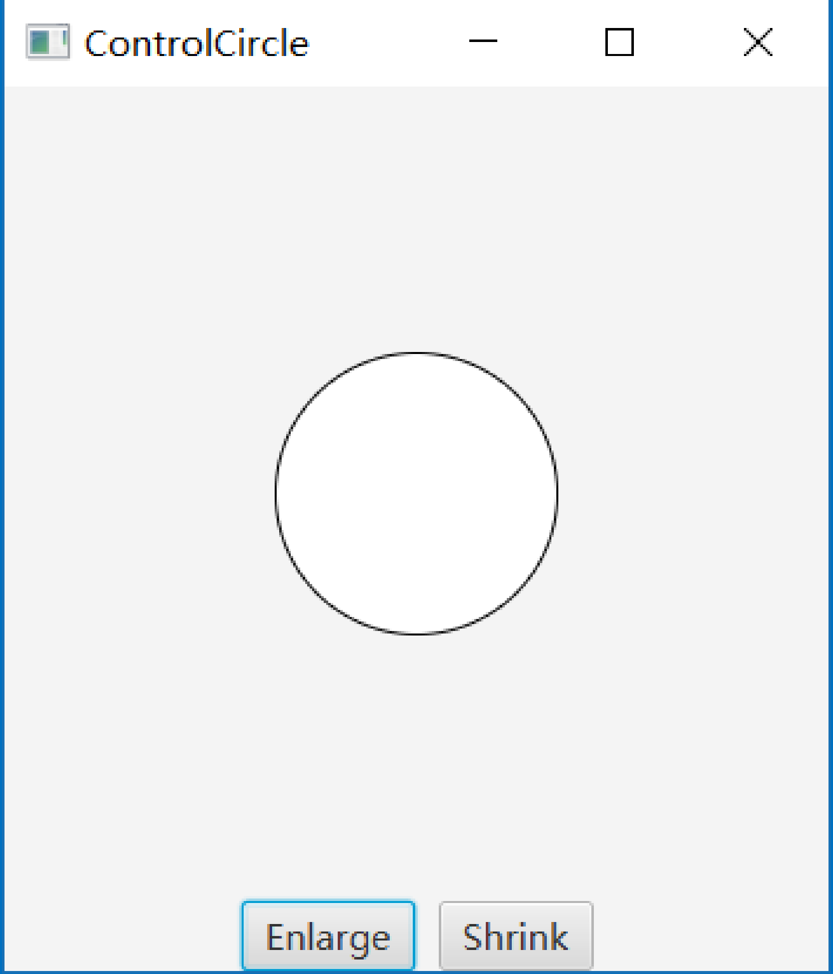
Example: Control Circle with Mouse and Key



```
// 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();
  }
});

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


ControlCircle
Enlarge
Shrink