Event-Driven Programming & GUI Basics

A **Graphical User Interface (GUI)** is a **collection of classes and components** that build the visual part of an application, allowing users to interact with software through graphical elements rather than text commands. It includes **windows, forms, dialogs, and various controls like buttons, text fields, and menus**. GUIs provides an intuitive, accessible layer that enhances the user experience by visually organizing and managing the application's interactive elements, making it easier for users to navigate and control functionality.

In Java, Graphical User Interfaces (GUIs) are built using an **event-driven programming** paradigm, where the program's flow is determined by user interactions (events) such as button clicks, keystrokes, or mouse movements, rather than a linear sequence of instructions. This interaction model uses the **Delegation Event Model** to manage communication between GUI components and the code that responds to these events.

Key Components of Java Event-Driven Programming

The Java event model revolves around **three core components**:

- **Event Source:** The GUI component (e.g., a JButton, JTextField, or JFrame) that generates an event when a user interacts with it.
- **Event Object:** An object that encapsulates information about the event that occurred, such as which button was clicked, or the coordinates of a mouse click.
- **Event Listener:** An object that "listens" to specific events. **It must implement a predefined listener interface** (e.g., ActionListener, MouseListener) which contains the **event handler** methods (callbacks) that execute the desired logic when the event occurs.

How Event Handling Works

The process of handling an event involves a few steps:

1. **Implementation:** The programmer defines a class (often an inner class) that implements the appropriate event listener interface (**e.g., ActionListener**). This class provides the implementation for the required event handler method(s).
2. **Registration:** The listener object is registered with the specific event source object using a method like **addActionListener()** or **addMouseListener()**.

3. **Event Triggering:** When a user action occurs (e.g., clicking the registered button), the event source creates an event object.
4. **Notification:** The event source notifies all registered listeners by calling their specific event handler methods, passing the event object as an argument.
5. **Handling:** The event handler **method executes** the programmed response (e.g., updating a text field, changing a color, saving data).

Common Event Types and Listeners

Java provides **various event classes** and corresponding **listener interfaces**, primarily in the java.awt.event package, to handle different types of user interactions:

- **ActionEvent** (**Interface: ActionListener**): Used for high-level actions like button clicks, selecting a menu item, or pressing "Enter" in a text field.
- **MouseEvent** (**Interfaces: MouseListener, MouseMotionListener**): Handles mouse interactions like pressing, releasing, clicking, moving, or dragging the mouse.
- **KeyEvent** (**Interface: KeyListener**): Manages keyboard input, such as key presses or releases.
- **WindowEvent** (**Interface: WindowListener**): Deals with events related to a window's state, such as opening, closing, minimizing, or maximizing.

A Graphical User Interface (GUI)

While command-line interfaces are useful, one of the great advantages of the Java language is that its extensive class library makes it relatively easy to develop applications with **Graphical User Interfaces (GUIs)**. Today nearly all personal computing applications are GUI-based. GUI programming requires **event-driven programming**, which means that **GUI programs react to events that are generated mostly by the user's interactions with elements like buttons in the GUI**. We will learn how to use Java's **event model** to handle simple events. This chapter will provide coding for simple GUIs.

Graphical user interfaces (GUI) and OOP

- **GUI components or controls** are natural objects: windows, buttons, labels, text fields, etc., GUI programming is naturally asynchronous and event oriented
- In a GUI application, the **main method is responsible to initialize** and assemble the GUI and the application logic, and then to make the GUI visible

Java's GUI Components

The required classes are included in **packages** (**java.awt** and **javax.swing**). **AWT** (Abstract Window Toolkit) **components are platform dependent, as they use the operating system for rendering**, which may result in different appearances on different systems.
AWT provides the use of a minimal set of graphical interface components, which is owned by all Java-enabled platforms. It looks "equally mediocre" across all platforms. Packages:

- **java.awt**
- **java.awt.event**
- **java.awt.image**
- **java.awt.datatransfer**

Swing is a library written from scratch in Java. **It does not rely on the operating system**. It looks the same and works on all platforms. It has many components. Supports Drag & Drop as well as clipboard work. Complete support for the Unicode code page. **Swing components, such as buttons, lists, and menus, work independently of the system, providing a uniform appearance and greater functionality.**

Java GUI applications are structured using components and containers:

- **Components:** These are the **individual interactive elements of the GUI**.
  - JButton (buttons)
  - JLabel (text labels)
  - JTextField (text input fields)
  - JCheckBox and JRadioButton (selection options)
  - JTextArea (multi-line text input)
- **Containers: These objects hold and organize other components**.
  - **JFrame**: A top-level, independent window that acts as the main application window with a title bar and window controls (like close/minimize buttons).

    - **JPanel**: A general-purpose container used to **organize components within a window or even within other panels**, often for layout management.

Swing Package in java

Forms can be created dynamically by writing their own code or statically using programming tools.

Dynamic form creation.
It consists of **making objects** that represent components on a form. We use the following objects:

- Font Object
- Color Object.
- JFrame Object
- Frame Layout and Centering

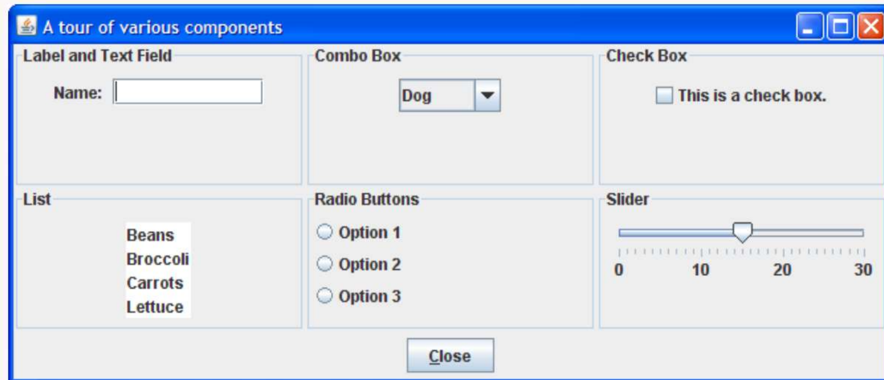
as well as swing controls:

- JButton Control
- JLabel Control
- JTextField Control
- JTextArea Control
- JCheckBox Control
- JRadioButton Control.
- JPanel Control
- JList Control
- JScrollPane Control
- JComboBox Control

Notably:

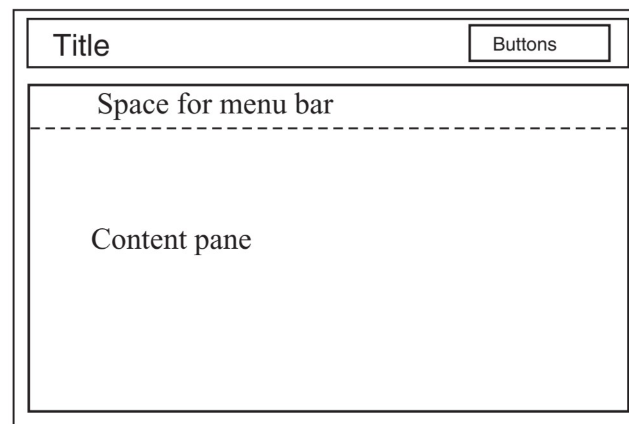
- **JFrame**: The main container that hosts the application window.
- **JPanel**: A container for organizing other components and often used for **layout management**.
- **Swing Controls**: Include interactive elements (JButton, JCheckBox, JRadioButton, etc.) that can trigger events.

Some common GUI components are:
– **buttons**, **labels**, **text fields**, **check boxes**, **radio buttons**, **combo boxes**, and **sliders**.



JFrame class

Each graphics application in java must have **at least one dialog box**. To create such a window, a **JFrame class object** must be created (the class is in the swing package in the class library).



Structure of a frame. The **title text** can be set. The **position** and **appearance** of the buttons depend on the operating system. Further graphical components can be embedded into the content pane

In doing so, this package must be imported into the application with the command:

import javax.swing.JOptionPane;

javax is a package containing a **subpackage swing**, while **JFrame** is a **class** that is imported into the program. Otherwise, all the components that need to be put on the dialog later are mostly in the swing package. **JFrame() is the default constructor**. The object that represents the dialog is called win and is defined as follows:

JFrame win = new JFrame();

Next, the **attributes of the win object** need to be defined. Attribute values are set using **"setter"**, the methods they employ with set. For example, **to set the title we use setter setTitle**:

win.setTitle("Title of graphics");

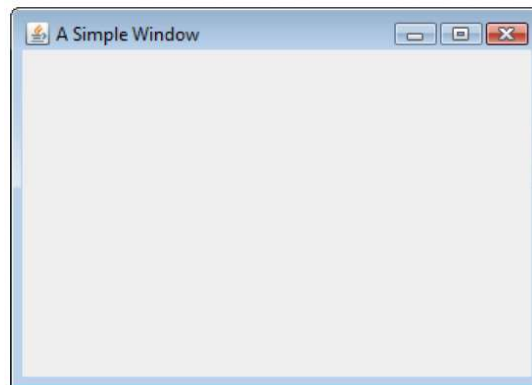
For the **JFrame dialog to be displayed**, at **least the following must be set**:

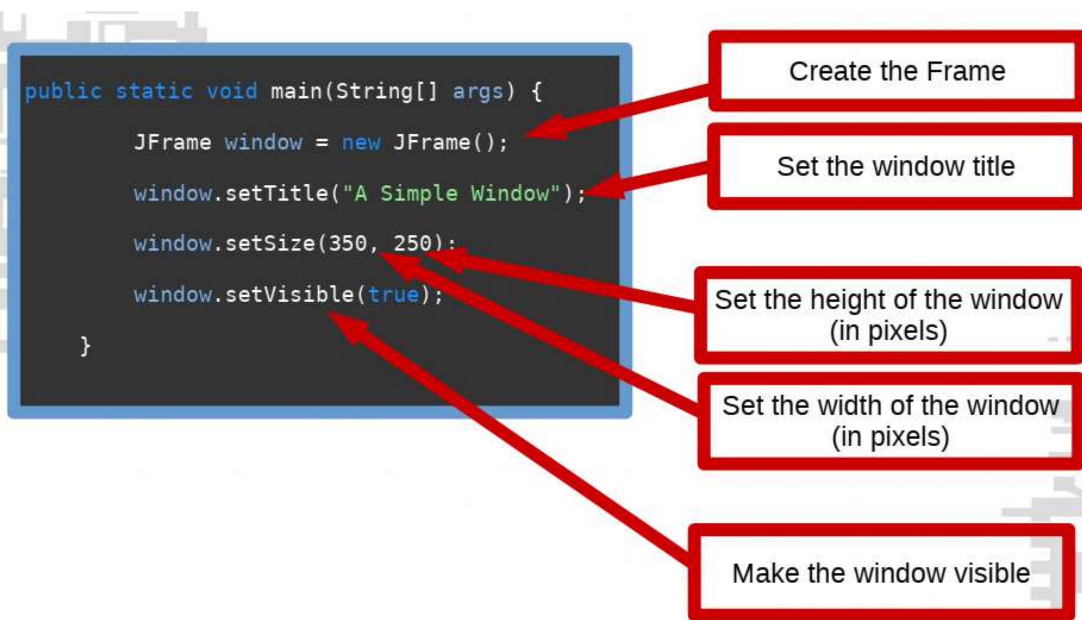
- Size (**setSize** method)
- Behavior of the application after clicking the close button (**x-button in the upper left corner of the window**), the method is **setDefaultCloseOperation**
- **visibility**, using the **setVisible (true)** method

The current code looks like: **Swing User Package. Creating a JFrame class object**

Jframe Window

A basic window may look like this:





```
import javax.swing.JOptionPane;
import javax.swing.JFrame; //Import of class JFrame

public class SolarSystemFrame {

public static void main(String[] args) {
//Creating object
JFrame win = new JFrame();
//Sets attributes
win.setTitle("Solar system"); //Sets the title
win.setSize(800, 600); //Setting the window size to 800 * 600px

//clicking the close button exits the program
win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//visibility is set, the JFrame object, is invisible by default
win.setVisible(true);//the last command will remain
} }
```

The Color class defines several standard color constants.

Constants: Color.BLACK, Color.BLUE, Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE, and Color.YELLOW.

The **setLayout** method is responsible for the **layout of the panel** and should be passed to an object of the class inherited from the **LayoutManager** class. It is the object responsible for the arrangement of components per panel. If this object is null, then it will be arranged by defining the coordinates within which the component will be located (**setBounds** method).

The **setContentPane** method connects a **panel** to a **win** object.

Next, we create a **label for the text** (object of the **class JLabel**), place text on it, define borders i.e., the rectangle within which it will be placed, and finally we add it to the panel. A **rectangle is defined by 4 numbers** where the **first two** refer to the **position of the rectangle** defined by the x and y coordinates of the point of the upper left corner of the rectangle, while the remaining **two numbers represent the size**, i.e., the width and height of the rectangle.

Example

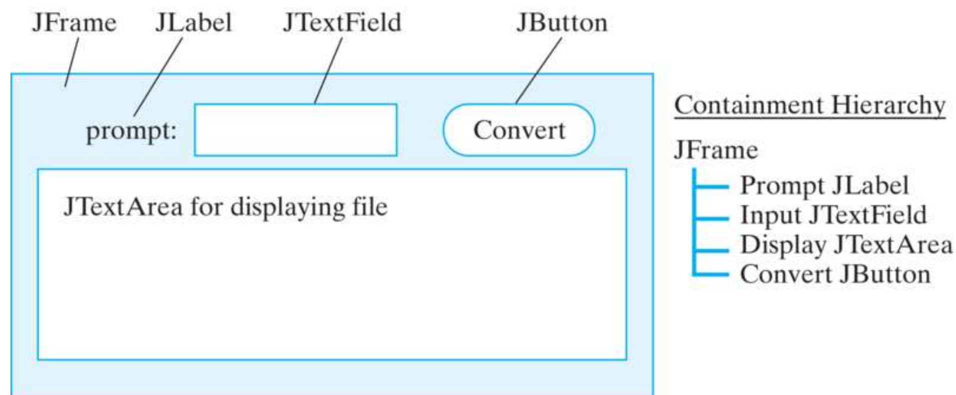
```
import java.awt.*;
import javax.swing.*;

public class Main {

public static void main(String[] args) {
//Creating object
JFrame win = new JFrame();
//Methods
win.setTitle("Solar system"); //Sets the title
win.setSize(400, 400); //Setting the window size to 800 * 600px
//clicking the close button exits the program
win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// JPanel used to group and organize other components (like buttons, text fields, and
other panels) within a graphical user interface (GUI).

JPanel pan = new JPanel();
pan.setBackground(Color.BLUE); //Panel color
win.setContentPane(pan); //Connecting the panel to the win object

//Makes a label with text

JLabel textLabel = new JLabel();//Defining a label object
textLabel.setText("Solar system");
textLabel.setForeground(Color.white);//Font color

Font f = new Font("Vedrana", Font.BOLD, 20);
textLabel.setFont(f);
/*defines a rectangle that represents the boundary within which the component is
placed adds a label to the panel*/

textLabel.setBounds(5, 5, 300, 30);
pan.add(textLabel);
win.setContentPane(pan); //Connecting the panel to the win object visibility is set,
the JFrame object is invisible by default
```

```
//visibility is set, the JFrame object, is invisible by default
win.setVisible(true);//the last command will remain
} }
```

- **Font**: The class (java.awt.Font) used to represent fonts.
- **"Verdana"**: The font name. It is recommended to use logical font names (Serif, SansSerif, Monospaced, Dialog, DialogInput) for better cross-platform compatibility, though physical font names can be used.
- **Font.BOLD**: Style constant, which can be Font.PLAIN, Font.BOLD, Font.ITALIC, or Font.BOLD | Font.ITALIC.
- **20**: Point size of the font.

The Java library comes with **two separate but interrelated packages of GUI components**, the **older java.awt package** and the **newer javax.swing package**. For the most part, the **Swing** classes supersede the **AWT** classes. For example, the java.awt.Button class is superseded by the javax.swing.JButton class, and the java.awt.TextField class is superseded by the javax.swing.JTextField class. As these examples show, the **newer Swing components add an initial 'J' to the names of their corresponding AWT counterparts**.

Figure below illustrates how some of the main components appear in a GUI interface. As shown there, **a JLabel is simply a string of text displayed on the GUI**, used here as a prompt. A **JTextField is an input element that can hold a single line of text**. In this case, the **user has input his name**. A **JTextArea is an output component that can display multiple lines of text**. In this example, it displays a simple greeting. A **JButton is a labeled control element**, which is an element that allows the user to control the interaction with the program. In this example, the user will be greeted by the name input into the JTextField, whenever the JButton is clicked. As we will learn, clicking on the JButton causes an event to occur, which leads the program to take the action of displaying the greeting. Finally, all these components are contained in a JFrame, which is a top-level container. A **container** is a GUI component that can contain other GUI components.



Various GUI components from the javax.swing package

How to Listen to Events on Buttons in Java

To execute a method when a **button is clicked** in a Java GUI, you must attach an **ActionListener to the button**.

There are **three steps programmers** need to follow to **listen to an event on a button**.

- **First**, you need to **implement the ActionListener interface** on your event handling class. You could also **extend** a class that **implements ActionListener** instead. Here is how that looks in Java code:

```
class EventClass implements ActionListener {
        //some code here
    }
```

- **Second**, you need to **add an instance of the event handler** as an action listener to one or more components using the **addActionListener() method**:

```
GuiComponent.addActionListener(EventClassInstance);
```

The **addActionListener method** in Java is used to **register an object** that will be notified when a user performs a specific action on a component, such as clicking a button or pressing Enter in a text field.

- **The final step** is to provide an implementation of the **actionPerformed(ActionEvent e)** method, which performs some action whenever an event is registered on a component. This method is the only method in the **ActionListener interface** and it is always called when an action is performed.

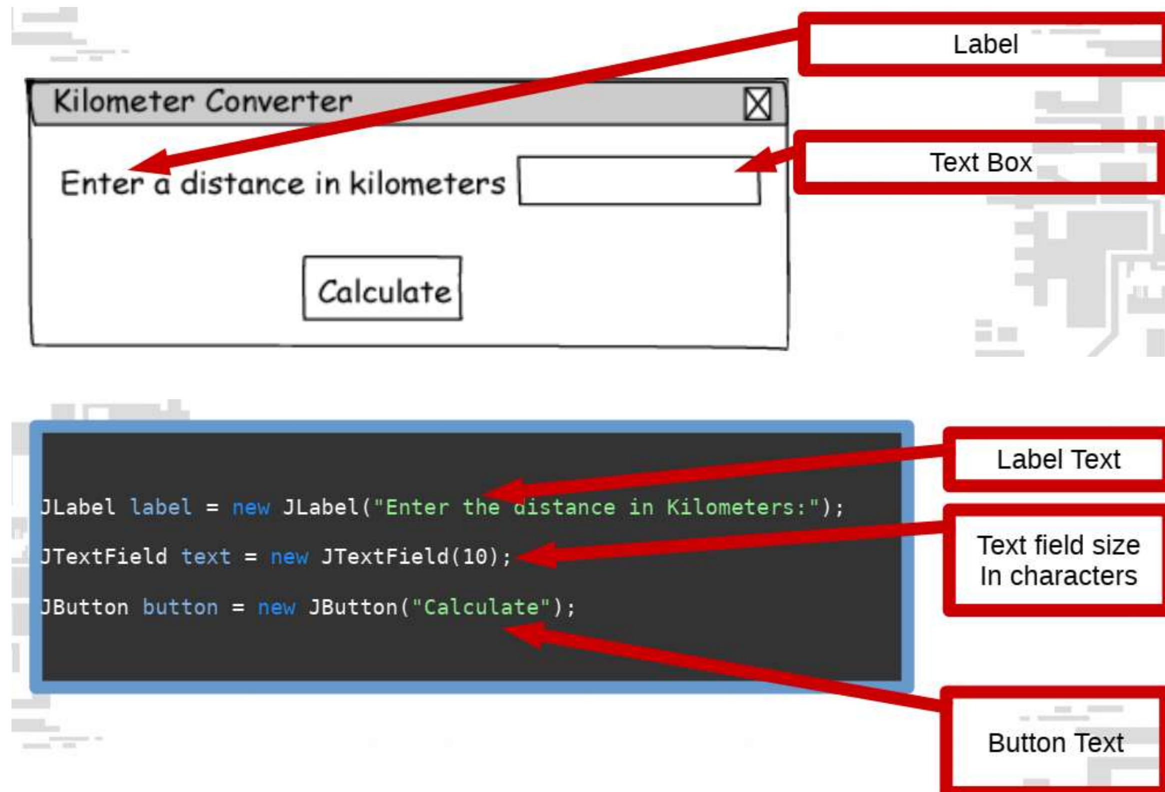
To understand the label and textbox, the program could look like this

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

public class JJJ {
    public static void main(String[] args) {
        // 1. Create the frame
        JFrame frame = new JFrame("Label and Button Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 150);
        /* setLayout(new FlowLayout())  is used to assign the FlowLayout manager to
a container (such as a JFrame or JPanel), which arranges components in a left-to-
right flow, wrapping to the next line if space is insufficient. new FlowLayout(): The
default, center-aligned, with 5-pixel gaps.*/

        frame.setLayout(new FlowLayout());

        // 2. Create the label
        JLabel label = new JLabel("Click the button to change me.");

        // 3. Create the button
        JButton button = new JButton("Click Me!");

        // 4. Add action listener to button
        button.addActionListener(new ActionListener()) {
            public void actionPerformed(ActionEvent e) {
                label.setText("Button was clicked!");  } }

        // 5. Add components to frame
       //frame.add(label)    used    to place    a JLabel component    inside
a JFrame container, making the label visible when the frame is displayed.

        frame.add(label);
        frame.add(button);

        // 6. Display the frame
        frame.setVisible(true); }}
```

Example

Hints

- **JTextField**: This is the class for the text field component, which is part of the javax.swing package.
- **text**: This is the variable name (an identifier) used to reference the newly created text field object in your program.
- **new JTextField(10)**: This calls a constructor of the JTextField class to create a new object.
    - The integer argument 10 specifies the number of columns, which is a hint to the layout manager for the preferred width in characters. It does not limit the amount of text a user can enter.




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

public class BuLabel {
   public static void main(String[] args) {
```

```
// 1. Create the frame
  JFrame window = new JFrame();
  window.setTitle("Kilometer convertor");
  window.setSize(350,250);
  window.setVisible(true);
  window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// 2. Create the label
  JLabel label = new JLabel("Enter the distance in kilometer");

// 3. Create the textfield
  JTextField text = new JTextField(10);

// 4. Create the button
  JButton button = new JButton("Calcualtor");

// 5. Create the panel
JPanel  panel = new JPanel();

  panel.add(label);
  panel.add(text);
  panel.add(button);
  window.add(panel);}}
```

Java Code Example for Button Click Events

The Java code example below **displays the number of clicks a user has so far made when they click Button1**:

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

class ClicksCount implements ActionListener{
int count = 0;// store number of clicks

ClicksCount(){
JFrame frame = new JFrame();
```

JButton button1 = new JButton("Button1");
JButton button2 = new JButton("Button2");

// **this** refers to the current class **implementing ActionListener**
button1.addActionListener(this);
frame.setLayout(new BoxLayout(frame.getContentPane(), BoxLayout.**Y_AXIS**));
frame.add(button1);
frame.add(button2);

// getRootPane().setDefaultButton used to specify which JButton in a top-level container should be the **default button**.

frame.getRootPane().setDefaultButton(button1); // sets default button
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(450,450);
frame.setVisible(true); }

public void **actionPerformed(ActionEvent e)** {
count++;
System.out.println("You have clicked the ACTIVE button " + count + " times");}

public static void main(String args[]){
ClicksCount Clicks = new ClicksCount(); } }

Output
You have clicked the ACTIVE button 1 times