COMP 3021 Fall 2021

Java Programming

Written By: Ljm

Topic 9 Event-Driven Programming

1 Define an Event Handler

Motivation: Up to now, our programs are always executed in a *procedural* order, but sometimes, we need certain codes to execute *upon activation of events*, such as button presses.

To handle GUI events, there are two components:

- **Event Source:** like button, keyboard, etc.
- **Event listener:** which *handles the event*, usually a method(known as **event handler**)

```
public void handle(ActionEvent e) {
    // what to perform when event occurs
}
```

There are two steps to do:

1. write a handler.
2. create a instance of the class **implements this handler**, and **register** this instance to listen to the **event source**.
3. then java will automatically call the handler when the event occurs.

Example. The button is defined as follow:

```
var btOK = new Button("OK");
var okHandler = new OKHandlerClass();
btOK.setOnAction(okHandler);
```

And the handler class is:

```
class OKHandlerClass implements EventHandler<ActionEvent> {
    @Override
```

```
3       public void handle(ActionEvent e) {
4           System.out.println("OK button clicked");
5       }
6   }
```

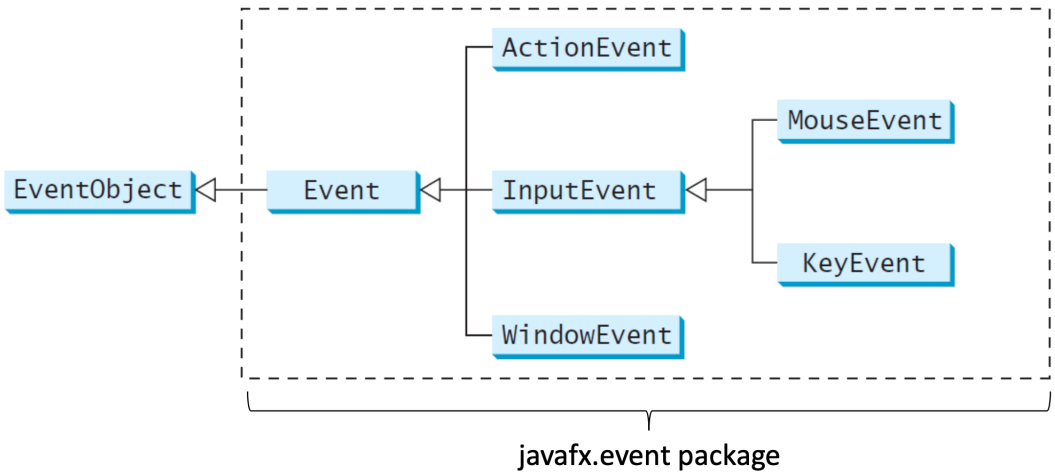
When the button "OK" is pressed, the program prints "OK button clicked" on the console.

There are lots of commonly used **Actions, Events** 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>)

For example, if we want to listen to and handle a **mouse pressed** event, we need to define a class that `implements EventHandler<MouseEvent>`, and **override** the `handle()` method inside.

Below are some commonly-used Events and there hierarchy.



The `EventHandler<T extends Event>` interface accepts type **T** that is a subtype of **Event** class.

2 Inner Class

Motivation: Mostly, an event handler is *exclusively owned by an application*, which means it's *improper to be accessible by other applications*. Thus **inner class** is introduced.

An **inner class** is a **non-static** class defined inside another class. **Note:** we can only declare `protected`, `private` class in inner class. When we define an inner class as "private", it means *only its outer class can access it*, even the class in same package *cannot*.

```
public class MyFXApplication extends Application {
    private int data = 3021;
    @Override
    public void start(Stage primaryStage) {
        var btOK = new Button("OK");
        btOK.setOnAction(new MyListener());
        // ...
        System.out.println("i = " + new MyListener().i);    // can access var in inner class
    }
    // Inner class, exclusively owned by MyFXApplication
    private class MyListener implements EventHandler<ActionEvent> {
        private int i = 0;
        @Override
        public void handle(ActionEvent event) {
            i = data;     // can access to var in outer class
            System.out.println(i);
        }
    }
}
```

Notice when **outer class** access to instance variables in **inner class**, we need to create an instance of inner class first, but when **inner class** access to instance variables in **outer class**, there is no need to do so, since one outer class instance *must have been created before* the inner class exists, and in this situation, *Inner class uses the* ***current snapshot*** *of the Outer class.*

3 Accessability of Inner Class

Inner Class and **Outer Class** are "friends" *in both directions*, which means they can access to the other's private members.

[*Compare with C++*: The "friend" mechanism in C++ breaks the rule of encapsulation, and lots of "friend" relationships make it messy for other team members to maintain the code.]