

CS6308

JAVA PROGRAMMING

V P Jayachitra

Syllabus

MODULE IV GUI	L	T	P	EL
	3	0	4	3
Creating UI, Frames, layout manager, Panels, components, Event Driven Programming				
SUGGESTED ACTIVITIES : <ul style="list-style-type: none">• flipped classroom• Practical – Mouse, key events, creating interactive forms using AWT/Swing and adding functionality• EL – Understand AWT and SWING				

Event-Driven Programming

- **Definition:** Event-driven programming is an **event-driven paradigm** in which the flow of the program is determined by events such as **mouse click, key pressed or sensors output or message** from other programs or threads.
- **Program control flow is determined by events or sensor input or user actions or messages**
- **Event driven is a asynchronous process**
 - since components don't communicate directly, or waiting for a response from each other.
 - The services that participate in the communication, simply publish a message and then moves on. Whoever is interested can listen to the event, whenever is useful for them.
- **Asynchronous** is about not blocking threads of execution and it's often more efficient use of resources. It helps minimize congestion on shared resources in the system, this improves scalability, low latency, and high throughput.

Event-Driven Programming


- Event-driven programs do not control the sequence in which input events occur; instead, they are written to **react to any reasonable sequence of events**.
- Events also do not occur in any particular or predictable order.
- In this control model, **the input data govern the particular sequence of operations that** is carried out by the program, **but the program does not determine the input sequence** order.
- Moreover, an event-driven program is usually designed to **run indefinitely**, or at least until the user selects an exit button.
- Example of an event-driven program:
 - **Graphical user interface (GUI)**
 - Used in desktop , laptop computers and smartphones
 - **Web-based applications**
 - online student registration system, online airline reservation system
 - **Embedded applications**
 - cell phones, car engines, airplane navigation and home security systems
- **Model-view-controller (MVC) design pattern** provide effective support for event-driven programming
 - Languages that support MVC framework or design pattern are Java, Visual Basic, Tcl/Tk, JavaScript, Python, Perl, Ruby, PHP, C#, Swift

EVENT-DRIVEN CONTROL


- **How Event driven programming differs from Traditional view ?**
 - In general, statements are executed in sequential, from the entry point in main, according to which conditions occur during execution.
 - However, GUIs follow a style of programming called event-driven programming must be able to execute statements according to some events that may occur at any time while the program in execution.
 - According to Stein (1999), Computation is a function from its input to its output.
 - Computation is made up of a sequence of functional steps that produce - at its end - some result as its goal. These steps are combined by temporal sequencing.
 - Unpredictable modern computations and solutions: Modern computations are embedded in physical environments where the temporal sequencing of events is unpredictable and (potentially) without an explicit end.
 - To cope with this unpredictability computation should be modeled as *interaction!*
- Computation is a community of persistent entities coupled together by their ongoing interactive behaviour.
 - Example applications under this view includes, robotics, video games, global positioning systems, and home security alarm systems.

Event-Driven Programming

Conventional model

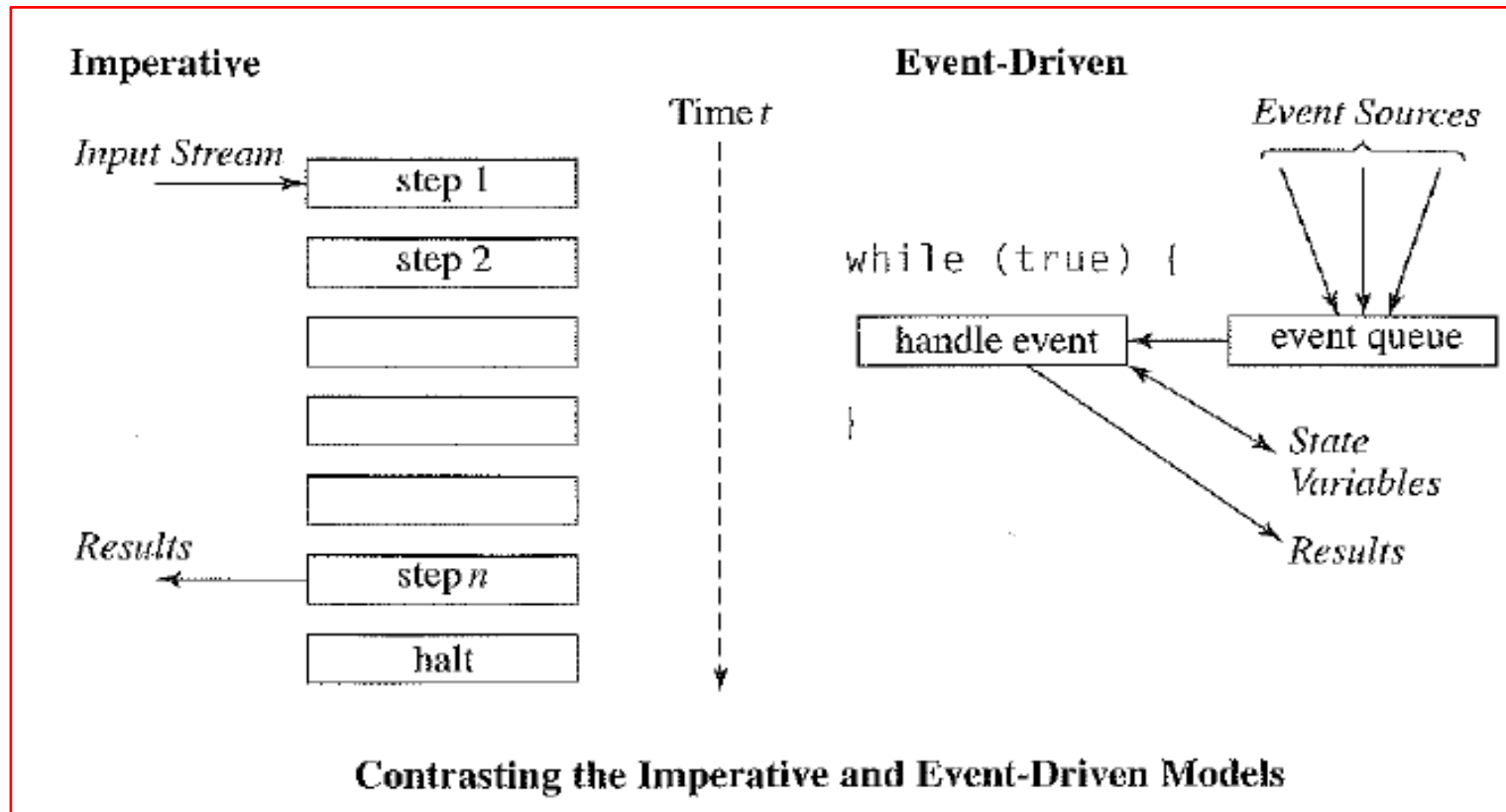
- The program is designed so that the input order determines the computational design. 
 - input is gathered near the beginning of the computation and results are emitted near the end.
- The programmer determines the order of the input data, which in turn determines the computational design.
- The computation consumes input data and produces results in a continuous loop.
- Supported paradigms:
 - Imperative, object-oriented, functional and logic
- Control flow is based on the structure
 - Program controls the sequence of steps that occur at run time

Event Driven model

- The program prescribes the set of event handlers. 
 - The input comes from distinct autonomous *event sources*. i.e., sensors on a robot or buttons in an interactive application.
- Can't determine Which events happen in which order?
- The program may consult and/or change the value of a variable or produce intermediate *results*.
- Supported paradigms:
 - Event-driven paradigm
- Control flow is based on the internal and external events
 - Flow of control of program to change according to events that are external to the program itself.

Properties of event-driven programs

- Importantly, two *properties of event-driven programs*:
 - An event-driven program typically has **no preconceived stopping point**, such as reaching the end of file in reading data.
 - The explicit read-evaluate-print loop present in traditional input-controlled programs **does not explicitly** appear in event-driven programs.



Model-View-Controller

- Gamma *et al.*, 1995 discussed many Object-oriented design pattern, and, model-view-controller (MVC), is particularly useful in the design of event-driven GUI applications.
- **Definition:** In MVC the model is the actual object being implemented, the controller is the input mechanism (buttons, menus, combo boxes, etc.) and the view is the output representation of the model.
- MVC pattern it is good practice to decouple the model, the view, and the controller as much as possible.
- The MVC framework supports separation of logic and keeps application tasks separate, such as UI, business logic, and input.
 - Easily manage complexity
 - Supports Test Driven Development

MVC Architecture

- **Model**

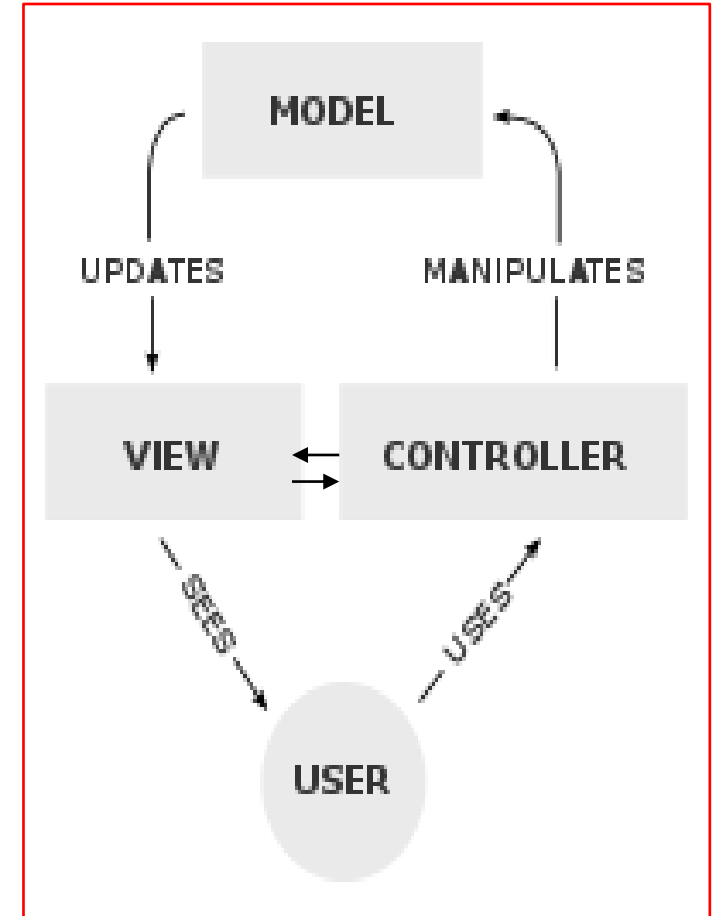
- Domain layer (database instances)
- **Manage application data and business logic**
- Accepts request from controller and update its content accordingly

- **View**

- **UI**, display results to user
- Accepts request from controller and generate output using model

- **Controller**

- Control logic , Handles **user interaction**.
- Receive user inputs, instruct model what to do and view what to display



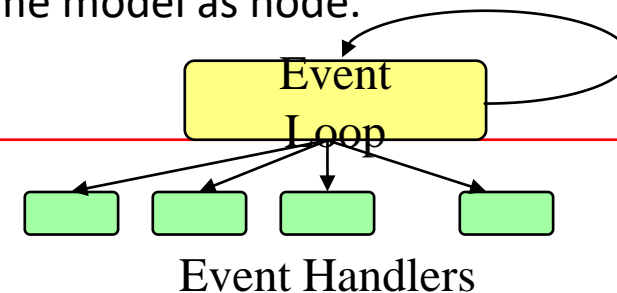
Source: Wikipedia

Thread

- **Multithreading**: allows applications to handle activities concurrently leading to efficient execution.
- Solution for concurrency
- Multiple independent execution streams
- **Shared state**
- **Synchronization**(locks, conditions)
- synchronous I/O operation
- Only use threads where true CPU concurrency is needed. Scalable performance on multiple CPUs
- Use threads only for performance-critical kernels.
- **Apache** rely on multithreading
- important for high-end servers (e.g. **databases**).
- **Problem**:
 - Deadlock
 - Threads break abstraction, i.e can't design modules independently
 - Locking increases complexity and reduces performance

Event

- **Event loop**: uses indefinite loop to receive requests and process them
 - When multiple requests are made, the first is processed while the rest are blocked (until the first is complete).
 - Handling more and more concurrent client's request is very easy.
- no concurrency, no preemption, no synchronization, no deadlock.
- One execution stream i.e one handler for each event(button press, release)
- Support **Callback mechanism**
- **Asynchronous** I/O operation
- Use events, not threads, for GUIs, distributed systems, low-end servers
- Use events as primary development tool (both GUIs and distributed systems).
- **Node.js** based on asynchronous I/O eventing model to create high performance web applications easily. Single threaded event loop model.
- Facebook's chat server is being powered by **Erlang**, which uses same model as node.

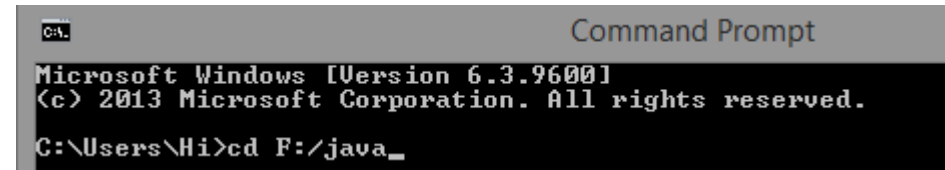


Graphical User Interface

UI User Interface

- Command Line Interface (CUI)

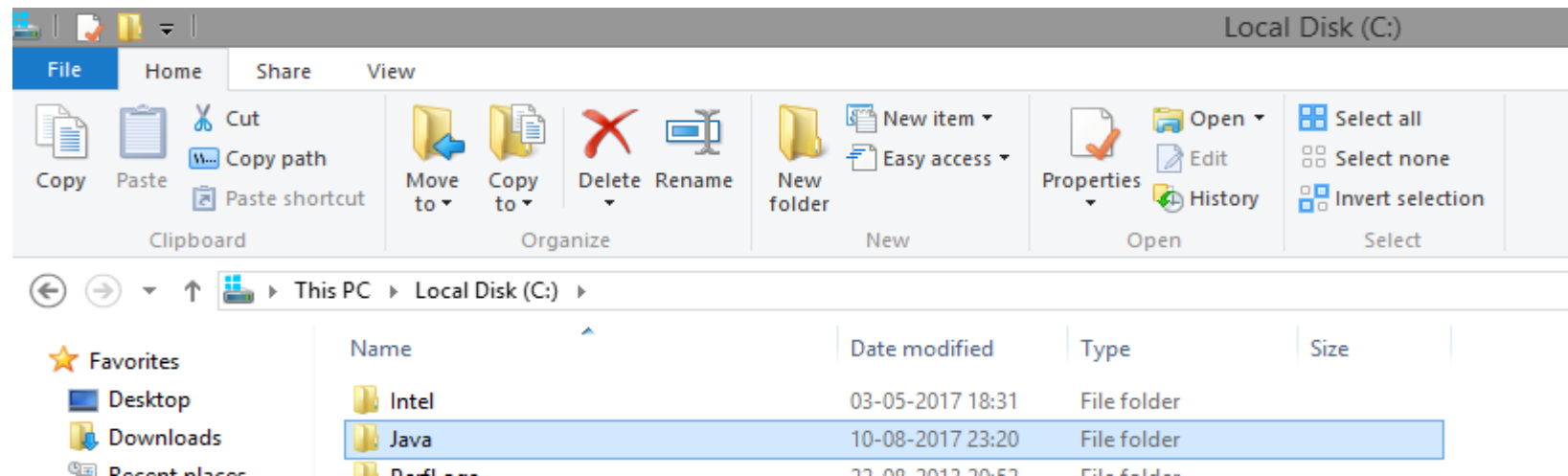
- Type commands using keyboard to interact with computers.
- Non-graphical UI



```
C:\>  
Microsoft Windows [Version 6.3.9600]  
<©> 2013 Microsoft Corporation. All rights reserved.  
C:\Users\Hi>cd F:/java_
```

- Graphical User Interface(GUI)

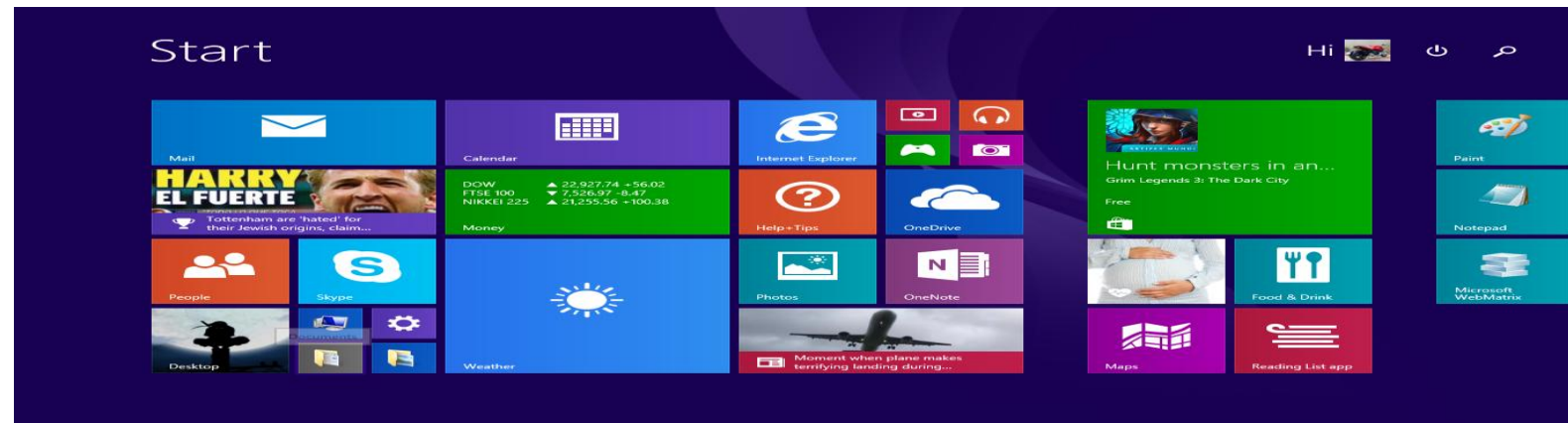
- Click readymade commands or icons using mouse to interact with computer.



GUI

- Graphical User Interface
- G-U-I or Gooney
- Its an **interface between human and computer interaction.**
- Includes graphical elements such as icons, buttons, windows or visual indicators to interact with electronic devices.
- Introduced by **Steve Jobs** for Apple Macintosh in 1984, Windows 1 in 1985, Windows 95 for Microsoft office, Nokia and Apple iphone 2007,ipad in 2010, Windows 8 and 10 in 2012 and 2015, Holograms and Virtual reality now.

A graphical user interface (GUI) application is a program that runs inside its own window and communicates with users using graphical elements such as buttons and menus.



GUI in Java

- **JFC** Java Foundation Class
 - API that provides features for building GUI, Graphics and interactivity to Java applications.
- **AWT** Abstract Window Toolkit
- **SWING**
- **JavaFX**

AWT vs Swing

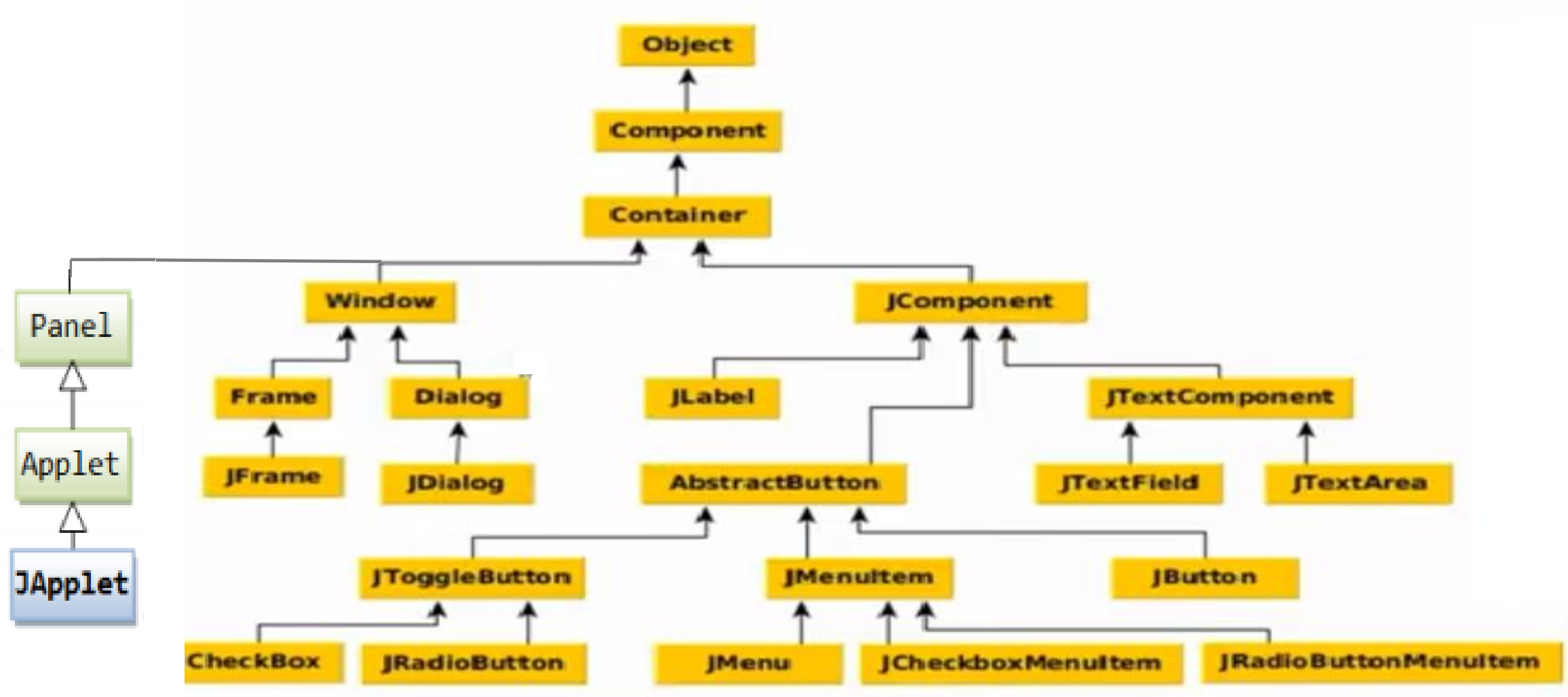
AWT

- Heavy-weight components
- Platform specific
 - Rely on native GUI
 - need Paint() of specific platform to draw
- Introduced in java 1.1
- Simple applications
- Package java.awt

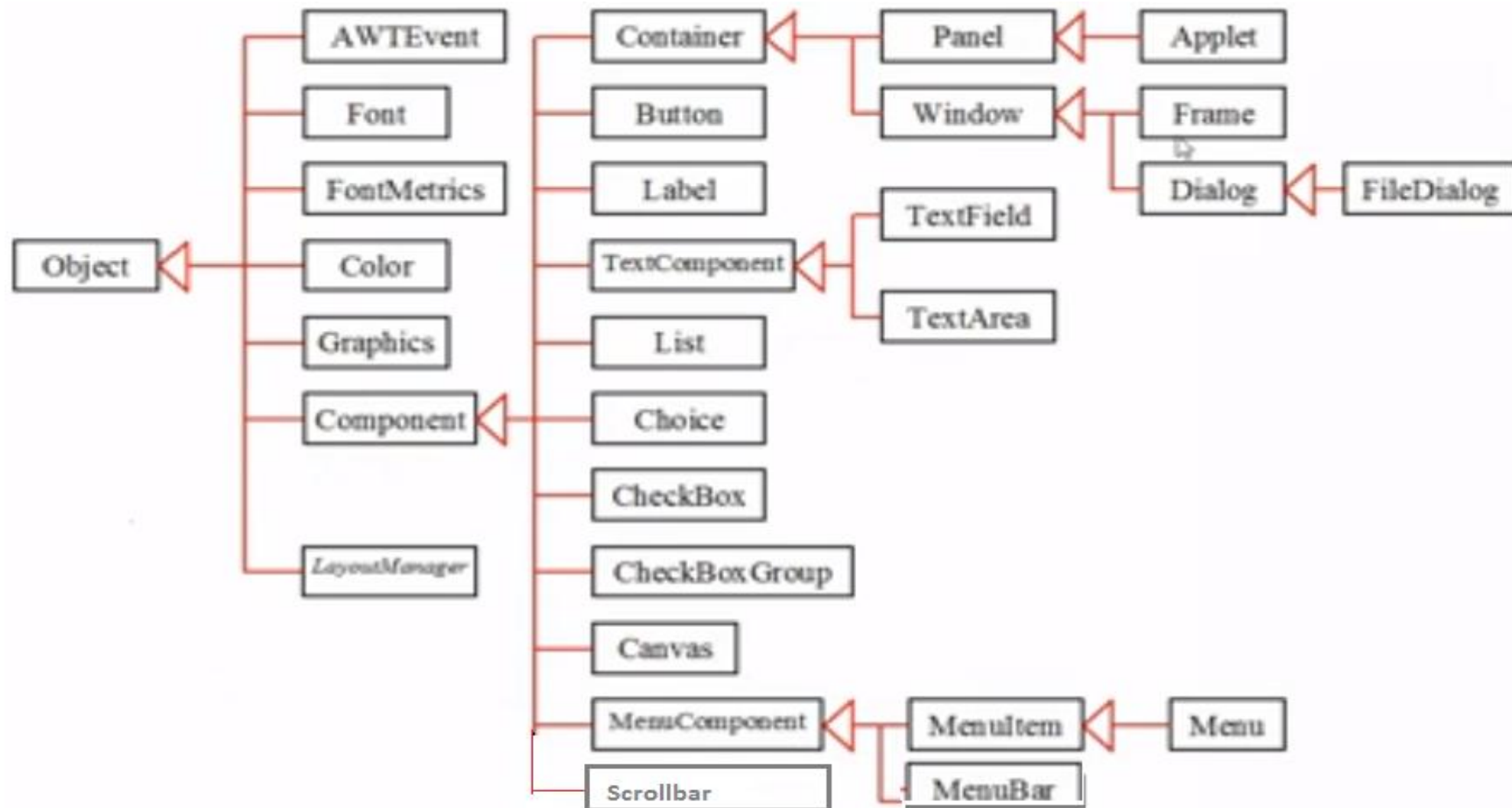
SWING

- Light-weight components
- Platform independent
 - Donot rely on native GUI except for java.awt.Window and java.awt.Panel
- Introduced in java 2
- Complex applications
- All classes begins with 'J'
 - i.e., JButton, JTextField etc.,
- Package javax.swing

Swing classes



AWT



Ref : google.com/image

Events in Java: Event Handling Model

Name

- Event object
- Event source
- Event listener
- Event handling methods

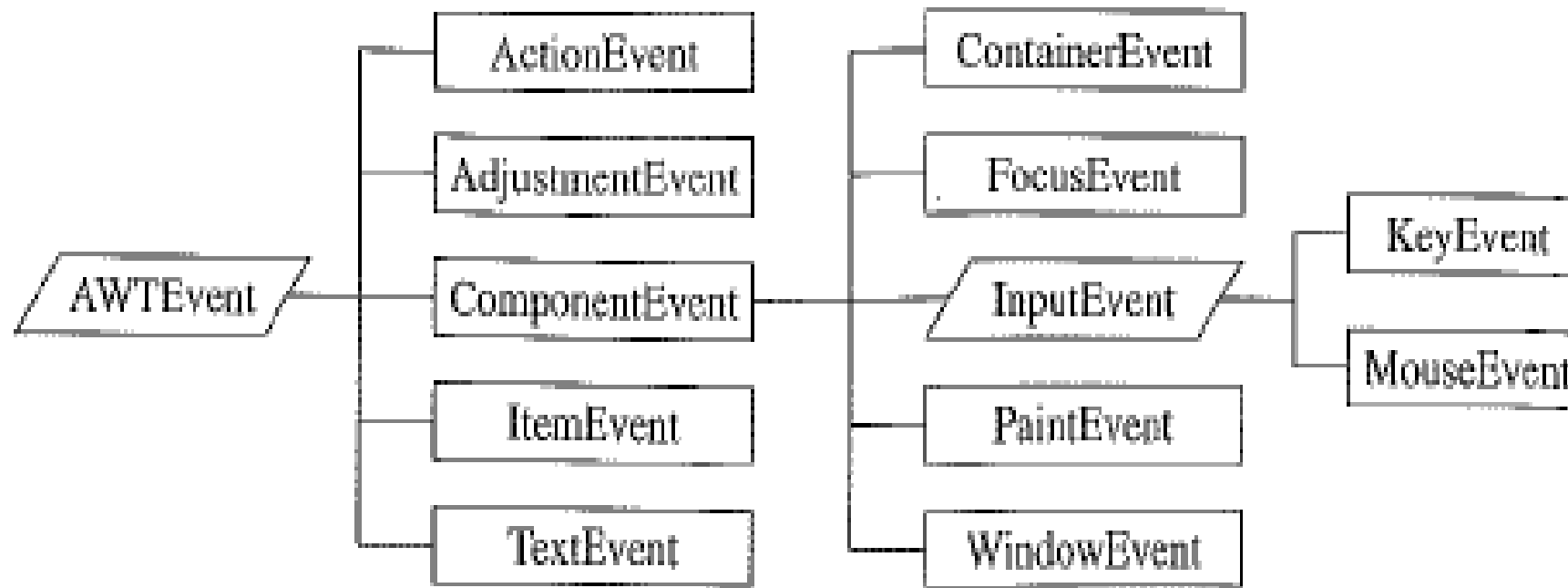
s -> event source
l -> event listener of type XXX
s.addXXX(l)

Element

- ActionEvent
- Button
- ActionListener
- actionPerformed(..)

```
b = new Button("Hello!");  
add(b);  
b.addActionListener(this);
```

- The types of **events that can occur in Java** are defined by the subclasses of the predefined abstract class **AWTEvent**.
- Every event source in an interaction can generate an event that is a member of one of these classes.
- For instance, if a **button is an event source**, it generates **events** that are **members of the ActionEvent** class.



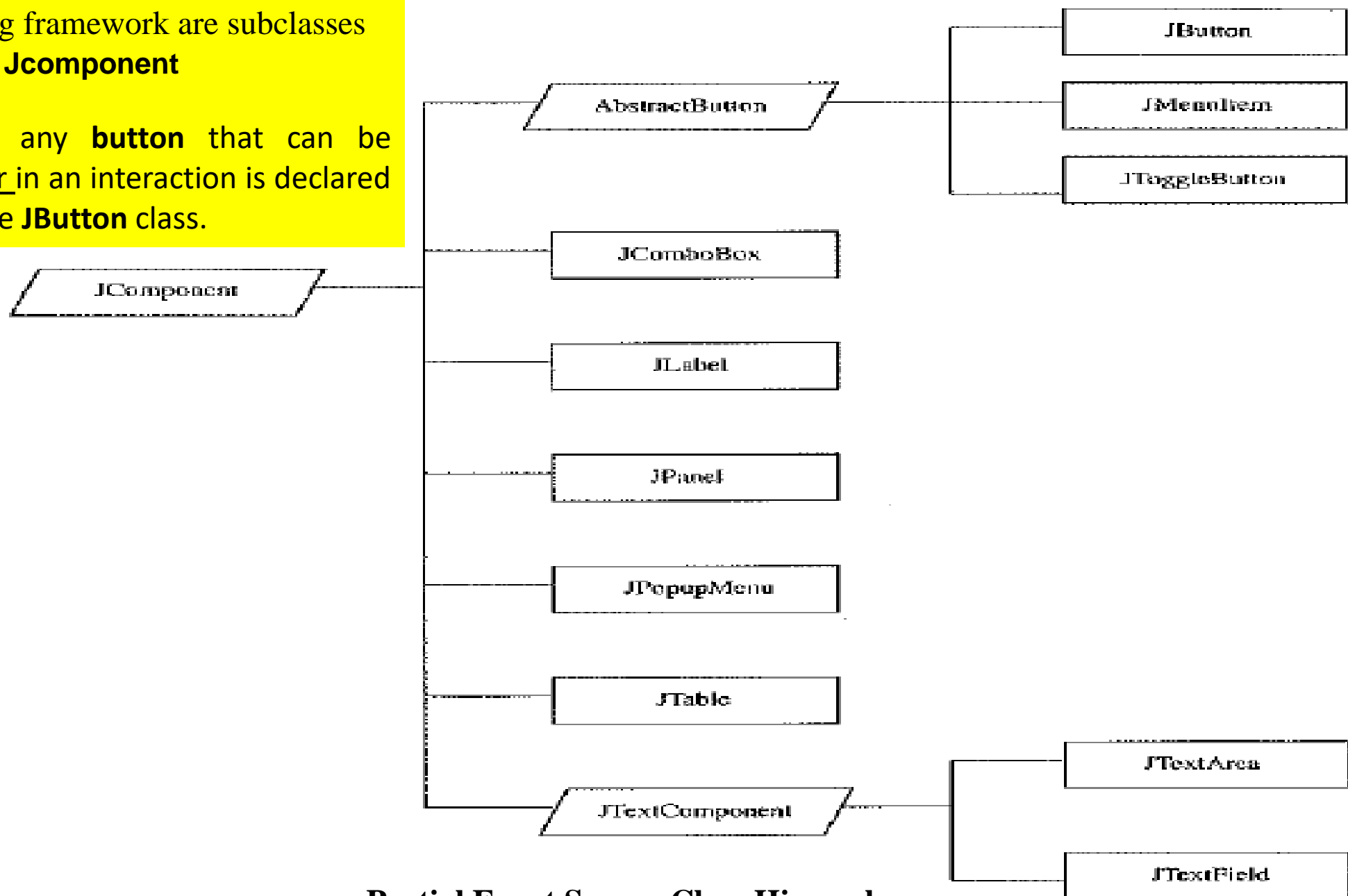
Java Class AWTEvent and Its Subclasses

Event Class : Description

ActionEvent	button is pressed; list item is double-clicked; menu item is selected;
AdjustmentEvent	scroll bar is manipulated. [String getActionCommand() , long getWhen()]
ComponentEvent	component is hidden, moved, resized, visible.
ContainerEvent	component is added to a container or removed from a container.
FocusEvent	component gains keyboard focus or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	check box or list item is clicked , selected or deselected. [int getStateChange(), Object getItem()]
KeyEvent	input is received from the keyboard.
MouseEvent	mouse is dragged, moved, clicked, pressed, or released; mouse enters or exits a component.
MouseWheelEvent	mouse wheel is moved
MouseWheelEvent	text area value or text field value is changed.
WindowEvent	window is activated, closed, deactivated, opened, or quit.

The objects themselves that can be **event sources** in the Swing framework are subclasses of the abstract class **JComponent**

for example, that any **button** that can be selected by the user in an interaction is declared as an **instance** of the **JButton** class.

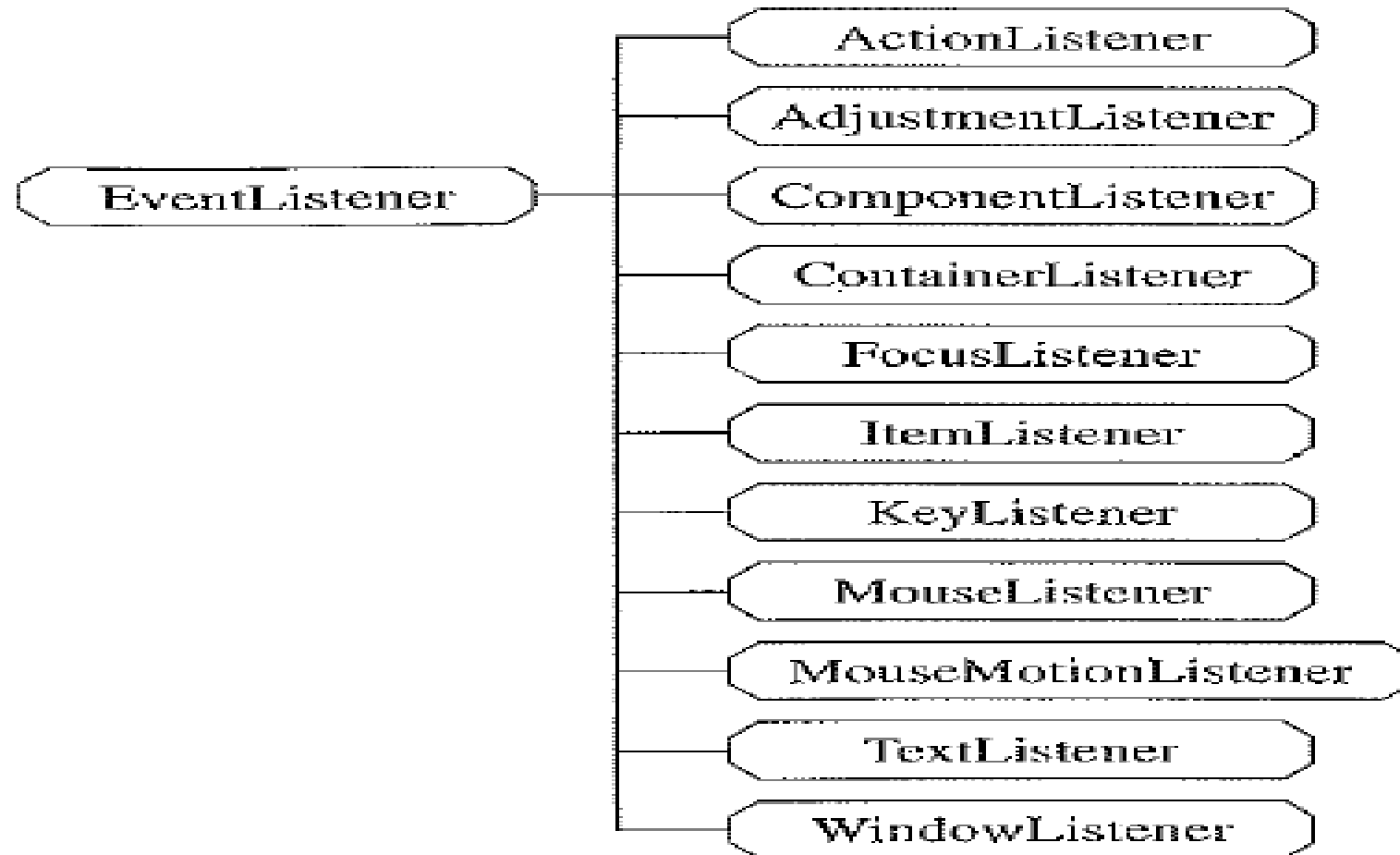


Partial Event Source Class Hierarchy

For a **program to handle an event**, it must be equipped with appropriate "listeners" that will recognize when a particular event, such as a **mouse click**, has occurred on an object that is an **event source**.

For example, to equip a **button** so that the program can "detect" an occurrence of that button's selection, the button needs to invoke its **addActionListener** method.

If this is not done, button events will not be detected by the program.



Java EventListener Classes

To respond to events initiated by objects in these classes, we need to implement **special methods** called *handlers*. Each class of events predefines the name(s) of the handler(s) that can be written for it.

Components and Their Event Handlers

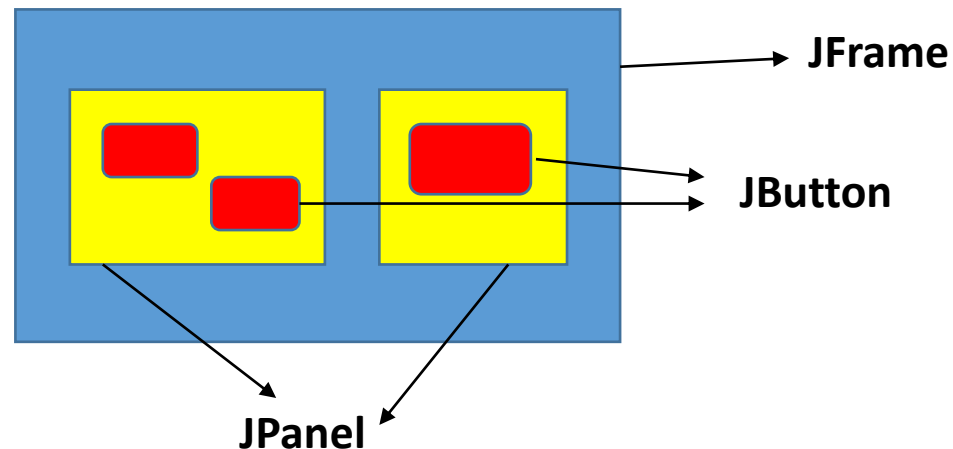
Widget	Listener	Interface Methods
JButton	ActionListener	actionPerformed(ActionEvent e)
JComboBox	ActionListener	actionPerformed(ActionEvent e)
JLabel	MouseListener	mouseClicked(MouseEvent e) mouseEntered(MouseEvent e) mouseExited(MouseEvent e) mousePressed(MouseEvent e) mouseReleased(MouseEvent e)
	MouseMotion- Listener	mouseDragged(MouseEvent e) mouseMoved(MouseEvent e)
JTextArea	ActionListener	actionPerformed(ActionEvent e)
JTextField	ActionListener	actionPerformed(ActionEvent e)

Four different kinds of user-initiated events can be handled by:

- **Mouse motion events (handled by the `MouseMotionListener`).**
 - **Mouse events (handled by the `MouseListener`).**
 - **Button and text field selections (handled by the `ActionListener`).**
 - **Selections from a combo box (handled by the `ActionListener`).**
-
- Importantly, the program cannot know, or predict, the order in which different events will occur, or the number of times each one will be repeated;
 - Program has to be prepared for all possibilities. That is the essence of event-driven programming.
 - Another important difference is that the programmer writes each method to accomplish its task and exit.
 - The loop that awaits events and dispatches them is part of the framework that Java provides.

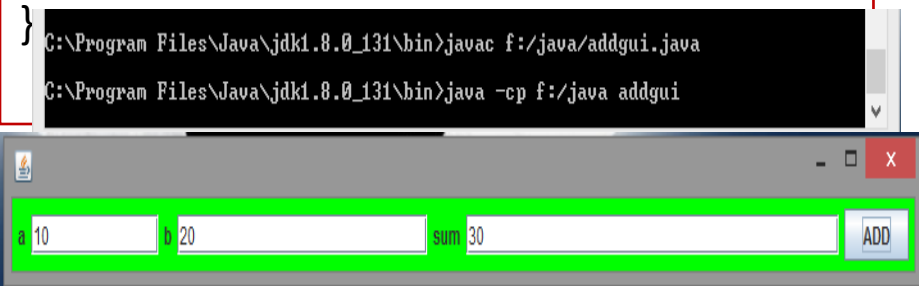
Java Swing Components

- Outer-level component: **JFrame** or **JDialog**
- inner components: **Jpanel**
 - containers, used to group and layout atomic components
- Atomic components: GUI elements
 - **JButton**, **JLabel**, **TextField**, **JTable**, **JScrollBar**, . . .



```
import java.awt.FlowLayout;
import java.awt.Color;

public class addgui{
    public static void main(String args[])
    {
        addition n =new addition();
    }
}
```



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

public class <classname> extends JPanel implements <listeners> {
    <instance variable declarations>

    public <classname> () {
        <code to initialize the GUI> }
    <event handlers>
}
```

Overall Structure of a GUI Java Application

```
//GUI example program to add two numbers using swing class
class addition extends JFrame implements ActionListener{
    JTextField a,b,sum;
    JButton add;
    JLabel al,bl,suml;

    addition()
    {
        al=new JLabel("a");      a=new JTextField(10);
        bl=new JLabel("b");      b=new JTextField(20);
        suml=new JLabel("sum");  sum=new JTextField(30);
        add=new JButton("ADD");
        add(al);add(a);add(bl); add(b);add(suml); add(sum); add(add);
        setLayout(new FlowLayout());
        setSize(400,400);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        getContentPane().setBackground(Color.GREEN);
        add.addActionListener(this);
        pack();    }

    public void actionPerformed (ActionEvent e){
        int num1=Integer.parseInt(a.getText());
        int num2=Integer.parseInt(b.getText());
        int totalSum=num1 + num2;
        sum.setText(Integer.toString(totalSum));
    }
}
```

The actionPerformed Method

- An actionPerformed method must have only one parameter, `ActionEvent`
- `public void actionPerformed(ActionEvent e) { if (e.getActionCommand().equals("click")) . . . }`

```
import java.awt.*;           // AWT components, and layout managers
import java.awt.event.*;     // event-handlers
import javax.swing.*;        // Swing components
```

```
public class SampleGUI extends JFrame {
    protected Container contentPanel;
    SampleGUI() {
        // Access default content pane, AWT container
        contentPanel = this.getContentPane();
        // add Button
        JButton button=new JButton("Hello GUI !");
        contentPanel.add(button);
        // close window
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //resize frame to arrange components tightly
        pack();
        setTitle("GUI"); //set frame title
        setSize(250, 70); //set Frame size
        //display frame
        setVisible(true);
        button.addActionListener(new AddCount(button)) } }
```

```
public class SampleGUI extends JFrame {
    protected JPanel contentPanel;
    SampleGUI() {
        // set contentpanel to swing JPanel
        contentPanel = new JPanel();
        this.setContentPane(contentPanel);
        JButton button=new JButton("Hello GUI !");
        contentPanel.add(button );
        // close window
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //resize frame to arrange components tightly
        pack();
        setTitle("GUI"); //set frame title
        setSize(250, 70); //set Frame size
        //display frame
        setVisible(true);
        button.addActionListener(new AddCount(button)) } }
```

```
public class SampleGUI extends JFrame {  
    protected JPanel contentPanel;  
    SampleGUI() {  
        // set contentpanel to swing JPanel  
        contentPanel = new JPanel();  
        this.setContentPane(contentPanel);  
        JButton button=new JButton("Hello GUI !");  
        contentPanel.add(button );  
        // close window  
        setDefaultCloseOperation(JFrame.EXIT_ON_CL  
        OSE);  
        //resize frame to arrange components tightly  
        pack();  
        setTitle("GUI"); //set frame title  
        setSize(250, 70); //set Frame size  
        //display frame  
        setVisible(true);  
        button.addActionListener(new  
        AddCount(button)) } }
```

```
public class AddCount implements ActionListener  
{  
    private int count;  
    AddCount(JLabel label) {  
        this.label = label;  
        count = 0; }  
        // When button clicked  
    public void actionPerformed(ActionEvent e) {  
        // increment counter  
        count++;  
        // update button's text  
        button.setText("I am clicked " + count + " times");  
    }  
}
```

Labels, TextAreas, and Textfields

Component

- **JTextArea** is an object on the screen which is named and can hold multiline text messages.
- It is a scrollable object, so the object can hold more text than is visible on the screen at one time.
- an ActionEvent is raised when the user types the return/enter key.
- Initialization requires the number of lines of text and the number of characters per line to be specified.

```
JTextArea <variable1>;  
<variable1> = new JTextArea(<lines>, <chars>);  
add(<variable1>);
```

Example:

```
JTextArea echoArea;  
echoArea = new JTextArea(5, 40);  
add(echoArea);
```

//place a 5 line by 40 character textarea

A **JTextField** is an object which holds a single line of text. Either one can be used to gather text from the keyboard or merely to display messages from the application.

```
JTextField <variable>;  
JTextField typing;  
typing = new JTextField(40);  
add(typing);
```

Event handler

When the user types in a JTextField or JTextArea and hits the return key, the application can handle that event by writing additional code in its actionPerformed event handler:

```
public void actionPerformed (ActionEvent e) {  
    if (e.getSource() == <variable>) {  
        String s = <variable>.getText();  
        <action> } }
```

Here, <variable> refers to the name of the JTextArea or JTextField variable that was declared and placed in the frame by the constructor, like typing in the example above.

A better solution is to use an internal class that listens specifically to the JTextField typing:

In this case, the handler need not check for the source of the triggering event; it must be the user pressing the return (or enter) key in the JTextField typing.

```
private class TextHandler implements ActionListener {  
    public void actionPerformed (ActionEvent e) {  
        String s = echoArea.getText();  
        echoArea.setText(echoArea.getText() + s + "\n");    } }
```

A JLabel is an object whose string value can be placed inside a frame to label another object, such as a JTextField.

```
add(new JLabel("Fahrenheit"));
```

Buttons

A *button* is a named object on the screen which can be selected by a mouse click.

Because any number of variables can be declared with class JButton and placed in the application, button handlers are usually implemented via a separate class, rather than by the application itself.

```
JButton <variable> = new JButton(<string>);
add(<variable>);
<variable>. addActionListener( <listener> );
```

The class which handles the user selecting the button must implement the ActionListener interface. The listener class must implement an actionPerformed method to handle the button selection event:

```
JButton clearButton =new JButton("Clear");
add(clearButton);
public void actionPerformed (ActionEvent e) {
if (e.getSource() =<variable>) {
<action> }}

```

```
clearButton.addActionListener(new ClearButtonHandler( ) );

private class ClearButtonHandler implements ActionListener {
public void actionPerformed(ActionEvent e) {
repaint(); }}
```

Note that the **repaint method** is a **JPanel method**; thus, this class works as written only if it is an inner class to the application class which **extends JPanel**. However, An external class requires writing a constructor which takes the application as an argument.

Note that using an external class makes it more difficult for a listener class to determine which button was selected since the button method must be passed within the actionPerformed method.

Mouse Clicks

For the program to **handle mouse clicks**, the **MouseListener interface** must be implemented, either by *the application itself or by a mouse handler class*.

If the listener is being handled directly by the class, then the class must specify the listener in its implements clause and activate the listener, usually inside the constructor:

Application itself handle mouse clicks

```
public class MyApplication extends JPanel implements MouseListener
{
    public MyApplication( ) {
        ...
        addMouseListener(this);
    } }
```

Separate class to handle mouse clicks

```
MyApplication extends JPanel {
    public MyApplication( ) {
        ...
        addMouseListener(new MouseHandler());
    }
    private class MouseHandler implements MouseListener
    {
        ...
    }
}
```

The alternative is to use a separate class to handle mouse clicks. Commonly, this class is an inner class to the application, so that the mouse handler has access to all of the application's methods, particularly the graphics context:

An advantage to using a separate class is that Java provides a *MouseAdapter* class. This means that the separate class can extend the *MouseAdapter* class (which an application cannot do), overriding exactly the methods for which actions are to be provided. In most instances, this is usually only the *mouseClicked* or *mousePressed* method.

```
public class MyApplication extends JPanel {
    public MyApplication( ) { ...
        addMouseListener(new MouseHandler()); ...}
    private class MouseHandler extends MouseAdapter {
        public void mouseClicked(MouseEvent e) { <action>
        }
    }
}
```

Capture the x-y pixel coordinates where that event occurs on the frame.

```
public void mouseClicked(MouseEvent e) {
    x = e.getX();
    y = e.getY();
}
```

Combo Boxes

A *combo box* is an object on the screen that offers several options to be selected by the user; it is like a pull-down menu, but it can be placed anywhere within the application.

```
JComboBox <variable> ;  
JComboBox combo;
```

```
<variable>= new JComboBox( );  
<variable>.addItem(<string1>);  
...  
add(<variable>);  
<variable>. addItemListener(<listener>);
```

```
combo= new JComboBox( );  
combo.addItem("North");  
combo.addItem("East");  
combo.addItem("South");  
combo.addItem("West");  
add(combo);  
combo.addItemListener(new ComboHandler());
```

```
private class ComboHandler implements ActionListener {  
    public void actionPerformed (ActionEvent e) {  
        String s =(String)combo.getItem( );  
        if (s.equals(<string1>)) {  
            <action1> in response to a selection of <string1> }  
        else if (s.equals(<string2>)){  
            <action2> in response to a selection of <string2> }  
        } }  
}
```

Mouse Motion

```
addMouseListener(<listener>);
```

```
public void mouseDragged(MouseEvent e) { }  
public void mouseMoved(MouseEvent e) { }
```


Simple GUI Interface

Consider the design of a simple interface, in which the user can insert rectangles and place texts in arbitrary locations of the frame. The user should be able to accomplish this as simply as possible, and so providing buttons, menus, and text typing areas, as well as handling mouse-click actions on the screen is essential

combo => User can select Nothing, Rectangle, or Message from this menu.

echoArea=> A text area for reporting the most recent event that has occurred.

typing=> A text field for entering user input.