Unit 11:

GUI 2 - Event-Driven Programming

Object-Oriented Programming (OOP)
CCIT 4023, 2025-2026

U11: GUI 2 - Event-Driven Programming

- Event Handling (User Input Action Handling)
  - Event Source and Event Listener

Event Handling
(User Input Action Handling)

- Apart from composing components with proper layout, developing Graphical User Interface (GUI) applications requires us to write codes responding to user actions
- An action (of user) involving a GUI object in Java, such as clicking a button, is called an **event**
- The mechanism to process events is called **event handling**
- Event handling in Java is implemented with two types of objects, based on a delegation event model:
  - **event source** objects
  - **event listener** objects (or even handler objects)

Event Handling
(User Input Action Handling)

- The *delegation event model* is based on the idea that:
  - the event source object generates events (e.g. responding to user input), but does not handle them
  - the event source object delegates the another one (a registered event listener or handler) to handle their generated events, for specific tasks and applications
- There are different types of events in Java, and their corresponding event sources and listeners, e.g.
  - Event source `JButton` creates event type `ActionEvent`, which could be handled by event listener `ActionListener`
  - Event source `JComboBox` creates event type `ItemEvent`, which could be handled by event listener `ItemListener`

Event Source

- An **event source** is a GUI object where an event occurs, and we say an event source *generates* events
- Essentially event sources are GUI components for user inputs. E.g.
  - Buttons, text boxes, checkboxes, and menus are common event sources in GUI-based applications.
- A set of very useful and common event sources (e.g. `JButton`) predefined in standard API packages
- When programming GUI-based applications, developers add some predefined GUI event source components for user inputs
  - Although possible, we rarely define our own event sources when writing GUI-based applications under normal circumstances
  - In delegation event model, predefined GUI event sources are not handling their generated events responding to user actions for specific application. Instead, developers of this specific application would create event listeners to handle these events to achieve specific tasks.

Event Listener

- An **event listener** object is an object that includes a method that gets executed in response to some generated events
  - Event listeners typically are defined by developers, to attach operations responding to particular user actions for specific tasks, e.g.
    - Responding to user-click action of a specific event source `JButton` for opening file, developer creates an event listener with a corresponding method to open a file, etc.
  - In Java, typically different listeners implement *interfaces* from a set of predefined listeners in the package `java.awt.event`.
    - E.g. Button-click action event is handled by an object which implements `java.awt.event.`**`ActionListener`**.
- A *listener* object must be associated, linked, or *registered* to a *source* object first, so it can be notified when the source generates events; otherwise they are isolated objects.

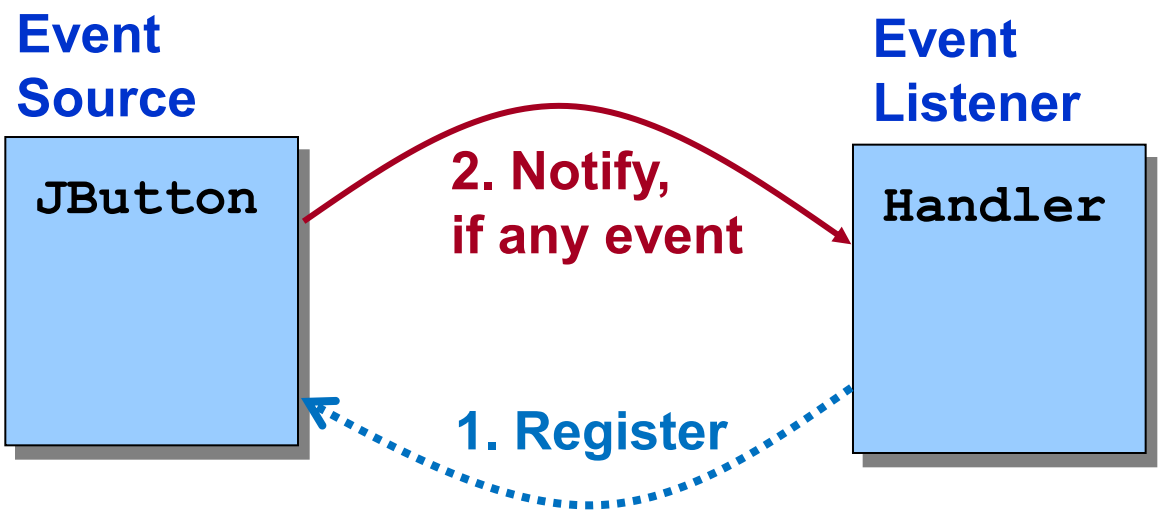
Connecting Event Source and Listener

1. A listener must be registered to an event source, e.g.

```
aButton.addActionListener(aHandler);
```

2. Once registered, listener will get notified when the event source generates events, e.g. defining a method which is called if an event comes

```
public void actionPerformed(ActionEvent event) {
//..
}
```

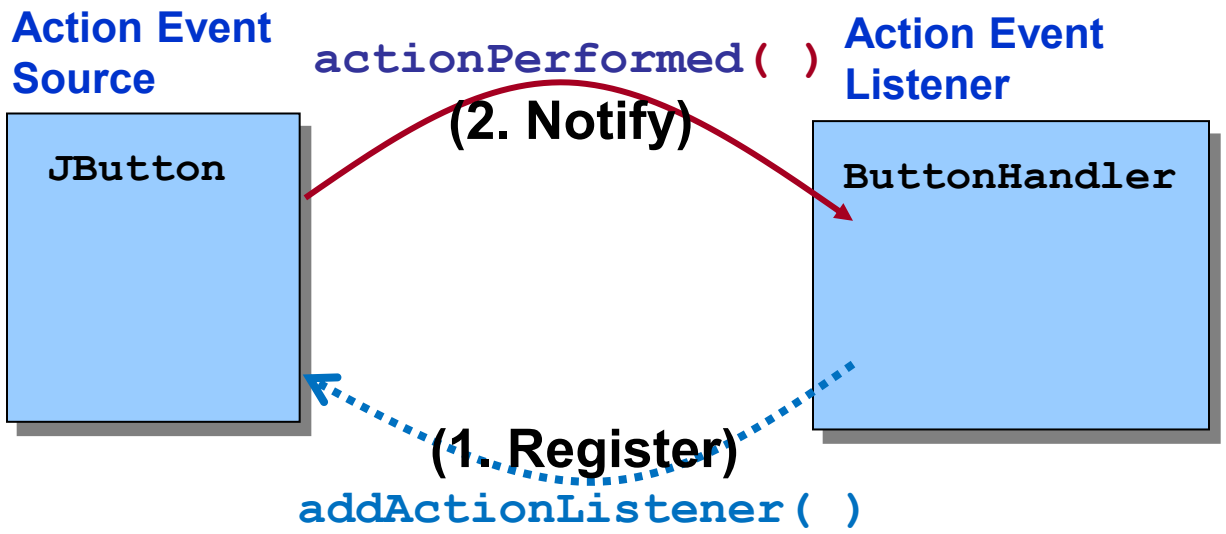


Event Types

- Registration and notification are specific to event types
  - `ActionListener` handles action events `ActionEvent`
  - `ItemListener` handles item selection events `ItemEvent`
  - `MouseListener` handles mouse events `MouseEvent`
  - etc.
- Among different types of events, the action event is the most common, e.g.
  - Clicking on a button generates an action event
  - Selecting a menu item generates an action event
- Action events are generated by action event sources and handled by action event listeners

Handling Action Events

`SubJFrameWithButtonsAndHandler.java` `ButtonHandler.java`



* Register an object of `ButtonHandler` with `addActionListener()` method of the source, as the action listener of the button (suppose we have a defined class `ButtonHandler` for handling Button Event:

```
JButton abutton = new JButton("Press Me!");
ButtonHandler handler = new ButtonHandler( );
abutton.addActionListener(handler);  // 1. register
```

Java Interface (Recall)

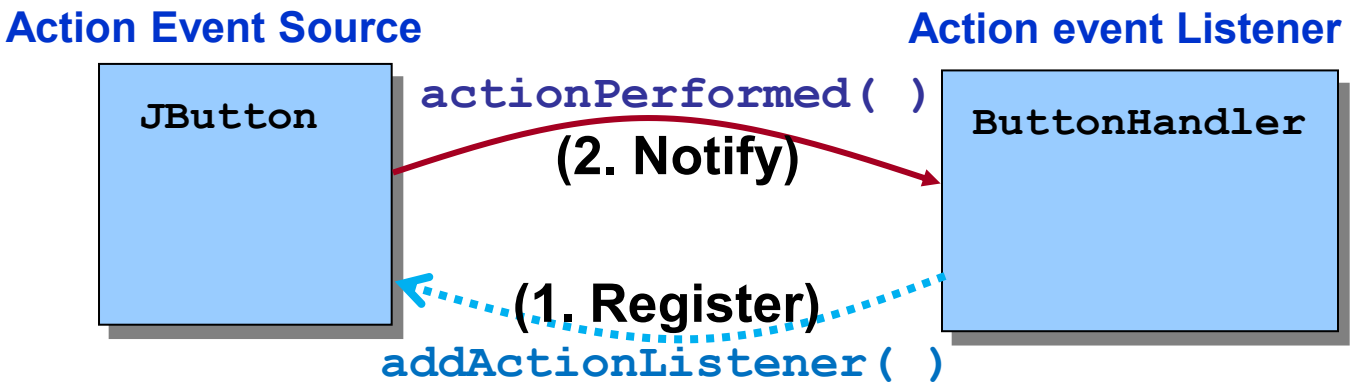
- A Java ***interface*** is a reference type that can *only* contain constants, and method signatures
  - Method signatures have no method bodies. (Methods in an *interface* are *implicitly* public abstract, and `public abstract` modifiers may be omitted)

    *** Newer Java version is a bit different.*

- Typically an interface is a group of related methods without method bodies
  - These methods define certain behaviors the related classes (and their objects) promises to provide for interfacing with outside world.
- To use an interface, the related classes **`implements`** the interface

Defining the Handler
(by Implementing `ActionListener` Interface)

- The `ActionListener` interface includes one method called:
  `actionPerformed (ActionEvent e)`
- Since `actionPerformed()` is the method that will be called when an action event is generated, this is the place where we put a code we want to be executed in response to the generated events
  - In previous case, `ButtonHandler` should implement method `actionPerformed()` for handling Button event

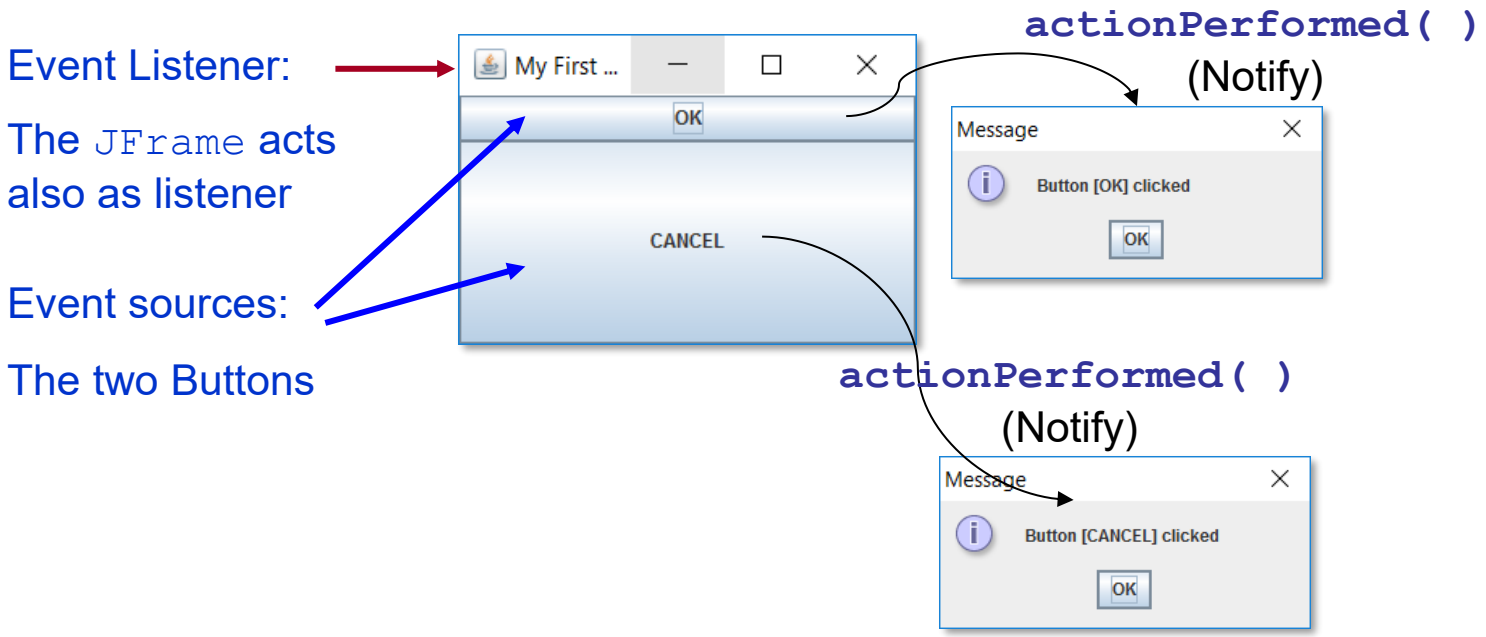


Refer to: https://docs.oracle.com/en/java/javase/25/docs/api/java.desktop/java/awt/event/ActionListener.html

`JFrame` as Event Listener

- Instead of defining a separate event listener such as **`ButtonHandler`**, we can define an object that contains the event sources be a listener, such as the `JFrame`

  E.g. The frame acts as the listener of the action events of the buttons it contains



Combination Version

```
// Also make this SubJFrame class act as a listener
public class SubJFrameWithALL extends JFrame
                                implements ActionListener {
    // ...
      okButton.addActionListener(this);
      cancelButton.addActionListener(this);
    }
    public void actionPerformed(ActionEvent event) {
// Here, this class implements the interface method
// ..
    }
}
```

Body of method `actionPerformed()`

Determine the Event Source

- It is possible that an event listener is added by many event sources
  - May get the event source of a specific event, with the method of event **`getSource()`**
- For example,

```
public void actionPerformed(ActionEvent event) {
        if (event.getSource() == okButton) {
// define actions after pressing a specific button
        }
        else {          // other event sources
                // ...
        }
}
```

Body of method `actionPerformed()`

Functional Interfaces with Lambda Expressions

- ***Functional Interface***: interface contains only one abstract method.
  - E.g. Interface `ActionListener` has only one abstract method: `actionPerformed()`

- ***Lambda Expression*** is convenient and useful, for functional interface, where we can define and create an object implementing the function interface in one portion. E.g.

```
// register a listener (with Lambda Expression) to a Button
    okButton.addActionListener( event -> {
// Below, implements body of interface method: actionPerformed
       if (event.getSource() == okButton) {
       // define actions after pressing a specific button
       }
       else {          // other event sources
              // ...
       }
    });
```

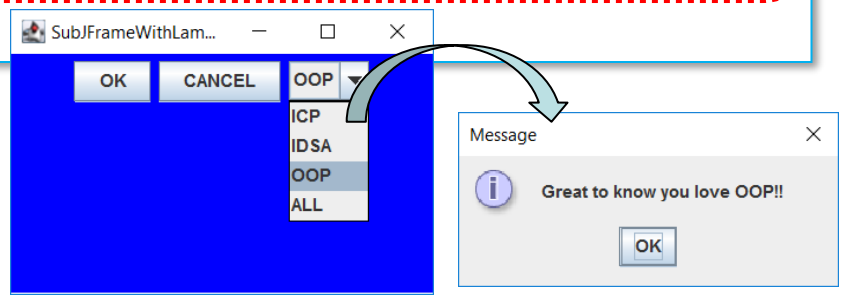
Body of method `actionPerformed()`, of listener `ActionListener`

Functional Interfaces with Lambda Expressions

- Another example of adding a item event source `JComboBox` with Lambda Expression for the item event listener `ItemListener`, a functional interface with only one method `itemStateChanged(ItemEvent e)`:

```
// Create and Add a JComboBox
String [] items = {"ICP", "IDSA", "OOP", "ALL"};
JComboBox <String> comboBox = new JComboBox<String> (items);
contentPane.add(comboBox); // Add the JComboBox
// register a listener (with Lambda Expression) to a JComboBox
comboBox.addItemListener( ie -> {
// Here implements the interface method itemStateChanged():
    if(ie.getStateChange() == ItemEvent.SELECTED)
        JOptionPane.showMessageDialog(null, // show simple info.
            "Great to know you love "
            + comboBox.getSelectedItem()
            +  "!!");
});
```

Body of method `itemStateChanged()`, of listener `ItemListener`



Defining a `ButtonHandler` Class

- The class that implements the *`ActionListener`* interface must provide the method body of *`actionPerformed()`*.

ButtonHandler.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;  // for ActionEvent, ActionListener

public class ButtonHandler implements ActionListener {
    //2.notify/call the below method, if any coming action/button event
    public void actionPerformed(ActionEvent event) {
    // retrieve the text of the event source
        JButton clickedButton = (JButton) event.getSource();
        // find the frame that contains the event source
        JRootPane rootPane = clickedButton.getRootPane( );
        Frame frame = (JFrame) rootPane.getParent();
        // update the text of the frame title
        frame.setTitle("You clicked " + clickedButton.getText());
    }
}
```

Register `ActionListener` of Buttons

SubJFrameWithButtonsAndHandler.java

```
import javax.swing.*;
import java.awt.*;
public class SubJFrameWithButtonsAndHandler extends JFrame {
  // ...
  public SubJFrameWithButtonsAndHandler( ) {
        // ...
  }
// Create a method to initialize the GUI windows and components
  private void initGUI() {
// ...
        okButton = new JButton("OK");
        cancelButton = new JButton("CANCEL");
        contentPane.add(okButton, BorderLayout.NORTH);
        contentPane.add(cancelButton, BorderLayout.CENTER);

// register a ButtonHandler object as action listener of 2 buttons
        ButtonHandler handler = new ButtonHandler();
        okButton.addActionListener(handler);
        cancelButton.addActionListener(handler);

  }
}
```

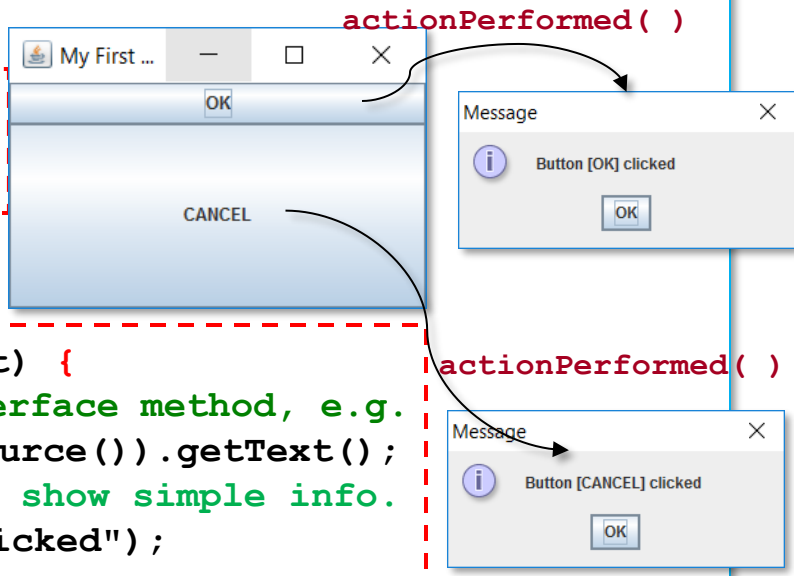
Register `this` as `ActionListener`

SubJFrameWithALL.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*; // for ActionEvent, ActionListener
public class SubJFrameWithALL extends JFrame implements ActionListener {
   // ...
// Create a method to initialize the GUI windows and components
   private void initGUI() {
// ...
         okButton = new JButton("OK");
         cancelButton = new JButton("CANCEL");
         contentPane.add(okButton);
         contentPane.add(cancelButton);
// register this as the action listener
         okButton.addActionListener(this);
         cancelButton.addActionListener(this);

   }

   public void actionPerformed(ActionEvent event) {
         // Here, this class implements the interface method, e.g.
         String butStr = ((JButton) event.getSource()).getText();
         JOptionPane.showMessageDialog(null, // show simple info.
                  " Button [" + butStr +  "] clicked");
   }
}
```



Nested Classes

- Java allows us to define a class within another class, called a *nested class*. A nested class is a member of its enclosing class, e.g.

```
public class OuterClass { // outer enclosing
class
    // ...
    class NestedClass {// inner nested class
        // ...
    }
}
```

- Nested classes enable us to
  - logically group classes that are only used in one place
  - increase the use of encapsulation, and
  - create more readable and maintainable code

Nested Classes

- Nested classes include two categories:
  - *Static Nested Class* (with keyword `static`), which does not have access to other members of the enclosing class.
  - *Non-Static Nested Class* (or simply *Inner Class*), which has access to other members of the enclosing class.
    - As with instance methods and fields/variables, inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields
    - two special kinds of inner classes:
      - *Local Class* (defined within a block, such as a method), defined in a *block*, which is a group of zero or more statements between balanced braces, typically in a method.
      - *Anonymous class* (without a class name), like local classes except that they do not have a name.

Anonymous Class

- There is a specific inner local class, which has no name
  - *Anonymous Class* is used to define an inner class AND create an object / instance of that class in one single step
  - Anonymous class will be compiled into a bytecode class file, with the file name of `<NameofEnclosingClass_$_AnInteger>`.
- Anonymous class lets us make program code much concise
  - *Declare and instantiate* a class at the same time
  - Use them if using an inner class only once

  * It is common to use anonymous class approach in GUI event handling, with different *event listener interfaces*
  - General form:

```
// define anonymous class (extends ASuperClass) & create its object
    new ASuperClass(){ //... Implement or override methods in
    superclass
    }
// define anonymous class (implements AInterface) & create its object
    new AInterface(){ //... Implement methods in interface
    }
```

Anonymous Class & Lambda Expression
(for Implementing Functional Interfaces)

- Very often the interface methods associated to individual GUI object is unique, and thus, the implemented interface methods are used *only once* for the specific GUI object.
  - Furthermore, many interfaces for GUI objects have only one abstract method. Such interface is known as “functional interface”, an interface that contains only one abstract method
  - However, a functional interface may contain one or more default methods or static methods.
- To make our code more concise, different features of coding methods are provided in newer Java, including *Anonymous class* and *Lambda Expression*
  - These approaches are useful if we register and apply such interface *only once*, as we would not refer the associated listener object back

Action Listener: Anonymous Class

* Anonymous Class is convenient and useful, if we register and apply the interface (e.g. event listener) *only once*

SubJFrameWithAnonymous.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SubJFrameWithAnonymous extends JFrame {
   // ...
// Method to initialize the GUI windows and components
  private void initGUI() {
// ...
    contentPane.add(okButton);
    contentPane.add(cancelButton);

// registering a new listener object (of an anonymous class) to okButton
// Note: this listener cannot be reused elsewhere as there's no reference name
    okButton.addActionListener ( new ActionListener() {
      public void actionPerformed(ActionEvent event) {
         // Here, this class implements the interface method, e.g.
         String butStr = ((JButton) event.getSource()).getText();
          JOptionPane.showMessageDialog(null, // show simple info.
                  " Button [" + butStr +  "] clicked");
      }
    });
// ...
}
```

Action Listener: Anonymous Class

The complete code sample

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SubJFrameWithAnonymous extends JFrame {
    Container contentPane = getContentPane();
    private JButton okButton;
    private JButton cancelButton;
    public SubJFrameWithAnonymous ( ) { // constructor, to initialize the JFrame
        setTitle ( "SubJFrameWithAnonymous" );
        setSize ( 300, 200 );
        setLocation ( 150, 250 );
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        initGUI ();
    }
    private void initGUI() { // Method to initialize the GUI windows and components
        contentPane.setBackground(Color.BLUE);
        contentPane.setLayout(new FlowLayout());
        okButton = new JButton("OK");
        cancelButton = new JButton("CANCEL");
        contentPane.add(okButton);
        contentPane.add(cancelButton);
        okButton.addActionListener(new ActionListener() {// registering a new listener object (of an anonymous class)
            public void actionPerformed(ActionEvent event) { // Here, this class implements the interface method
                String butStr = ((JButton) event.getSource()).getText();
             JOptionPane.showMessageDialog(null, // show simple info.
                                    " Button [" + butStr +  "] clicked");
            }
        });
    }
    public static void main(String[] args) { // main() method to start app, with a new Jframe shown
                        SubJFrameWithAnonymous frame = new SubJFrameWithAnonymous();
                        frame.setVisible(true);
    }
}
```

Action Listener: Lambda Expression

* Lambda Expression is convenient and useful, only for functional interface (only one abstract method, e.g. `ActionListener` has only one abstract method `actionPerformed(ActionEvent e)`

SubJFrameWithLambda.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SubJFrameWithLambda extends JFrame {
  // ...
  private void addButton() {
    okButton = new JButton("OK");
    cancelButton = new JButton("CANCEL");
    contentPane.add(okButton);
    contentPane.add(cancelButton);

// registering a new listener object (with Lambda Expression) to a Button
// Note: this listener cannot be reused elsewhere as there's no reference name
    cancelButton.addActionListener( e -> {
  // Here, this class implements the interface method, e.g.
      String butStr = ((JButton) e.getSource()).getText();
      JOptionPane.showMessageDialog(null, // show simple info.
                          " Button [" + butStr +  "] clicked");
    });
// ...
}
```

Handling Mouse Events

- Mouse events include such user interactions: e.g.
  - Moving, Clicking, Dragging the mouse (moving the mouse while the mouse button is being pressed)
- The `MouseListener` interface handles mouse button events
  - E.g. `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed`, and `mouseReleased`
- The `MouseMotionListener` interface handles mouse motion events
  - E.g. `mouseDragged` and `mouseMoved`

Reference for More on Writing Event Listeners:
Reference: https://docs.oracle.com/javase/tutorial/uiswing/events/index.html

Listeners that `Swing` Support - 1

Component	Action Listener	Caret Listener	Change Listener	Document Listener, Undoable Edit Listener	Item Listener	List Selection Listener	Window Listener	Other Types of Listeners
button	✔		✔		✔			
check box	✔		✔		✔			
color chooser			✔					
combo box	✔				✔			
dialog							✔	
editor pane		✔		✔				hyperlink
file chooser	✔							
formatted text field	✔	✔		✔				
frame							✔	
internal frame								internal frame
list						✔		list data
menu								menu
menu item	✔		✔		✔			menu key menu drag mouse
option pane								
password field	✔	✔		✔				

Listeners that `Swing` Support - 2

Component	Action Listener	Caret Listener	Change Listener	Document Listener, Undoable Edit Listener	Item Listener	List Selection Listener	Window Listener	Other Types of Listeners
popup menu								popup menu
progress bar			✔					
radio button	✔		✔		✔			
slider			✔					
spinner			✔					
tabbed pane			✔					
table						✔		table model table column model cell editor
text area		✔		✔				
text field	✔	✔		✔				
text pane		✔		✔				hyperlink
toggle button	✔		✔		✔			
tree								tree expansion tree will expand tree model tree selection
viewport (used by scrollpane)			✔					

List of Some Event Listeners

Listener Interface	Adapter Class	Listener Methods
ActionListener	*none*	actionPerformed(ActionEvent)
CaretListener	*none*	caretUpdate(CaretEvent)
CellEditorListener	*none*	editingStopped(ChangeEvent) editingCanceled(ChangeEvent)
ChangeListener	*none*	stateChanged(ChangeEvent)
DocumentListener	*none*	changedUpdate(DocumentEvent) insertUpdate(DocumentEvent) removeUpdate(DocumentEvent)
ItemListener	*none*	itemStateChanged(ItemEvent)
KeyListener	KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
ListSelectionListener	*none*	valueChanged(ListSelectionEvent)
MenuListener	*none*	menuCanceled(MenuEvent) menuDeselected(MenuEvent) menuSelected(MenuEvent)
MouseListener	MouseAdapter, MouseInputAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener	MouseMotionAdapter, MouseInputAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener	WindowAdapter	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)
WindowStateListener	WindowAdapter	windowStateChanged(WindowEvent)

Listener Interfaces in package `java.awt.event`

Package java.awt.event

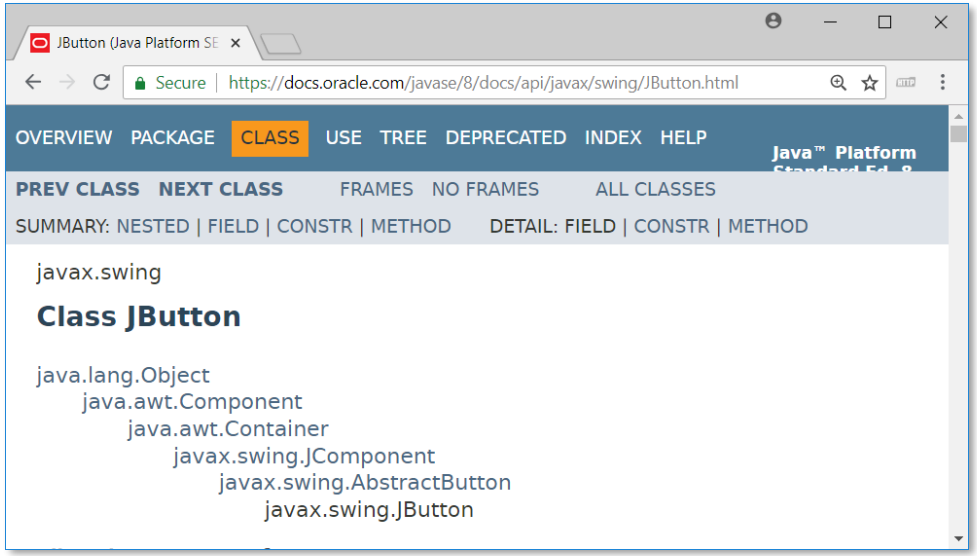
Provides interfaces and classes for dealing with different types of events fired by AWT components. See the `java.awt.AWTEvent` class for details on the AWT event model. Events are fired by event sources. An event listener registers with an event source to receive notifications about the events of a particular type. This package defines events and event listeners, as well as event listener adapters, which are convenience classes to make easier the process of writing event listeners.

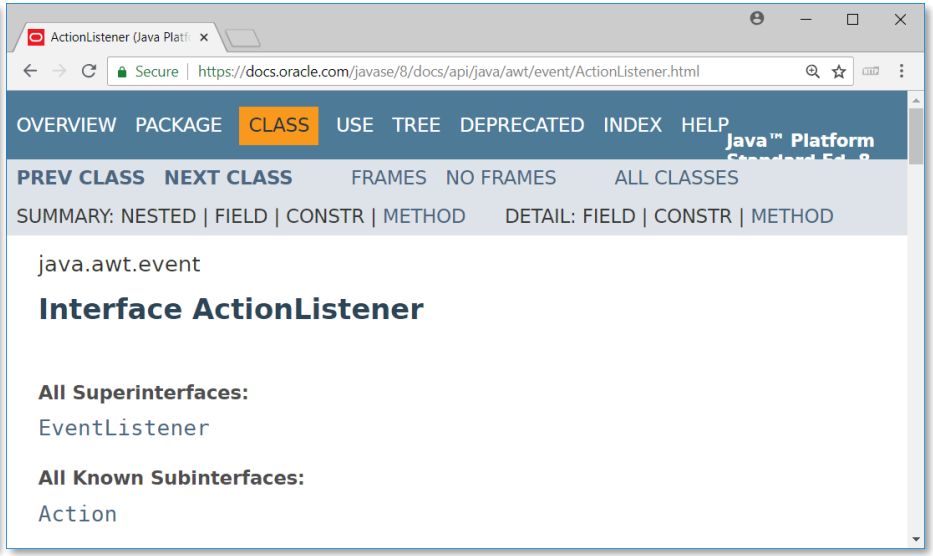
Since:
1.1

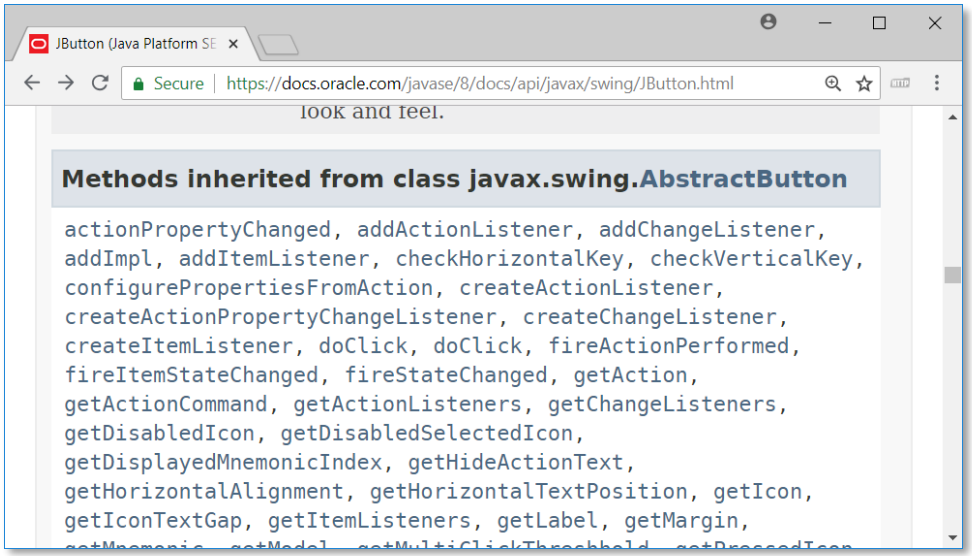
Interface Summary

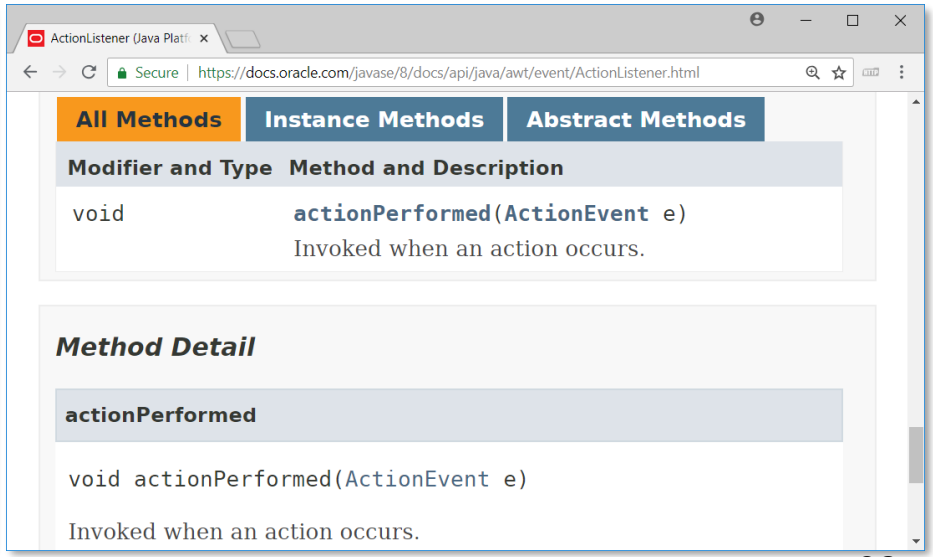
Interface	Description
ActionListener	The listener interface for receiving action events.
AdjustmentListener	The listener interface for receiving adjustment events.
AWTEventListener	The listener interface for receiving notification of events dispatched to objects that are instances of Component or MenuComponent or their subclasses.
ComponentListener	The listener interface for receiving component events.
ContainerListener	The listener interface for receiving container events.
FocusListener	The listener interface for receiving keyboard focus events on a component.
HierarchyBoundsListener	The listener interface for receiving ancestor moved and resized events.
HierarchyListener	The listener interface for receiving hierarchy changed events.
InputMethodListener	The listener interface for receiving input method events.
ItemListener	The listener interface for receiving item events.
KeyListener	The listener interface for receiving keyboard events (keystrokes).
MouseListener	The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component.
MouseMotionListener	The listener interface for receiving mouse motion events on a component.
MouseWheelListener	The listener interface for receiving mouse wheel events on a component.
TextListener	The listener interface for receiving text events.
WindowFocusListener	The listener interface for receiving `WindowEvents`, including `WINDOW_GAINED_FOCUS` and `WINDOW_LOST_FOCUS` events.
WindowListener	The listener interface for receiving window events.
WindowStateListener	The listener interface for receiving window state events.

API for Event Sources & Listeners - 1

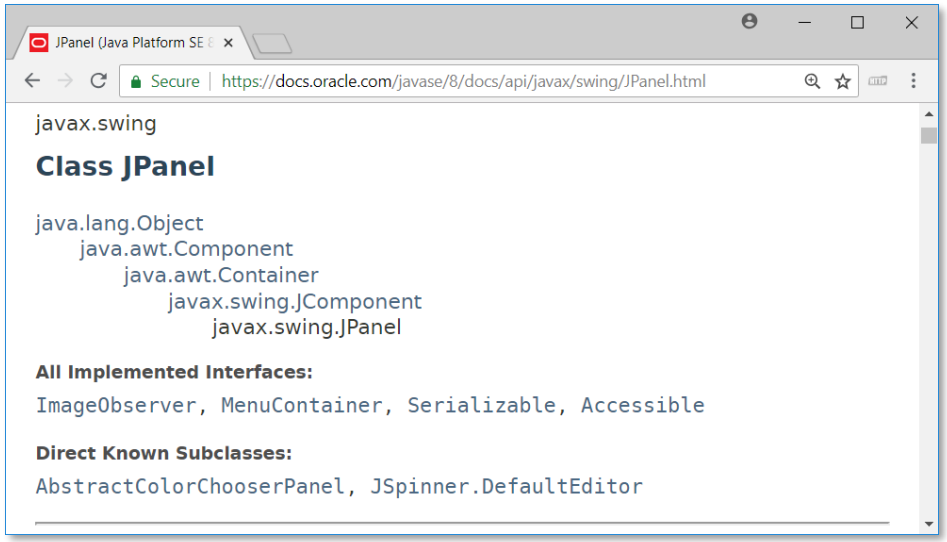


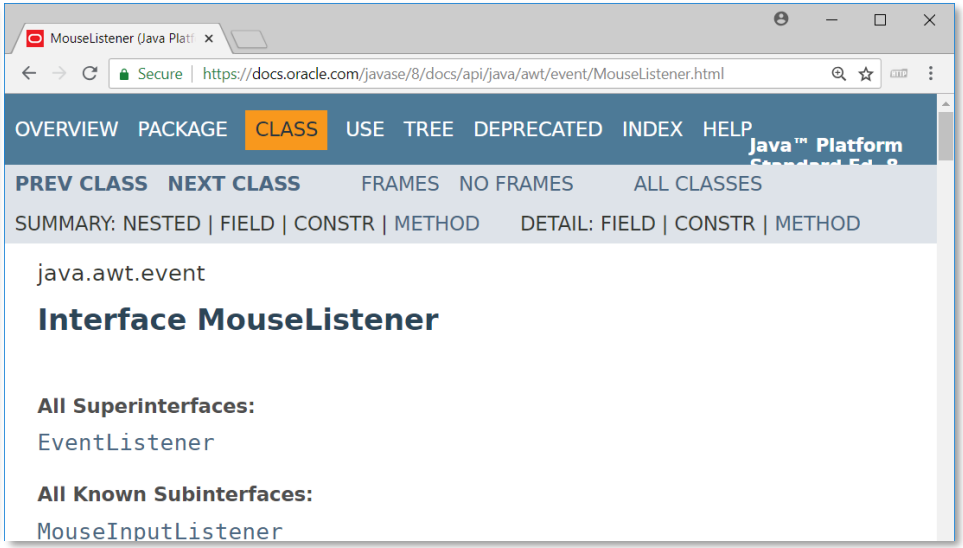


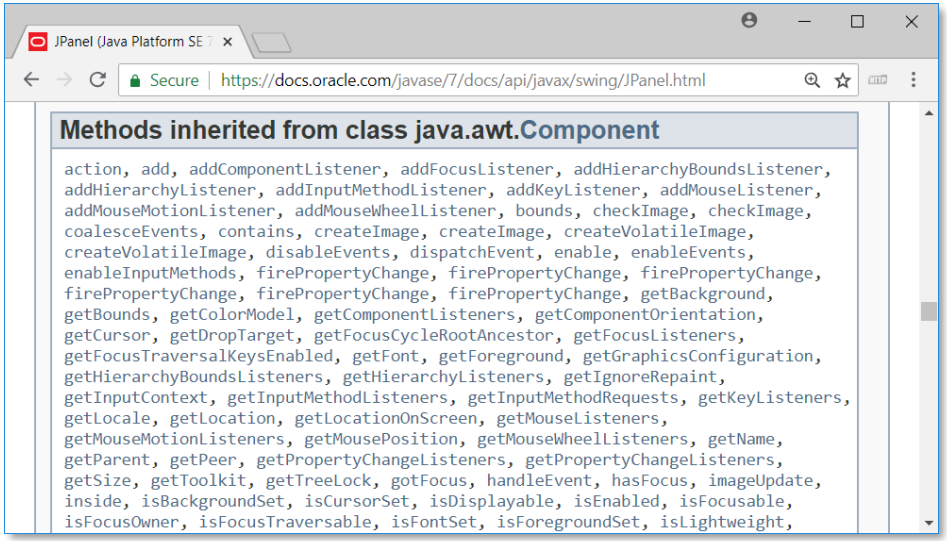


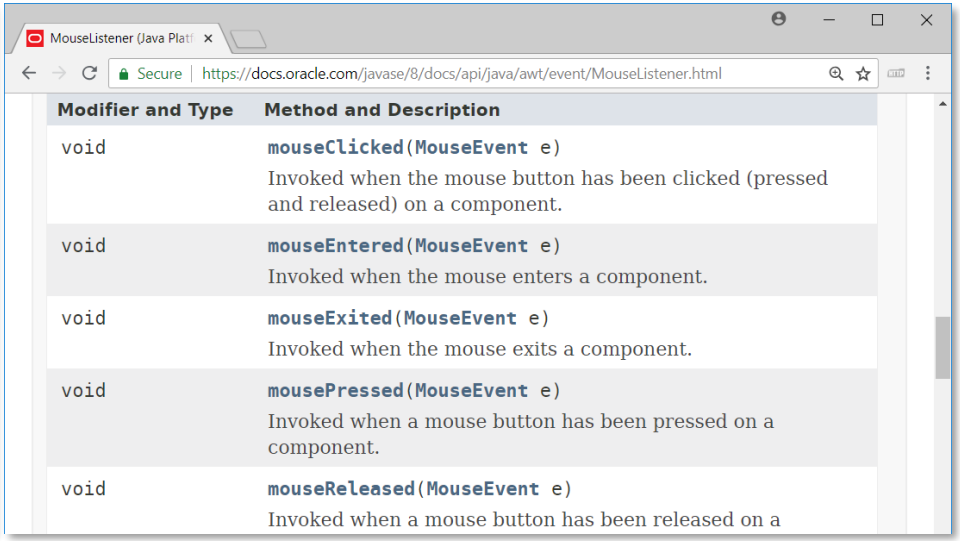


API for Event Sources & Listeners - 2









References

- This set of slides is only for educational purpose.

- Part of this slide set is referenced, extracted, and/or modified from the followings:
  - Deitel, P. and Deitel H. (2017) “Java How To Program, Early Objects”, 11ed, Pearson.
  - Liang, Y.D. (2017) “Introduction to Java Programming and Data Structures”, Comprehensive Version, 11ed, Prentice Hall.
  - Wu, C.T. (2010) “An Introduction to Object-Oriented Programming with Java”, 5ed, McGraw Hill.
  - Oracle Corporation, “Java Language and Virtual Machine Specifications” https://docs.oracle.com/javase/specs/
  - Oracle Corporation, “The Java Tutorials” https://docs.oracle.com/javase/tutorial/
  - Wikipedia, Website: https://en.wikipedia.org/