Building Components using Swing and JavaFX

Prativa Nyaupane

Recap

- GUI Controls
- Menu Elements and Tooltips
- Dialogs

Today’s Objective

- Frames
- Event handling and Listener Interfaces
- Handling Action Events

Frames

- Frames are the main windows of a GUI application.
- They serve as containers for other GUI elements, such as panels, controls and dialogs.
- Frames provide the overall structure of the application.
- They contain and organize the components necessary for the application’s functionality.

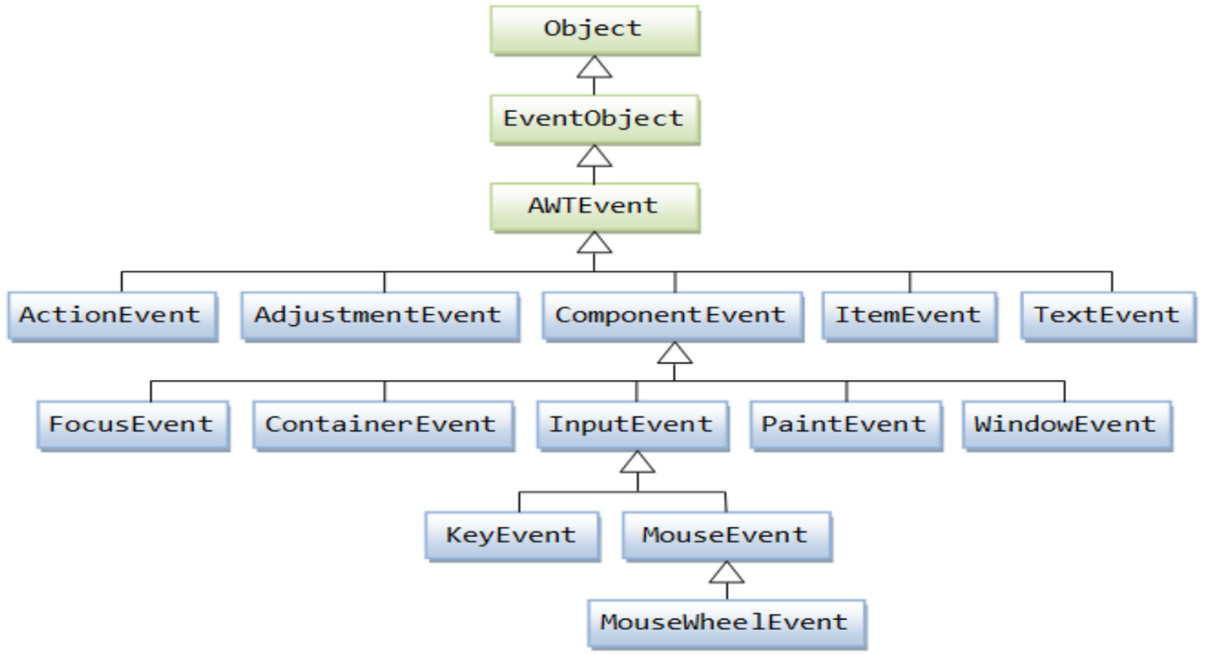
Events

- Any operating environment that supports GUI constantly monitor events, such as keystrokes or mouse clicks.
- The operating system reports these events to the programs that are running.
- Each program then decides, what, if anything to do in response to these events.
- Any program that is a GUI is event driven.
- Supported by packages: java.util, java.awt, java.awt.event

Events Contd..

- Events are generated when the user interacts with the GUI based program.
- Passed to program in a variety of ways.
- Several types of events - events generated by mouse, keyboard, button, scrollbar or a check box.

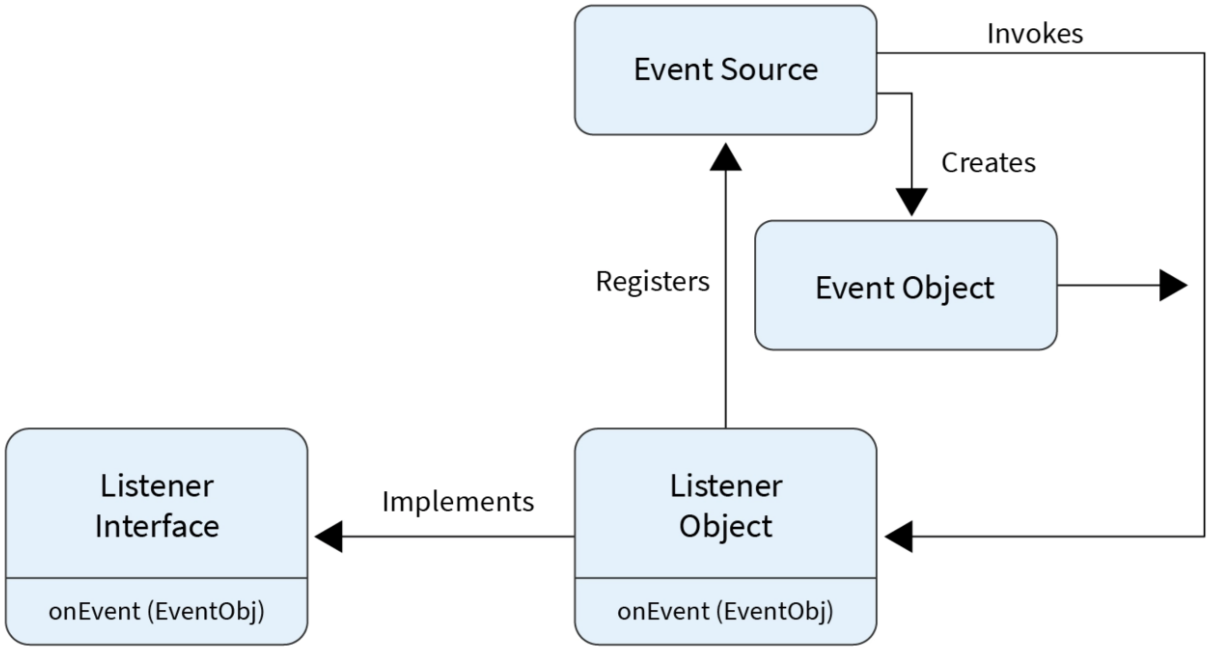
Event Class Diagram



Events

- An event is an object that describes a state change in the process.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Events can be generated
    - Pressing a button
    - Entering a character via the keyboard.
    - Selecting an item in list.
    - Clicking the mouse.

Event Handling



Event Handling Contd..

- Java uses Event Delegation Model to handle the events.
- This model defines the standard mechanism to generate and handle the events.
- A source generates an event and sends it to one or more listeners.
- Source can be a button, text field or any GUI component.
- Listener is an object that is registered to event source and is responsible for handling events that are generated by source.

Event Handling Contd...

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method executes and returns.

Event in AWT/SWING

- To handle events, we need to implement event listeners or adapters.

  Some commonly used listener interfaces in AWT include:

  - ActionListener: Handles action events.
  - MouseListener: Handles mouse-related events.
  - KeyListener: Handles keyboard-related events.
  - WindowListener: Handles window-related events.
- In AWT/Swing both adapters and event listeners are mechanisms for handling user inputs within GUI applications. However, they differ in implementation and usages.

Event Listeners vs Adapters

Event Listeners	Adapters
They are interfaces that define methods for handling specific types of events.	Abstract classes that provide defaults implementations for all methods of an event listener interface.
To handle an event using listeners, you need to implement the corresponding listener interface and provide an implementation for its methods.	It allows us to create objects that selectively override only the methods for the events we are interested in handling, rather than implementing all methods of the interface.
For example, to handle mouse clicks, you would implement the MouseListener interface and provide implementations for methods like mouseClicked, mousePressed, mouseReleased, etc.	For example, instead of implementing the MouseListener interface directly, you can extend the MouseAdapter class, which provides empty default implementations for all methods of MouseListener. Then, you only need to override the specific methods you want to handle.

Using Event Listeners:

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

// Assuming you have a JButton named "button"
       button.addMouseListener(new MouseListener() {
   @Override
   public void mouseClicked(MouseEvent e) {
       // Handle mouse click event here
       System.out.println("Mouse clicked!");
   }

   // Other methods of MouseListener interface that you must implement
   @Override
   public void mousePressed(MouseEvent e) {}
   @Override
   public void mouseReleased(MouseEvent e) {}
   @Override
   public void mouseEntered(MouseEvent e) {}
   @Override
   public void mouseExited(MouseEvent e) {}
});
```

Using Adapters

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

// Assuming you have a JButton named "button"
        button.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        // Handle mouse click event here
        System.out.println("Mouse clicked!");
    }
});
```

AWT Event Hierarchy

- The AWT (Abstract Window Toolkit) Event Hierarchy refers to the structure and organization of different types of events in AWT
- Event Object:
  - At the top of the hierarchy is the java.util.EventObject class.
  - This class serves as the base class for all events in Java.
  - It provides common methods and properties that are shared by all event objects.
- Component-Level Events:
  - Component-level events are generated by individual GUI components (e.g., buttons, text fields, etc.).
  - These events are subclasses of java.awt.AWTEvent.

AWT Event Hierarchy Contd...

- Container-Level Events:
  - Container-level events are generated by container components (e.g., panels, frames, etc.) and are subclasses of java.awt.event.ContainerEvent.
  - These events capture changes in the container's structure, such as components being added, removed, or rearranged within a container.
- Window-Level Events:
  - Window-level events are generated by top-level windows, such as frames or dialog boxes, and are subclasses of java.awt.event.WindowEvent.
  - These events capture events related to window openings, closings, iconifications, or deiconifications.
- System-Level Events:
  - System-level events represent events that occur at the system level and are subclasses of java.awt.event.SystemEvent.
  - Examples include events related to the user session, such as the user logging in or logging out.

Semantics and low level Events in AWT, Event Handling

- Semantics in event handling refers to the **meaning or purpose associated with a particular event**. For example, a button click event may trigger a specific action like submitting a form.
- AWT provides low-level events that represent low-level input, such as mouse or keyboard actions. These events are handled by components like buttons or text fields.
- The AWT event handling mechanism involves registering event listeners or handlers on components to capture and process specific events.
- Event handlers implement specific listener interfaces, such as ActionListener or MouseListener, and define methods to handle events triggered by user actions.

Individual Events. Separating GUI and Application code

- Individual events refer to specific actions or occurrences triggered by the user, such as button clicks, menu selections, or mouse movements.
- When handling events, it is good practice to separate the Graphical User Interface (GUI) code from the application logic code.
- The GUI code is responsible for creating and managing the visual components, while the application logic code handles the business logic or functionality associated with the events.
- By separating the GUI and application code, it becomes easier to modify or enhance the application without impacting the visual presentation or layout.

Multicasting

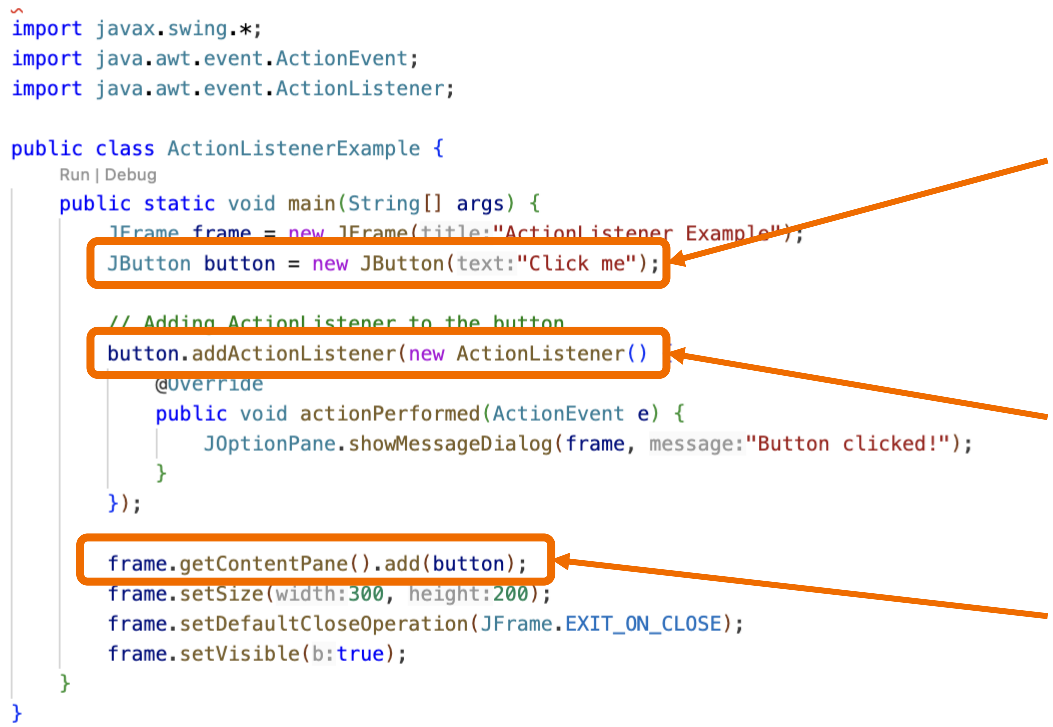
- Multicasting in the **context of event handling refers to the ability to handle an event with multiple event listeners**.
- In Java, you can register multiple event listeners to a single component or event source using the concept of multicasting.
- When an event occurs, all registered listeners will be notified and can respond to the event.
- Multicasting is particularly useful when you want multiple components or modules to react to the same event simultaneously.
- It allows for a modular and flexible design, where different parts of the application can independently register and handle events without tightly coupling them together.
- To implement multicasting, Java provides event listener interfaces and corresponding registration methods (such as addActionListener() or addMouseListener()) that accept multiple listener objects.

Advance Event Handling

- Advanced event handling refers to the use of more sophisticated techniques and patterns to manage and process events in a Java application.
- It involves going beyond the basic event listener model to achieve more complex event-driven behavior.
- Advanced event handling techniques enable developers to create more complex and responsive applications by allowing fine-grained control over event flow and processing.
- These techniques are commonly used in large-scale applications, user interfaces with complex interactions, or frameworks that require extensive event handling capabilities.

Advance Event Handling (contd...)

- Some common techniques and patterns used in advanced event handling include:
  - Event Delegation: This pattern involves delegating event handling to a higher-level component or a central event manager, allowing for better organization and separation of concerns.
  - Event Filtering: It involves filtering events based on specific criteria before they reach the event listeners, allowing for selective processing of events.
  - Event Chaining: It involves chaining multiple event handlers together, where the output of one handler becomes the input for the next, enabling event processing pipelines.
  - Custom Event Types: In addition to the built-in event types, developers can create their own custom event classes to represent application-specific events and handle them appropriately.



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

public class ActionListenerExample {
    Run | Debug
    public static void main(String[] args) {
        JFrame frame = new JFrame(title:"ActionListener Example");
        JButton button = new JButton(text:"Click me");

        // Adding ActionListener to the button
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, message:"Button clicked!");
            }
        });

        frame.getContentPane().add(button);
        frame.setSize(width:300, height:200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(b:true);
    }
}
```

Creating Button Object

Adding Action Listener to the button for showing the popup

Adding button to the frame