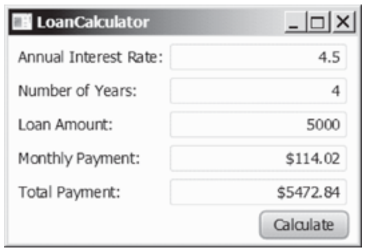
Note #8 – Event-Driven Programming

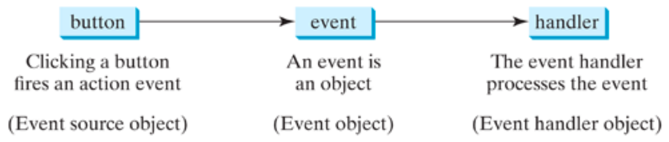
- **Event Handlers**

A GUI interacts with the users. You can write code to process **events** such as a button click, mouse movement, and keystrokes. Suppose you wish to write a GUI program that lets the user enter a loan amount, annual interest rate, and number of years then click the Calculate button to obtain the monthly payment and total payment. 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

To respond to a button click, you need to write the code to process the button-clicking action. The button is an event source object—where the action originates. You need to create an object capable of handling the action event on a button. This object is called an **event handler**.

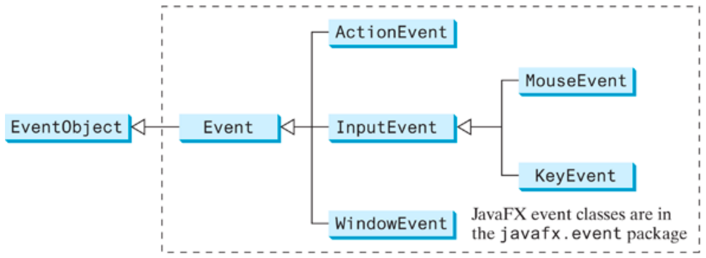


Not all objects can be the handlers for an action event. To be a handler of an action event, two requirements must be met:

1. The object must be an instance of the EventHandler interface. This interface defines the common behavior for all handlers.
2. The EventHandler object handler must be registered with the event source object using the method **source.setOnAction(handler).**

The EventHandler interface contains the **handle(ActionEvent)** method for processing the action event. Your handler class must override this method to respond to the event.

An **event** is an object created from an event source. Firing an event means to create an event and delegate the handler to handle the event. When you run a Java GUI program, the program interacts with the user and the events drive its execution. This is called event-driven programming. An event can be defined as a signal to the program that something has happened. **Events are triggered by external user actions**, such as mouse movements, mouse clicks, and keystrokes. The program can choose to respond to or ignore an event. The component that creates an event and fires it is called the **event source object**, or simply source object or source component. For example, a button is the source object for a button-clicking action event. An event is an instance of an event class. The root class of the Java event classes is **java.util.EventObject**. The root class of the JavaFX event classes is **javafx.event.Event**. The hierarchical relationships of some event classes are shown below.



You can identify the source object of an event using the **getSource()** instance method in the **EventObject** class. The subclasses of EventObject deal with specific types of events, such as action events, window events, mouse events, and key events. For example, **when clicking a button, the button creates and fires an ActionEvent**. Here, the button is an event source object, and an ActionEvent is the event object fired by the source object.

A handler is an object that must be registered with an event source object and it must be an instance of an appropriate event-handling interface.

- **Registering an Event**

Java uses a delegation-based model for event handling: A source object fires an event, and an object interested in the event handles it. The latter object is called an **event handler** or an **event listener**. For an object to be a handler for an event on a source object, two things are needed.

1. The handler object must be an instance of the corresponding event–handler interface to ensure the handler has the correct method for processing the event. JavaFX defines a unified handler interface EventHandler for an event T. The handler interface contains the **handle(T e)** method for processing the event. For example, the handler interface for ActionEvent is EventHandler; each handler for ActionEvent should implement the handle(ActionEvent e) method for processing an ActionEvent.
2. The handler object must be registered by the source object. Registration methods depend on the event type. For ActionEvent, the method is **setOnAction**. For a mouse-pressed event, the method is **setOnMousePressed**. For a key-pressed event, the method is **setOnKeyPressed**.

- **Anonymous Inner class and Lambda expression for event handler**

```
//anonymous inner class
  bt.setOnAction(new EventHandler<ActionEvent>() {
      @Override // Override the handle method
      public void handle(ActionEvent e) {
        bt.setText("Welcome World!!");
      }
  } );
//Lambda expression
bt.setOnAction(e-> {
  bt.setText("Welcome World!!");
} );
```

- **Mouse Event**

A MouseEvent is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene. The MouseEvent object captures the event, such as the number of clicks associated with it, the location (the x- and y-coordinates) of the mouse, or which mouse button was pressed.

https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/input/MouseEvent.html

- **Key Event**

A KeyEvent is fired whenever a key is pressed, released, or typed on a node or a scene. Key events enable the use of the keys to control and perform actions, or get input from the keyboard. The KeyEvent object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key.

https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/input/KeyEvent.html
key code: https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/input/KeyCode.html

- **Listeners**

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. The listener class must implement the functional interface InvalidationListener to override the invalidated(Observable o) method for handling the value change. Once the value is changed in the Observable object, the listener is notified by invoking its invalidated(Observable o) method. Every binding property is an instance of Observable.