



GUI PROGRAMMING USING SWING

Vijay Kumar Meena
Assistant Professor
KIIT University

ACKNOWLEDGEMENT

Slides in this presentation has been taken as it is or inspired from various resources. Some of these resources are -

- Course material of the course CECS 475 taught by Phuong Nguyen at California State University, Long Beach
- Swing tutorial on website <https://www.javatpoint.com/java-swing>

These slides are purely for educational purpose.

Thank You!

WHAT IS GUI?

- GUI (Graphical User Interface) allows users to interact with computers using visual icons. For example, your computer's file manager.
- GUIs include several visual components like buttons, icons, input fields, etc.
- GUIs enhances user experience by providing an easy way to interact with computers. Without GUIs users had to use terminals to interact with computers which might seem complex for beginners.

GUI PROGRAMMING WITH JAVA

- Initially, Java GUIs were built with components from the *Abstract Window Toolkit (AWT)* in package *java.awt*
- When a Java application with an AWT GUI executes on different Java platforms, the application's GUI components display differently on each platform.
- Consider an application that displays an object of type `Button` (package `java.awt`). On a computer running the Microsoft Windows operating system, the `Button` will have the same appearance as the buttons in other Windows applications. Similarly, on a computer running the Apple Mac OS X operating system, the `Button` will have the same look and feel as the buttons in other Macintosh applications.

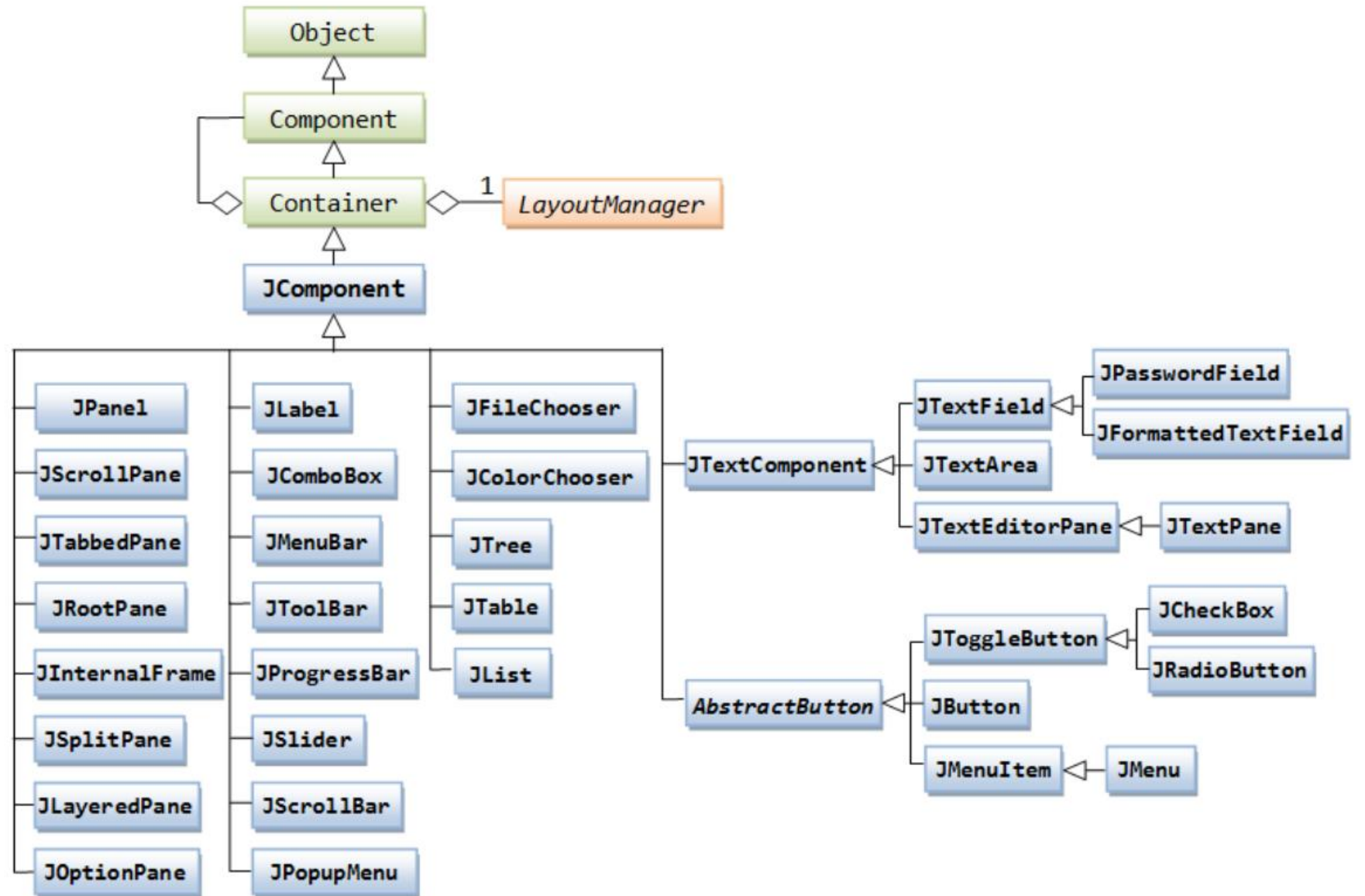
INTRO TO SWING

- Java swing is a powerful toolkit for creating graphical user interfaces (GUIs) in Java.
- It provides a wide range of components, such as buttons, text fields, and menus, that can be used to create attractive and functional GUIs.
- Swing is platform-independent, which means that your GUIs will look the same on any platform, including Windows, Mac OS, and Linux.
- Swing components are implemented in Java, so they are more portable and flexible than AWT components.

COMPONENTS & CONTAINERS

- A Swing GUI consists of two key items - *components* and *containers*.
- An component is an independent visual control, such as a push button or slider.
- A container is a special type of component that is designed to hold other components.
- In order for a component to be displayed, it must be held within a container. Therefore, all Swing GUIs will have at least one container.
- *JFrame* is the most common top most container used in Swing applications.

COMPONENTS





Buttons



Combo Box



List



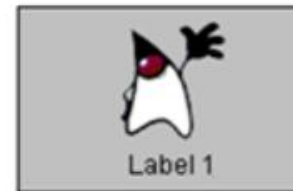
TextField



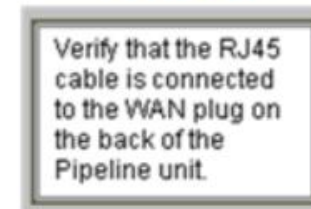
Slider



Menu



Label



Text Area



Tool Tip



Progress Bar



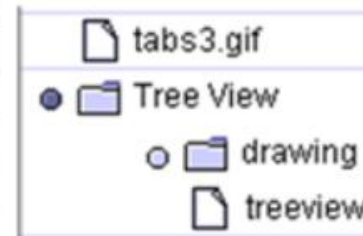
File Chooser



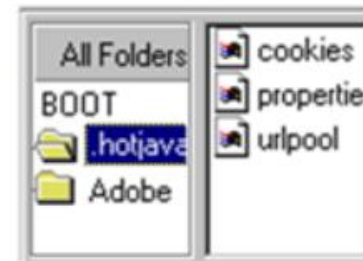
Color Chooser

First Na...	Last Name
Mark	Andrews
Tom	Ball
Alan	Chung
Jeff	Dinkins

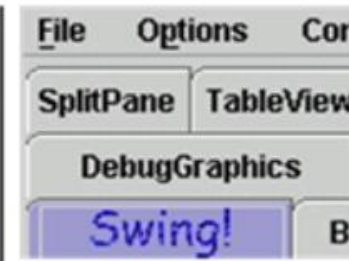
Table



Tree

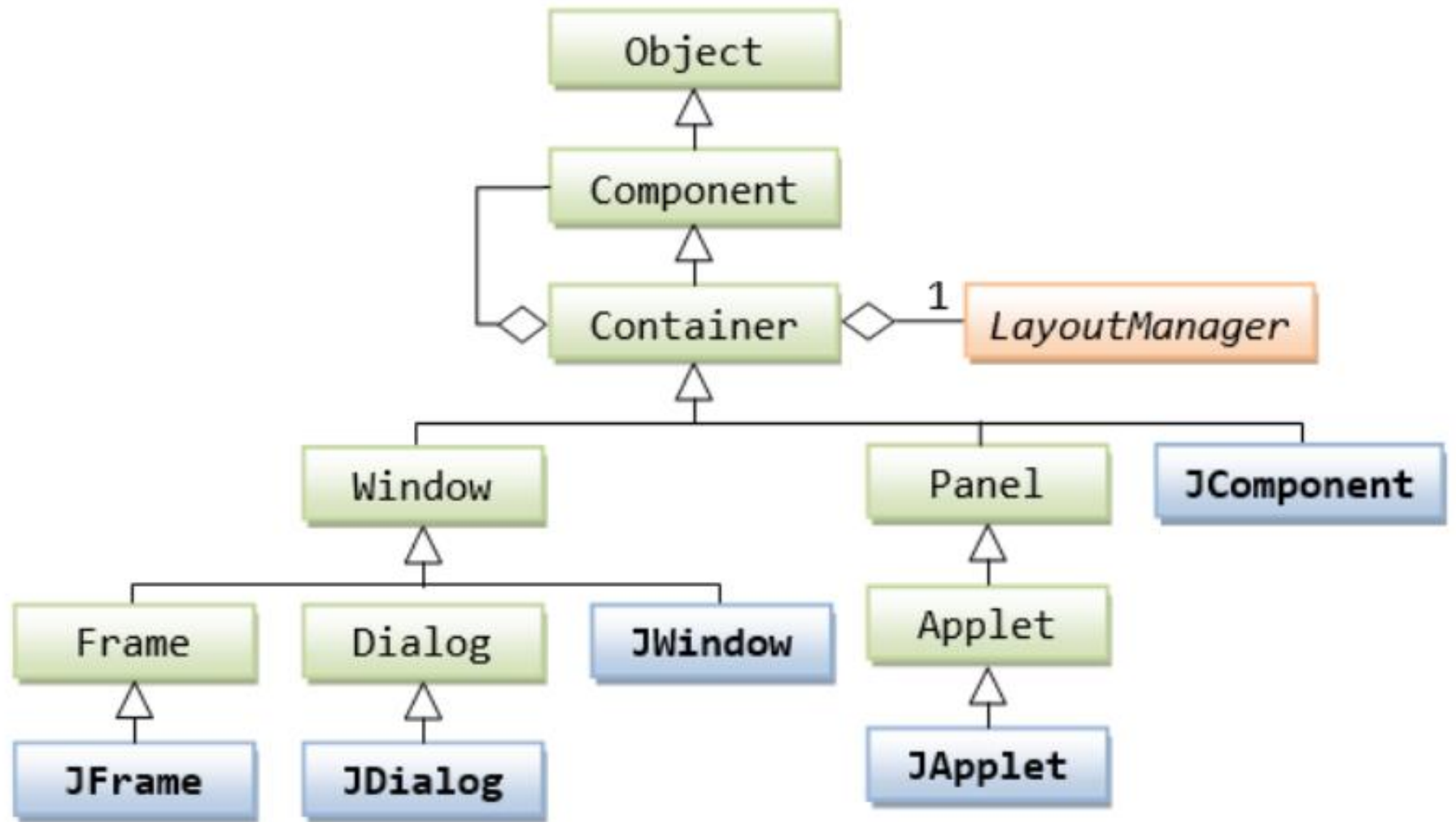


Split Pane



Tabbed Pane

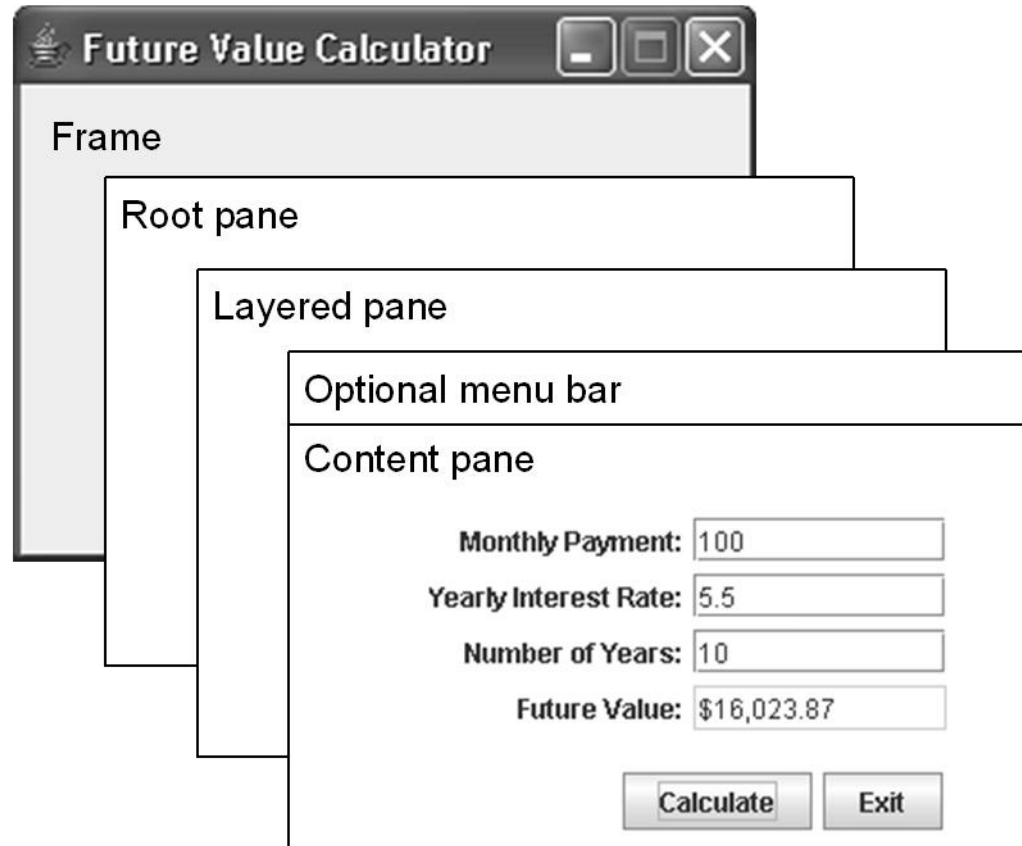
CONTAINERS



THE TOP-LEVEL CONTAINER PANES

- Each top-level container defines a set of panes. At the top of the hierarchy is an instance of JRootPane.
- JRootPane is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar.
- The panes that comprise the root pane are called *the glass pane*, *the content pane*, and *the layered pane*.
- Most of our interaction will be with content pane where swing component will be added to create GUI.

THE TOP-LEVEL CONTAINER PANES



SIMPLE GUI IN SWING

J HelloWorld.java

```
1  #!/ Import class defined in swing package
2  import javax.swing.*;
3
4  class HelloWorld{
5      public static void main(String[] args){
6          /* Create object of JFrame class which contains the GUI
7             JFrame frame = new JFrame("HelloWorld");
8             frame.setSize(500, 500);
9             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10
11             JLabel label = new JLabel("Hello, World!");
12             frame.add(label);
13             frame.setVisible(true);
14         }
15     }
```

PROBLEMS

TERMINAL

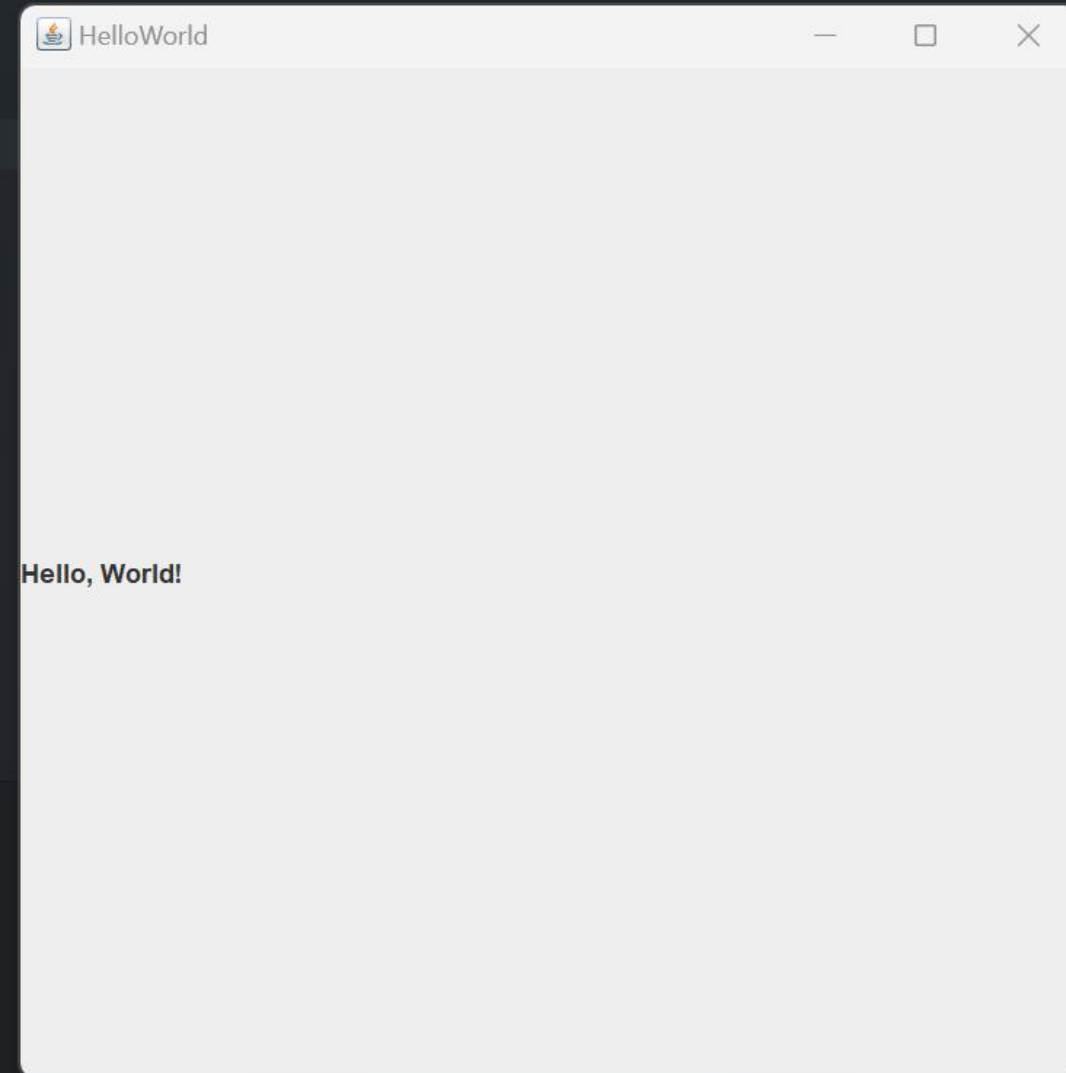
DEBUG CONSOLE

OUTPUT

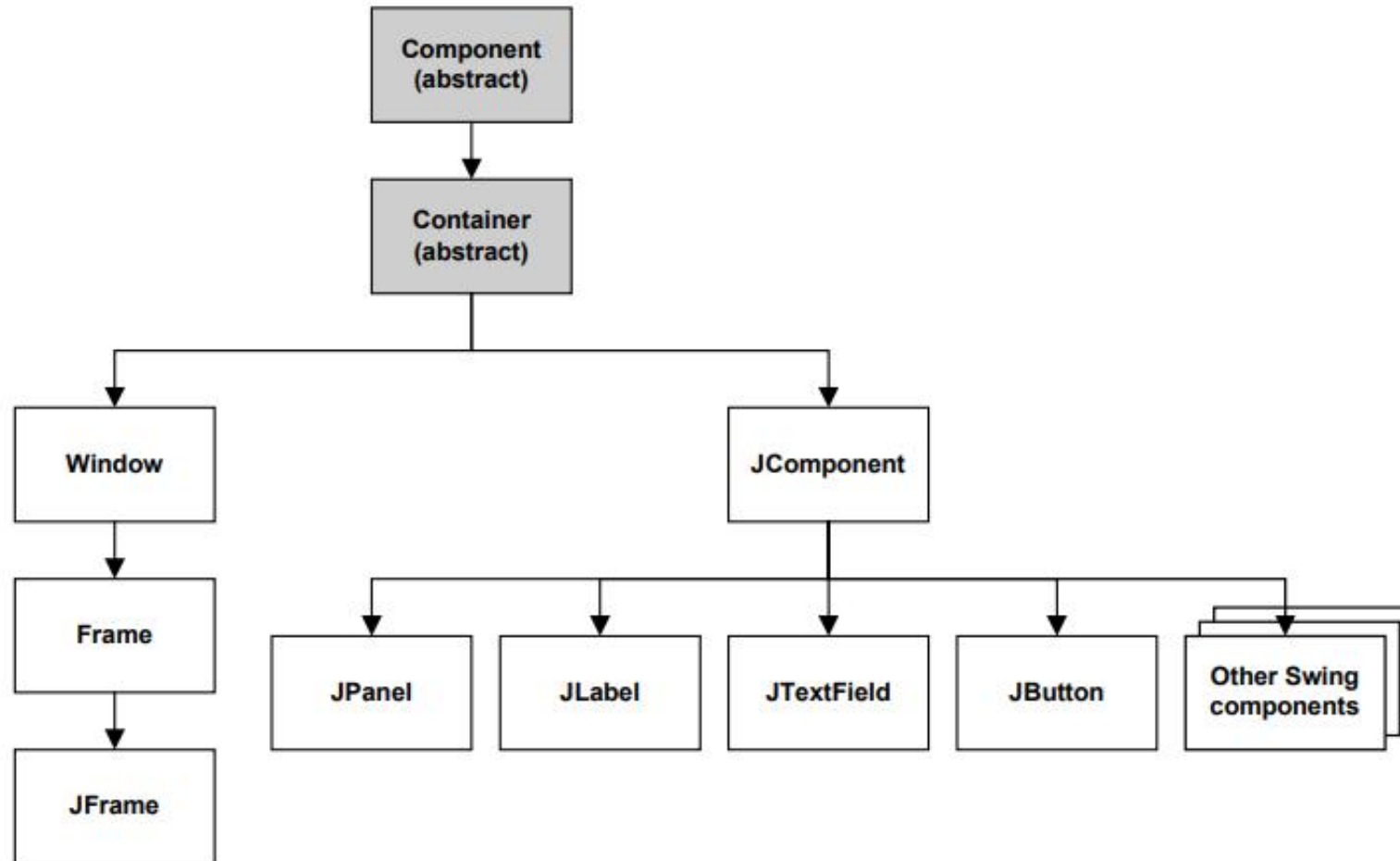
PORTS

D:\OOPJ\Codes\Slides\Swing>javac HelloWorld.java

D:\OOPJ\Codes\Slides\Swing>java HelloWorld



SWING COMPONENTS HEIRARCHY



SWING COMPONENT CLASSES

- **Component** - An abstract base class that defines any object that can be displayed.
- **Container** - An abstract class that defines any component that can contain other components.
- **Window** - The AWT class that defines a window without a title bar or border.
- **Frame** - The AWT class that defines a window with a title bar and border.
- **JFrame** - The Swing class that defines a window with a title bar and border.
- **JComponent** - A base class for Swing components such as JPanel, JButton, JLabel and JTextField.

SWING COMPONENT CLASSES

- **JPanel** - The Swing class that defines a panel which is used to hold other components. JPanel is used to group component inside a JFrame.
- **JLabel** - The Swing class which is used to display text or image on the GUI.
- **TextField** - The Swing class which is used to take input from the user in GUI application.
- **Button** - The Swing class that defines a button.

SET METHODS OF THE COMPONENT CLASS

Method	Description
<code>setSize(intWidth, intHeight)</code>	Resizes this component using two int values.
<code>setLocation(intX, intY)</code>	Moves this component to the x and y coordinates specified by two int values.
<code>setBounds(intX, intY, intWidth, intHeight)</code>	Moves and resizes this component.

Notes

- When you set the location and size of a component, the unit of measurement is *pixels*, which is the number of dots that your monitor uses to display a screen.
- The preferred way to set the location of a component is to use a layout manager.

SET METHODS OF THE COMPONENT CLASS

Method	Description
setEnabled (boolean)	If the boolean value is true, the component is enabled. If false, the component is disabled, so it doesn't respond to user input or generate events.
setVisible (boolean)	Shows this component if the boolean value is true. Otherwise, hides it.
setFocusable (boolean)	Determines whether or not this component can receive the focus.
setName (String)	Sets the name of this component to the specified string.

GET METHODS OF THE COMPONENT CLASS

Method	Description
<code>getHeight()</code>	Returns the height of this component as an int.
<code>getWidth()</code>	Returns the width of this component as an int.
<code>getX()</code>	Returns the x coordinate of this component as an int.
<code>getY()</code>	Returns the y coordinate of this component as an int.
<code>getName()</code>	Returns the name of this component as a String.

SOME OTHER METHODS OF THE COMPONENT CLASS

Method	Description
<code>isEnabled()</code>	Returns true if the component is enabled.
<code>isVisible()</code>	Returns true if the component is visible.
<code>requestFocusInWindow()</code>	Moves the focus to the component.

COMMON METHODS OF THE FRAME CLASS

Method	Description
<code>setTitle(String)</code>	Sets the title to the specified string.
<code>setResizable(boolean)</code>	If the boolean value is true, the user can resize the frame.

SETDEFAULTCLOSEOPERATION() METHOD

Method	Description
<code>setDefaultCloseOperation(action)</code>	Sets the default close action for the frame.

Constant	Description
<code>JFrame.EXIT_ON_CLOSE</code>	Exits the application when the user closes the window.
<code>WindowConstants.DO_NOTHING_ON_CLOSE</code>	Provides no default action, so the program must explicitly handle the closing event.
<code>WindowConstants.HIDE_ON_CLOSE</code>	Hides the frame when the user closes the window. This is the default action.
<code>WindowConstants.DISPOSE_ON_CLOSE</code>	Hides and disposes of the frame when the user closes the window.

LAYOUT MANAGERS

- Positioning components inside a container using methods like *setSize()*, *setLocation()*, *setBounds*, etc. is a troublesome task as you need to give exact coordinates in terms of pixels.
- The LayoutManagers are used to arrange components in a particular manner.
- *LayoutManager* is an interface that is implemented by all the classes of layout managers.
- Some common layout manager classes are -

- <code>java.awt.BorderLayout</code>	- <code>java.awt.FlowLayout</code>
- <code>java.awt.GridLayout</code>	- <code>java.awt.CardLayout</code>
- <code>java.awt.GridBagLayout</code>	
- <code>javax.swing.BoxLayout</code>	- <code>javax.swing.GroupLayout</code>
- <code>javax.swing.ScrollPaneLayout</code>	- <code>javax.swing.SpringLayout</code>

BORDER LAYOUT

- The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only.
- It is the default layout of a frame or window.
- The BorderLayout provides five constants for each region -
 - `public static final int NORTH`
 - `public static final int SOUTH`
 - `public static final int EAST`
 - `public static final int WEST`
 - `public static final int CENTER`

BORDER LAYOUT

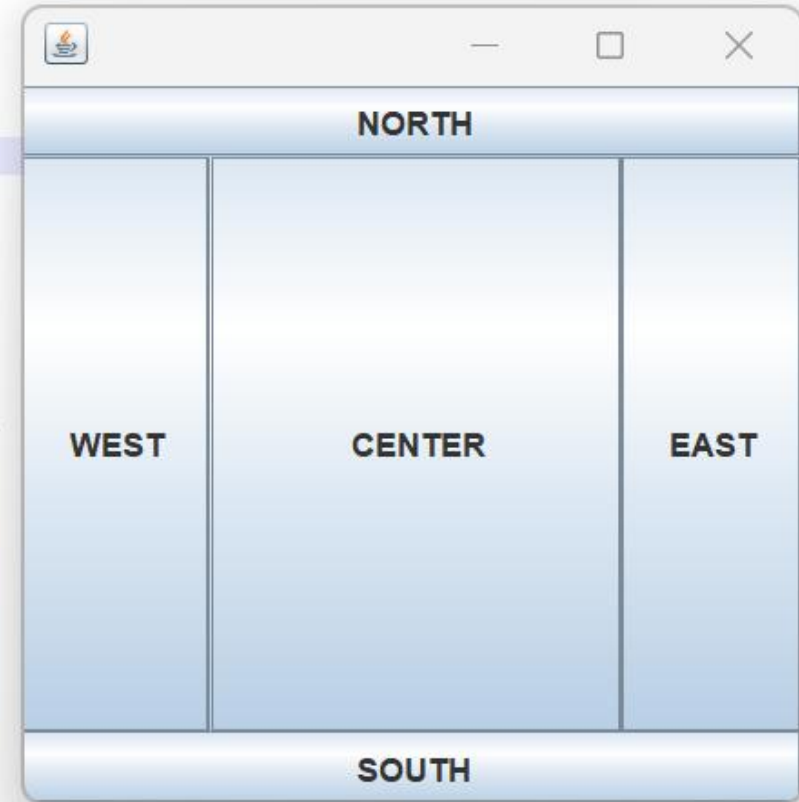
- BorderLayout class provides two constructors also -
 - **BorderLayout():** creates a border layout but with no gaps between the components.
 - **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.
- If you add more than one component in same direction of BorderLayout then only latest component is shown.
- The add() method of the JFrame class can work even when we do not specify the region. In such a case, only the latest component added is shown in the frame, and all the components added previously get discarded. The latest component covers the whole area.


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

class BorderLayoutDemo
{
    JFrame f;
    BorderLayoutDemo()
    {
        f = new JFrame();
        f.setLayout(new BorderLayout());
        // creating buttons
        JButton b1 = new JButton("NORTH"); // the button will be labeled as NORTH
        JButton b2 = new JButton("SOUTH"); // the button will be labeled as SOUTH
        JButton b3 = new JButton("EAST"); // the button will be labeled as EAST
        JButton b4 = new JButton("WEST"); // the button will be labeled as WEST
        JButton b5 = new JButton("CENTER"); // the button will be labeled as CENTER

        f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction
        f.add(b2, BorderLayout.SOUTH); // b2 will be placed in the South Direction
        f.add(b3, BorderLayout.EAST); // b2 will be placed in the East Direction
        f.add(b4, BorderLayout.WEST); // b2 will be placed in the West Direction
        f.add(b5, BorderLayout.CENTER); // b2 will be placed in the Center

        f.setSize(300, 300);
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new BorderLayoutDemo();
    }
}
```



```

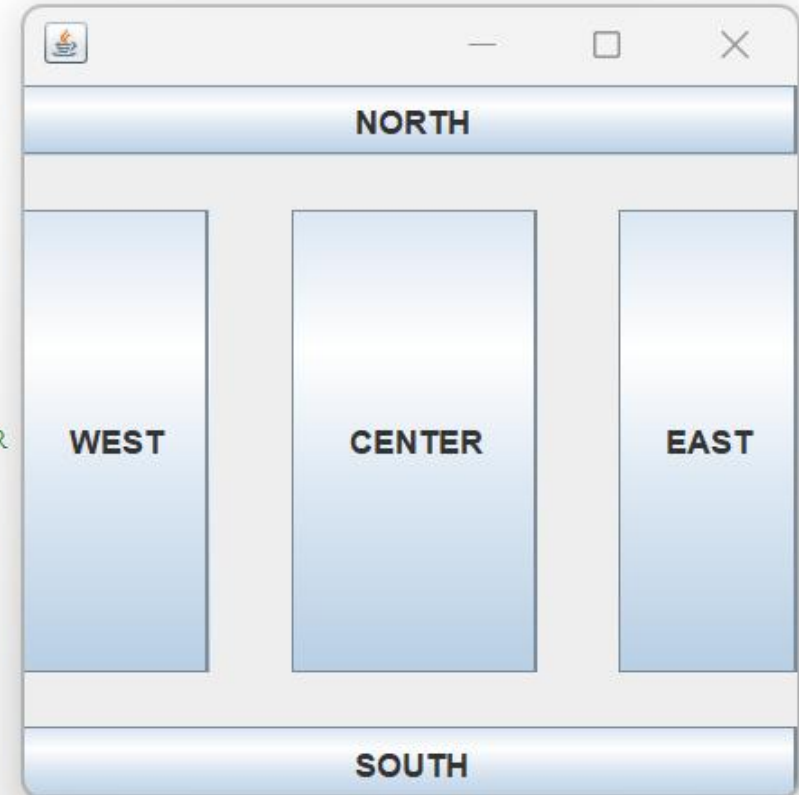
import java.awt.*;
import javax.swing.*;

class BorderLayoutDemo
{
    JFrame f;
    BorderLayoutDemo()
    {
        f = new JFrame();
        f.setLayout(new BorderLayout(30, 20));
        // creating buttons
        JButton b1 = new JButton("NORTH"); // the button will be labeled as NORTH
        JButton b2 = new JButton("SOUTH"); // the button will be labeled as SOUTH
        JButton b3 = new JButton("EAST"); // the button will be labeled as EAST
        JButton b4 = new JButton("WEST"); // the button will be labeled as WEST
        JButton b5 = new JButton("CENTER"); // the button will be labeled as CENTER

        f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction
        f.add(b2, BorderLayout.SOUTH); // b2 will be placed in the South Direction
        f.add(b3, BorderLayout.EAST); // b2 will be placed in the East Direction
        f.add(b4, BorderLayout.WEST); // b2 will be placed in the West Direction
        f.add(b5, BorderLayout.CENTER); // b2 will be placed in the Center

        f.setSize(300, 300);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        new BorderLayoutDemo();
    }
}

```



GRID LAYOUT

- The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle.
- Constructors -
 - **GridLayout()**: creates a grid layout with one column per component in a row.
 - **GridLayout(int rows, int columns)**: creates a grid layout with the given rows and columns but no gaps between the components.
 - **GridLayout(int rows, int columns, int hgap, int vgap)**: creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

```

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

class GridLayoutExample
{
    JFrame frameObj;

    // constructor
    GridLayoutExample()
    {
        frameObj = new JFrame();

        // creating 9 buttons
        JButton btn1 = new JButton("1");
        JButton btn2 = new JButton("2");
        JButton btn3 = new JButton("3");
        JButton btn4 = new JButton("4");
        JButton btn5 = new JButton("5");
        JButton btn6 = new JButton("6");
        JButton btn7 = new JButton("7");
        JButton btn8 = new JButton("8");
        JButton btn9 = new JButton("9");

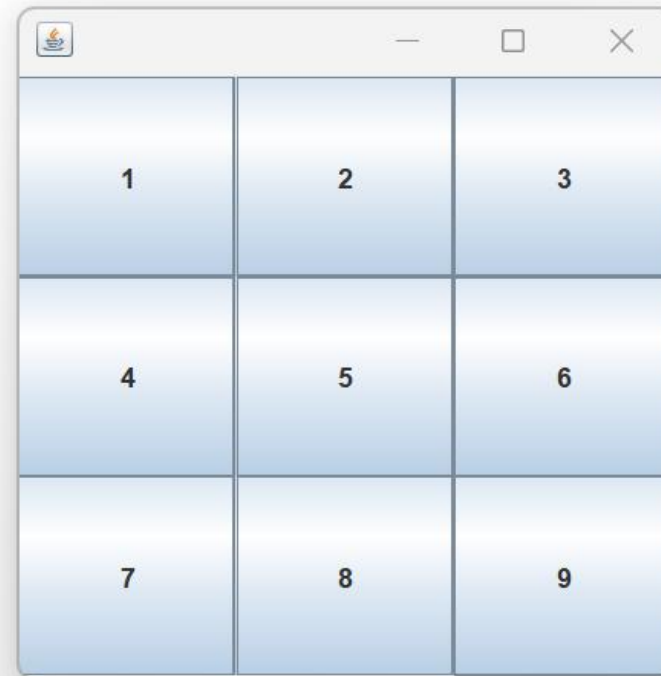
        frameObj.add(btn1); frameObj.add(btn2); frameObj.add(btn3);
        frameObj.add(btn4); frameObj.add(btn5); frameObj.add(btn6);
        frameObj.add(btn7); frameObj.add(btn8); frameObj.add(btn9);

        frameObj.setLayout(new GridLayout(3, 3));

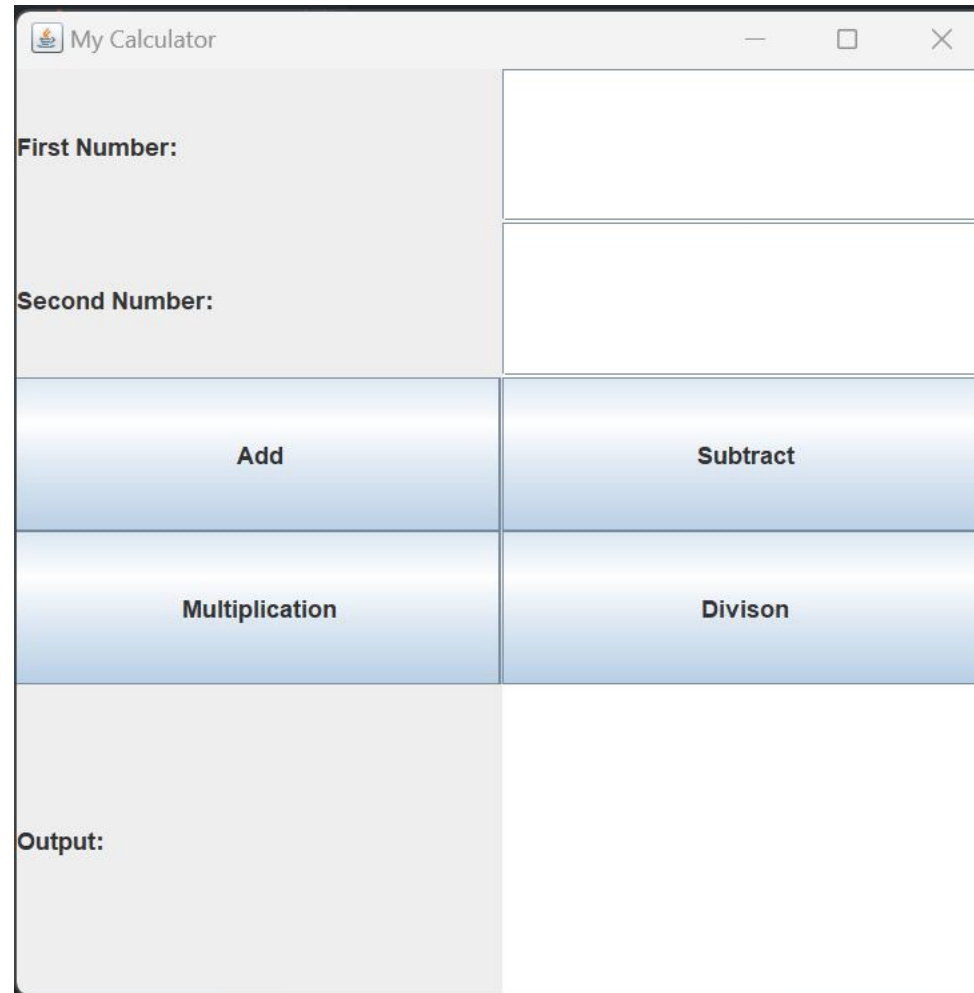
        frameObj.setSize(300, 300);
        frameObj.setVisible(true);
        frameObj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String args[])
    {
        new GridLayoutExample();
    }
}

```



EXERCISE: CREATE FOLLOWING GUI FOR CALCULATOR



The image shows a window titled "My Calculator" with a standard title bar. The window is divided into several sections:

- First Number:** A text label followed by an empty input field.
- Second Number:** A text label followed by an empty input field.
- Operations:** Four buttons arranged in a 2x2 grid:
 - Add**
 - Subtract**
 - Multiplication**
 - Divison**
- Output:** A text label followed by a large empty area for displaying the result.

FLOW LAYOUT

- The Java `FlowLayout` class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.
- Fields of `FlowLayout` class -
 - `public static final int LEFT`
 - `public static final int RIGHT`
 - `public static final int CENTER`
 - `public static final int LEADING`
 - `public static final int TRAILING`

FLOWLAYOUT

- Constructors of FlowLayout class -
 - **FlowLayout()**: creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
 - **FlowLayout(int align)**: creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
 - **FlowLayout(int align, int hgap, int vgap)**: creates a flow layout with the given alignment and the given horizontal and vertical gap.

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

class MyFlowLayout{
    JFrame f;
    MyFlowLayout(){
        f=new JFrame();

        JButton b1=new JButton("1");
        JButton b2=new JButton("2");
        JButton b3=new JButton("3");
        JButton b4=new JButton("4");
        JButton b5=new JButton("5");

        // adding buttons to the frame
        f.add(b1); f.add(b2); f.add(b3); f.add(b4); f.add(b5);

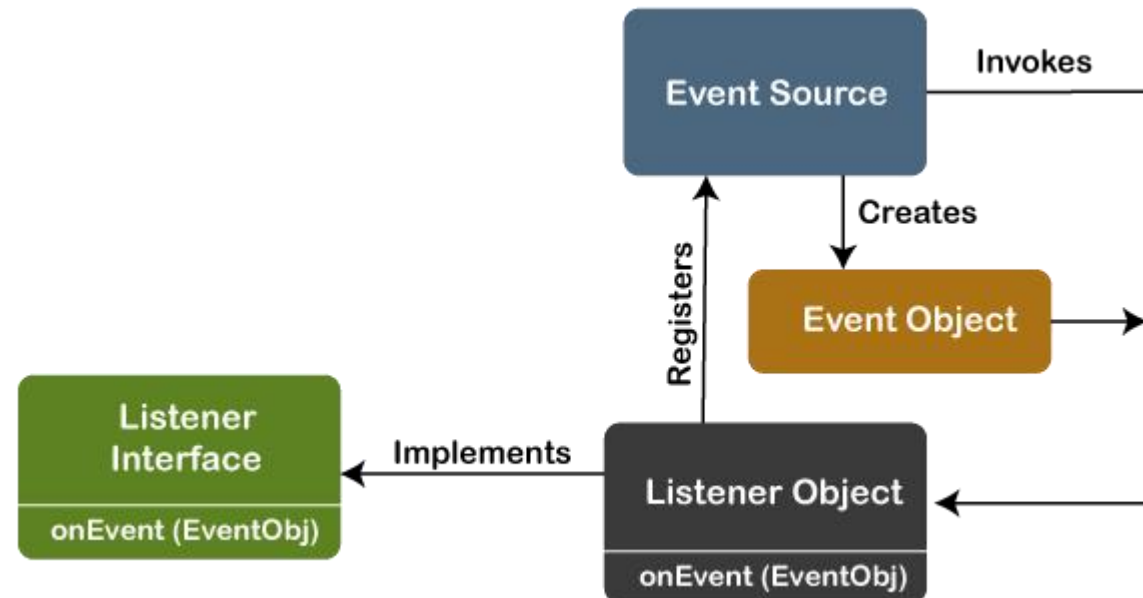
        // setting flow layout of right alignment
        f.setLayout(new FlowLayout(FlowLayout.RIGHT));

        f.setSize(300,300);
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new MyFlowLayout();
    }
}
```

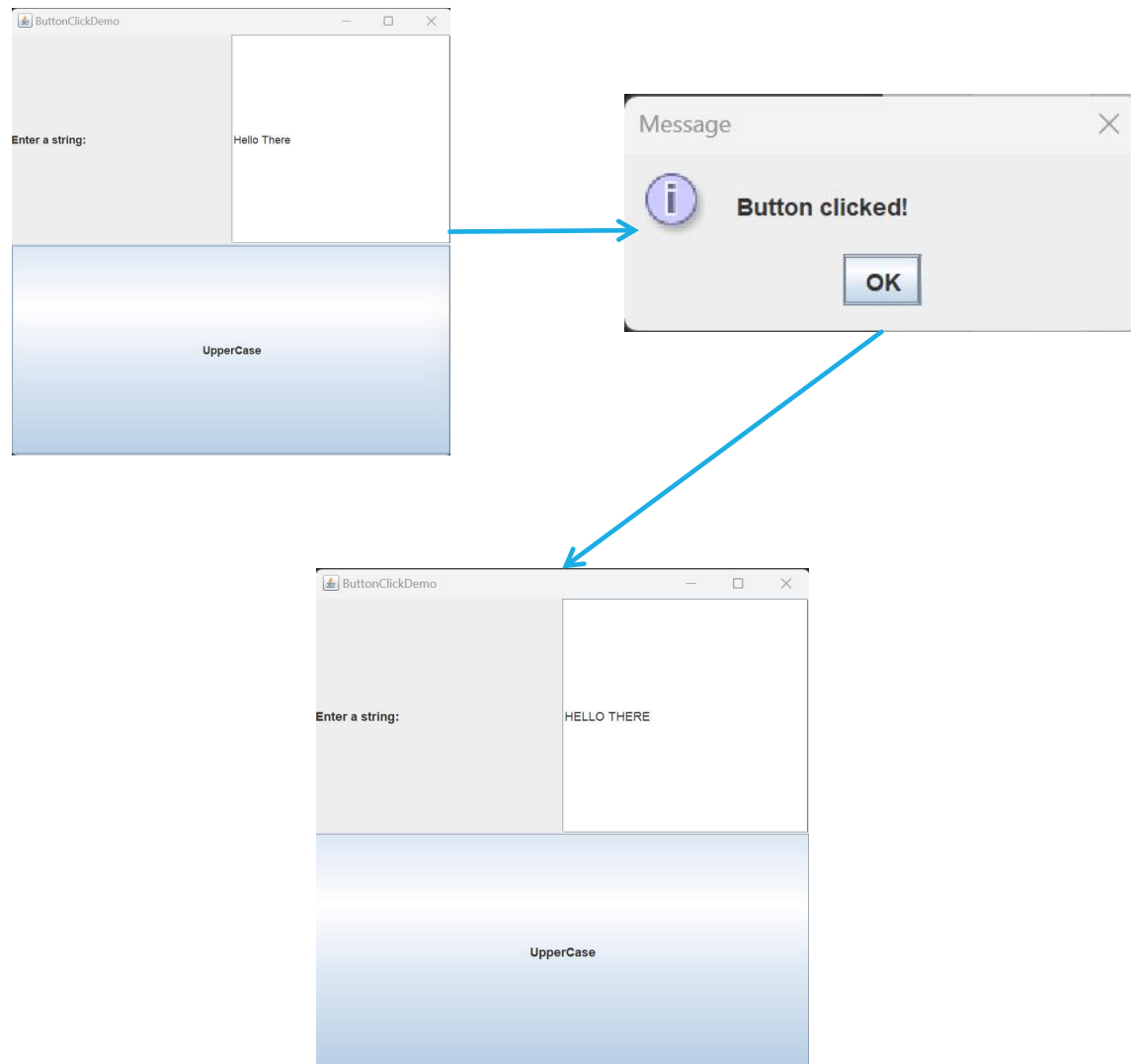


EVENT HANDLING PREVIEW

- So far we have been creating static GUIs without any interaction from the user.
- Event handling is used to capture action from user and generate appropriate responses.
- Modern event handling mechanisms uses *The Delegation Event Model* which separates the code of event generator (source) and event handler (listener).



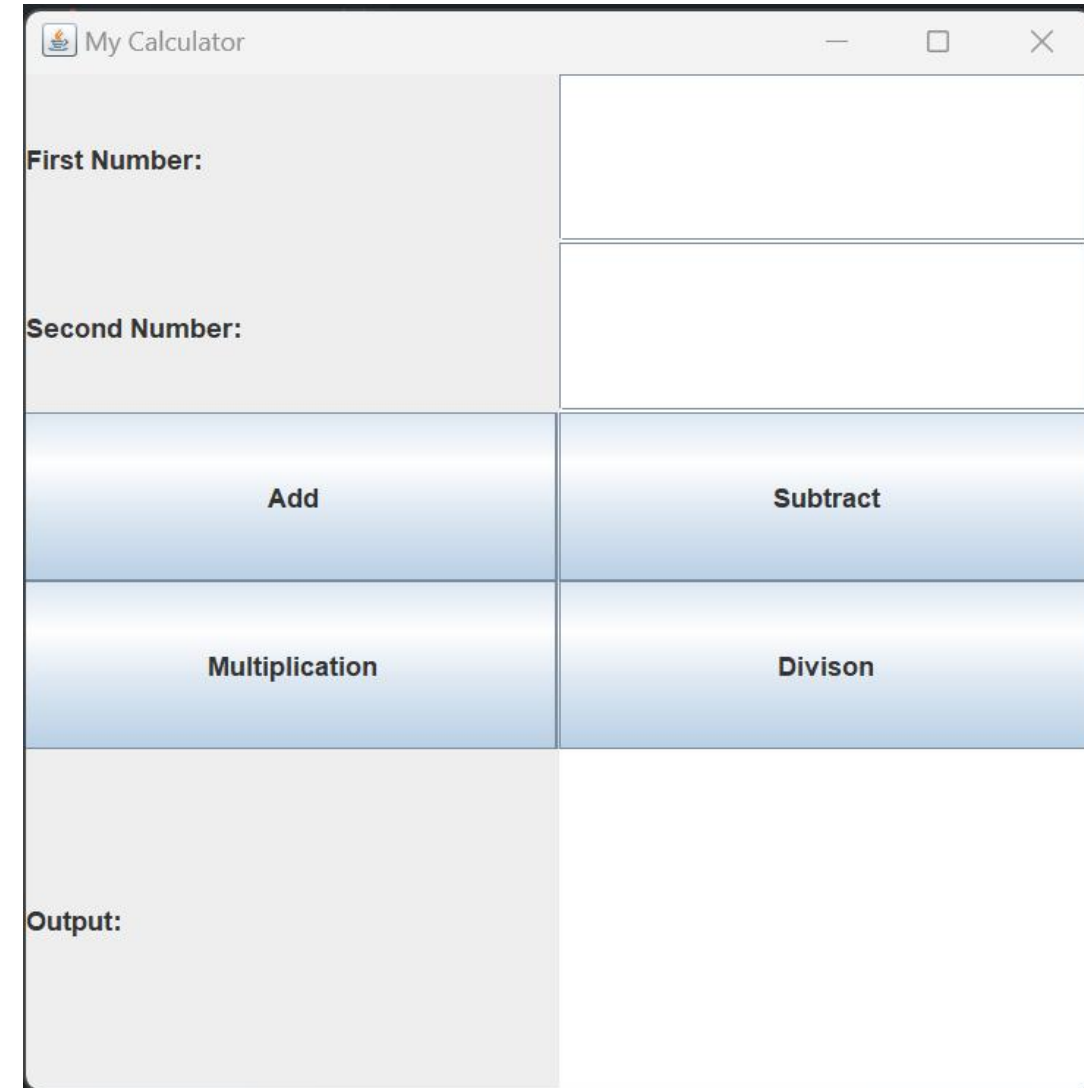
EVENT HANDLING DEMO: BUTTON CLICK EVENT



```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 class ButtonClickDemo implements ActionListener{
6     JFrame frame;
7     JTextField tfield;
8     JButton button;
9
10    ButtonClickDemo(){
11        frame = new JFrame("ButtonClickDemo");
12        frame.setSize(500, 500);
13        frame.setLayout(new GridLayout(2, 1));
14
15        /* Input panel and textfield
16        JPanel ipanel = new JPanel();
17        ipanel.setLayout(new GridLayout(1, 2));
18
19        JLabel label = new JLabel("Enter a string: ");
20        tfield = new JTextField();
21
22        ipanel.add(label);
23        ipanel.add(tfield);
24        frame.add(ipanel);
25
26        /* Button
27        button = new JButton("UpperCase");
28
29        /* Adding event listener to the button
30        /* Here 'this' refers to the object of ButtonClickDemo class as it implements ActionListener interface
31        button.addActionListener(this);
32        frame.add(button);
33
34        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
35        frame.setVisible(true);
36    }
37
38    void convertToUpperCase(JTextField tfield){
39        String text = tfield.getText();
40        String utext = text.toUpperCase();
41
42        tfield.setText(utext);
43    }
44
45    public void actionPerformed(ActionEvent e){
46        JOptionPane.showMessageDialog(null, "Button clicked!");
47
48        convertToUpperCase(tfield);
49    }
50
51    public static void main(String[] args){
52        new ButtonClickDemo();
53    }
```

EXERCISE: CALCULATOR

- Add button click events to the calculator exercise, we were working on.
- In finished version, when a user enter two numbers in each TextField, and press any button, Your program should read the numbers from TextField and display the result in Output TextArea.



The screenshot shows a Java Swing window titled "My Calculator". The window has a standard Mac OS-style title bar with minimize, maximize, and close buttons. The interface is divided into several sections:

- First Number:** A label followed by a large empty text field.
- Second Number:** A label followed by another large empty text field.
- Operation Buttons:** A 2x2 grid of buttons with a blue gradient:
 - Top-left: "Add"
 - Top-right: "Subtract"
 - Bottom-left: "Multiplication"
 - Bottom-right: "Divison" (note the spelling)
- Output:** A label followed by a large empty text area for displaying the result.

CARD LAYOUT

- The Java CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.
- Constructors -
 - **CardLayout():** creates a card layout with zero horizontal and vertical gap.
 - **CardLayout(int hgap, int vgap):** creates a card layout with the given horizontal and vertical gap.

CARDLAYOUT

- Some methods defined in the CardLayout class -
 - **public void next(Container parent):** is used to flip to the next card of the given container.
 - **public void previous(Container parent):** is used to flip to the previous card of the given container.
 - **public void first(Container parent):** is used to flip to the first card of the given container.
 - **public void last(Container parent):** is used to flip to the last card of the given container.
 - **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class CardLayoutExample1 extends JFrame implements ActionListener
{
    CardLayout crd;
    // button variables to hold the references of buttons
    JButton btn1, btn2, btn3;
    Container cPane;

    // constructor of the class
    CardLayoutExample1()
    {
        cPane = getContentPane();
        crd = new CardLayout();
        cPane.setLayout(crd);
        // creating the buttons
        btn1 = new JButton("Apple");
        btn2 = new JButton("Boy");
        btn3 = new JButton("Cat");
        // adding listeners to it
        btn1.addActionListener(this);
        btn2.addActionListener(this);
        btn3.addActionListener(this);

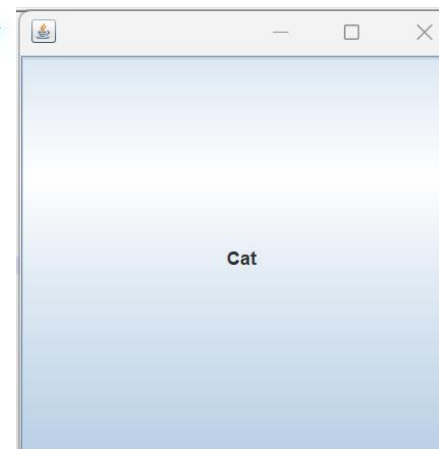
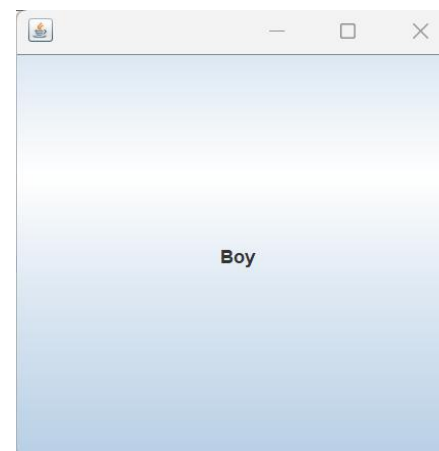
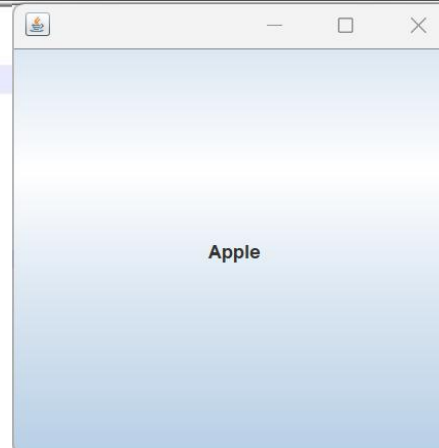
        cPane.add("a", btn1); // first card is the button btn1
        cPane.add("b", btn2); // first card is the button btn2
        cPane.add("c", btn3); // first card is the button btn3
    }

    public void actionPerformed(ActionEvent e)
    {
        // Upon clicking the button, the next card of the container is shown
        // after the last card, again, the first card of the container is shown upon clicking
        crd.next(cPane);
    }

    // main method
    public static void main(String args[])
    {
        // creating an object of the class CardLayoutExample1
        CardLayoutExample1 crd1 = new CardLayoutExample1();

        // size is 300 * 300
        crd1.setSize(300, 300);
        crd1.setVisible(true);
        crd1.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}

```



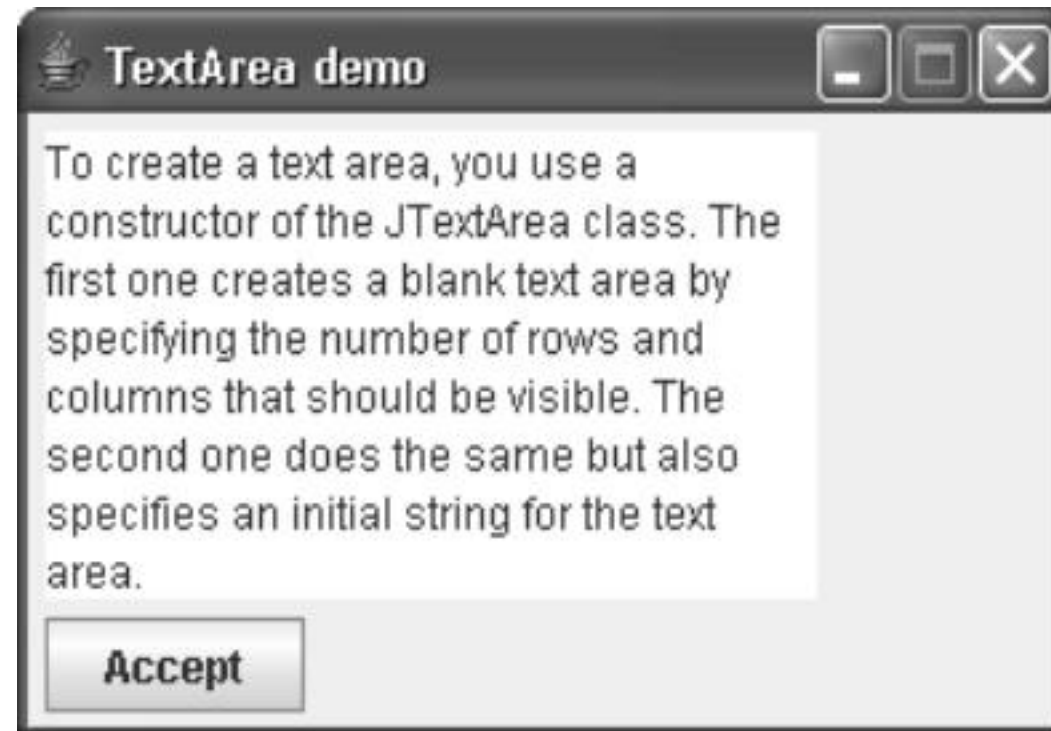
Some more Swing controls or GUI Components

Control	Class	Description
Text area	JTextArea	Lets the user enter more than one line of text.
Check box	JCheckBox	Lets the user select or deselect an option.
Radio button	JRadioButton	Lets the user select an option from a group of options.
List	JList	Lets the user select one or more items from a list of items.
Combo box	JComboBox	Lets the user select a single item from a drop-down list of items. A combo box can also let the user enter text into the text portion of the combo box.

Components that enhances Swing controls

Component	Class	Description
Border	JBorder	Can be used to visually group components or to enhance the appearance of an individual component.
Scroll pane	JScrollPane	Contains scroll bars that can be used to scroll through the contents of other controls. Scroll panes are typically used with text area and list controls.

A frame with a text area



Common constructors of the JTextArea class

Constructor	Description
JTextArea (int Rows, int Cols)	Creates an empty text area with the specified number of rows and columns.
JTextArea (String s, int Rows, int Cols)	Creates a text area with the specified number of rows and columns starting with the specified text.

Some methods that work with text areas

Method	Description
<code>setLineWrap(boolean)</code>	If the boolean value is true, the lines will wrap if they don't fit.
<code>setWrapStyleWord(boolean)</code>	If the boolean value is true and line wrapping is turned on, wrapped lines will be separated between words.
<code>append(String)</code>	Appends the specified string to the text in the text area.
<code>getText()</code>	Returns the text in the text area as a String.
<code>setText(String)</code>	Sets the text in the text area to the specified string.

Code that creates a text area

```
private JTextArea commentTextArea;  
commentTextArea = new JTextArea(7, 20);  
commentTextArea.setLineWrap(true);  
commentTextArea.setWrapStyleWord(true);  
add(commentTextArea);
```

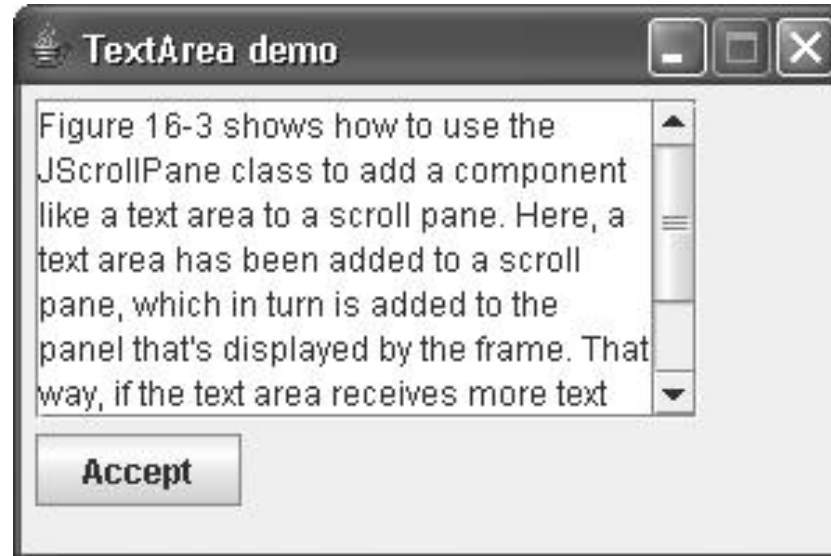
Code that gets the text stored in the text area

```
String comments = commentTextArea.getText();
```

Note

- If the text area is going to receive more text than can be viewed at one time, you should add the text area to a scroll pane.

A frame that displays a text area in a scroll pane



Common constructors of the JScrollPane class

Constructor	Description
JScrollPane (Component)	Creates a scroll pane that displays the specified component, along with vertical and horizontal scrollbars as needed.
JScrollPane (Component, vertical, horizontal)	Creates a scroll pane that displays the specified component and uses the specified vertical and horizontal policies.

Fields of the ScrollPaneConstants interface that set scrollbar policies

Field	Description
<code>VERTICAL_SCROLLBAR_ALWAYS</code>	Always display a vertical scrollbar.
<code>VERTICAL_SCROLLBAR_AS_NEEDED</code>	Display a vertical scrollbar only when needed.
<code>VERTICAL_SCROLLBAR_NEVER</code>	Never display a vertical scrollbar.
<code>HORIZONTAL_SCROLLBAR_ALWAYS</code>	Always display a horizontal scrollbar.
<code>HORIZONTAL_SCROLLBAR_AS_NEEDED</code>	Display a horizontal scrollbar only when needed.
<code>HORIZONTAL_SCROLLBAR_NEVER</code>	Never display a horizontal scrollbar.

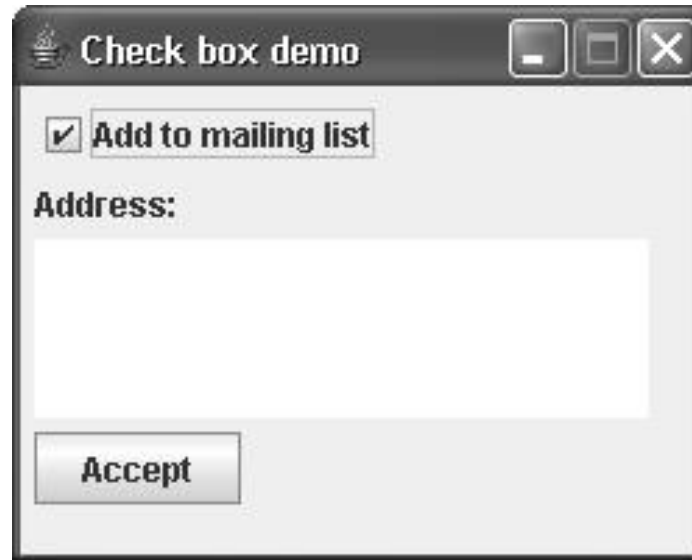
Code that uses a scroll pane with a text area

```
private JTextArea commentTextArea = new JTextArea(7, 20);
commentTextArea.setLineWrap(true);
commentTextArea.setWrapStyleWord(true);
JScrollPane commentScroll = new
JScrollPane(commentTextArea);
add(commentScroll);
```

Code that creates a scroll pane and sets the scroll bar policies

```
JScrollPane commentScroll =
    new JScrollPane(commentTextArea,
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
```


A frame with a check box



Common constructors of the JCheckBox class

Constructor	Description
JCheckBox (String)	Creates an unselected check box with a label that contains the specified string.
JCheckBox (String, boolean)	Creates a check box with a label that contains the specified string. If the boolean value is true, the check box is selected.

Some methods that work with check boxes

Method	Description
isSelected()	Returns a true value if the check box is selected.
setSelected (boolean)	Checks or unchecks the check box depending on the boolean value.
addActionListener (ActionListener)	Adds an action listener to the check box.

Code that creates the check box

```
private JCheckBox mailingCheckBox; mailingCheckBox =  
    new JCheckBox("Add to mailing list", true);  
mailingCheckBox.addActionListener(this);  
add(mailingCheckBox);
```

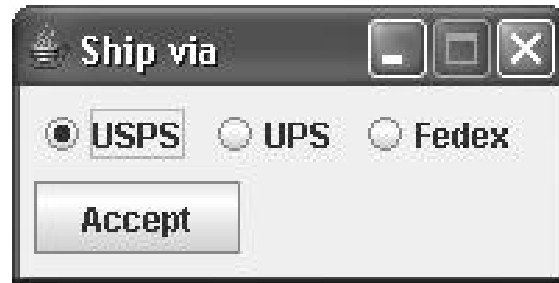
Code that checks the status of the check box

```
boolean addToList = mailingCheckBox.isSelected();
```

An actionPerformed method for the check box

```
public void actionPerformed(ActionEvent e)
{
    Object source = e.getSource();
    if (source == mailingCheckBox)
    {
        if (mailingCheckBox.isSelected())
            addressTextArea.setEnabled(true);
        else
            addressTextArea.setEnabled(false);
    }
}
```

A frame with three radio buttons



How to work with radio buttons

- You must add each radio button in a set of options to a `ButtonGroup` object.
- Selecting a radio button automatically deselects all other radio buttons in the same button group.

Common constructors and methods of the JRadioButton class

Constructor	Description
<code>JRadioButton(String)</code>	Creates an unselected radio button with the specified text.
<code>JRadioButton(String, boolean)</code>	Creates a radio button with the specified text. If the boolean value is true, the radio button is selected.
Method	Description
<code>isSelected()</code>	Returns a true value if the radio button is selected.
<code>addActionListener(ActionListener)</code>	Adds an action listener to the radio button.

Common constructor and method of the ButtonGroup class

Constructor	Description
<code>ButtonGroup()</code>	Creates a button group used to hold a group of buttons.
Method	Description
<code>add(AbstractButton)</code>	Adds the specified button to the group.

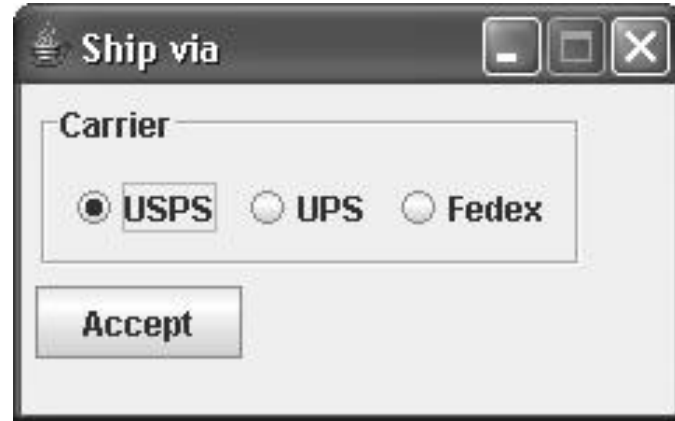
Code that creates three radio buttons and adds them to a panel

```
private JRadioButton uspsRadioButton, upsRadioButton,  
    fedexRadioButton;  
uspsRadioButton = new JRadioButton("USPS", true);  
upsRadioButton = new JRadioButton("UPS");  
fedexRadioButton = new JRadioButton("Fedex");  
add(uspsRadioButton);  
add(upsRadioButton);  
add(fedexRadioButton);  
ButtonGroup shipViaGroup = new ButtonGroup();  
shipViaGroup.add(uspsRadioButton);  
shipViaGroup.add(upsRadioButton);  
shipViaGroup.add(fedexRadioButton);
```


Code that determines which radio button is selected

```
if (uspsRadioButton.isSelected())  
    shipVia = "USPS";  
else if (upsRadioButton.isSelected())  
    shipVia = "UPS";  
else if (fedexRadioButton.isSelected())  
    shipVia = "Federal Express";
```

Radio buttons with an etched and titled border



How to work with borders

- To place controls in a border, you must create a panel, create a border and apply it to the panel, and add the controls to the panel.
- Because a border only groups controls visually, you must still use a `ButtonGroup` object to group radio buttons logically.
- To set borders, you must import the `javax.swing.border` package.

Static methods of the BorderLayout class

Method	Description
<code>createLineBorder()</code>	Creates a line border.
<code>createEtchedBorder()</code>	Creates an etched border.
<code>createLoweredBevelBorder()</code>	Creates a lowered bevel border.
<code>createRaisedBevelBorder()</code>	Creates a raised bevel border.
<code>createTitledBorder(String)</code>	Creates a line border with the specified title.
<code>createTitledBorder(Border, String)</code>	Adds the specified title to the specified border.

Method of the JComponent class used to set borders

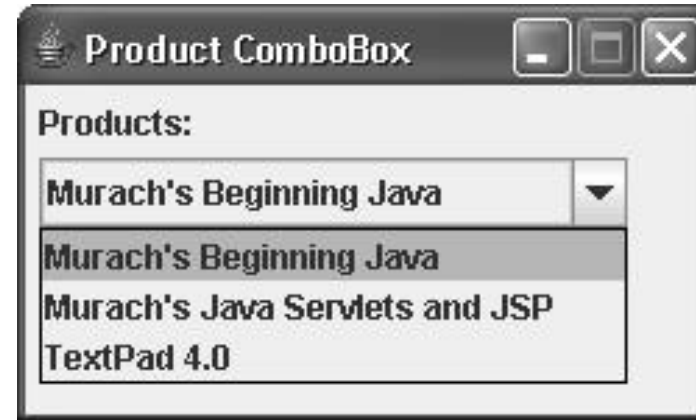
Method	Description
<code>setBorder(Border)</code>	Sets the border style for a component.

Code that creates bordered radio buttons

```
uspsRadioButton = new JRadioButton("USPS", true);  
upsRadioButton = new JRadioButton("UPS");  
fedexRadioButton = new JRadioButton("Fedex");  
ButtonGroup shipViaGroup = new ButtonGroup();  
shipViaGroup.add(uspsRadioButton);  
shipViaGroup.add(upsRadioButton);  
shipViaGroup.add(fedexRadioButton);
```

```
JPanel shipViaPanel = new JPanel();  
Border shipViaBorder =  
    BorderFactory.createEtchedBorder();  
shipViaBorder =  
    BorderFactory.createTitledBorder(shipViaBorder,  
    "Carrier");  
shipViaPanel.setBorder(shipViaBorder);  
shipViaPanel.add(uspsRadioButton);  
shipViaPanel.add(upsRadioButton);  
shipViaPanel.add(fedexRadioButton);  
add(shipViaPanel);
```

A frame with a combo box



Common constructors of the JComboBox class

Constructor	Description
<code>JComboBox ()</code>	Creates an empty combo box.
<code>JComboBox (Object [])</code>	Creates a combo box with the objects stored in the specified array.

Some methods that work with combo boxes

Method	Description
<code>getSelectedItem()</code>	Returns an Object type for the selected item.
<code>getSelectedIndex()</code>	Returns an int value for the index of the selected item.
<code>setSelectedIndex(intIndex)</code>	Selects the item at the specified index.
<code>setEditable(boolean)</code>	If the boolean value is true, the combo box can be edited.
<code>getItemCount()</code>	Returns the number of items stored in the combo box.
<code>addItem(Object)</code>	Adds an item to the combo box.
<code>removeItemAt(int)</code>	Removes the item at the specified index from the combo box.

Some methods that work with combo boxes (continued)

Method	Description
<code>removeItem(Object)</code>	Removes the specified item from the combo box.
<code>addActionListener(ActionListener)</code>	Adds an action listener to the combo box.
<code>addItemListener(ItemListener)</code>	Adds an item listener to the combo box.

Code that creates the combo box

```
private ArrayList<Product> products;  
products = getProducts();  
                // returns an ArrayList of products  
productComboBox = new JComboBox();  
for (Product p : products)  
    productComboBox.addItem(p.getDescription());  
add(productComboBox);
```

Code that determines which item was selected

```
int i = productComboBox.getSelectedIndex();  
Product p = products.get(i);
```

A frame that uses an action event listener to update the display based on the user's selection



The actionPerformed method of the ActionListener interface

Method	Description
<code>void actionPerformed(ActionEvent e)</code>	Invoked when an item is selected.

The itemStateChanged method of the ItemListener interface

Method	Description
<code>void itemStateChanged(ItemEvent e)</code>	Invoked when an item is selected or deselected.

Common methods of the ItemEvent class

Method	Description
<code>getSource()</code>	Returns the source of the event.
<code>getItem()</code>	Returns the selected item.
<code>getStateChanged()</code>	Returns an int value that indicates whether an item was selected or deselected. The field names for these values are <code>SELECTED</code> and <code>DESELECTED</code> .

Code that creates a combo box

```
products = getProducts();
productComboBox = new JComboBox();
for (Product p : products)
    productComboBox.addItem(p.getDescription());
productComboBox.setSelectedIndex(0);
productComboBox.addActionListener(this);
add(productComboBox);
```

Code that implements the ActionListener interface for the combo box

```
public void actionPerformed(ActionEvent e)
{
    Object source = e.getSource();
    if (source == productComboBox)
    {
        int i = productComboBox.getSelectedIndex();
        Product p = products.get(i);
        priceTextField.setText(p.getFormattedPrice());
    }
}
```

A frame that includes a list



Common constructors of the JList class

Constructor	Description
JList (Object[])	Creates a list that contains the objects stored in the specified array of objects.
JList (ListModel)	Creates a list using the specified list model.

Some methods of the JList class

Method	Description
<code>getSelectedValue()</code>	Returns the selected item as an Object type.
<code>getSelectedIndex()</code>	Returns an int value for the index of the selected item.
<code>isSelectedIndex(intIndex)</code>	Returns a true value if the item at the specified index is selected.
<code>setFixedCellWidth(intPixels)</code>	Sets the cell width to the specified number of pixels. Otherwise, the width of the list is slightly wider than the widest item in the array that populates the list.
<code>setVisibleRowCount(intRows)</code>	Sets the visible row count to the specified int value. This only works when the list is displayed within a scroll pane.

Some methods of the JList class (continued)

Method	Description
setSelectionMode (mode)	Sets the selection mode. To allow single selections, specify ListSelectionModel.SINGLE_SELECTION.
setSelectedIndex (intIndex)	Selects the item at the specified index.

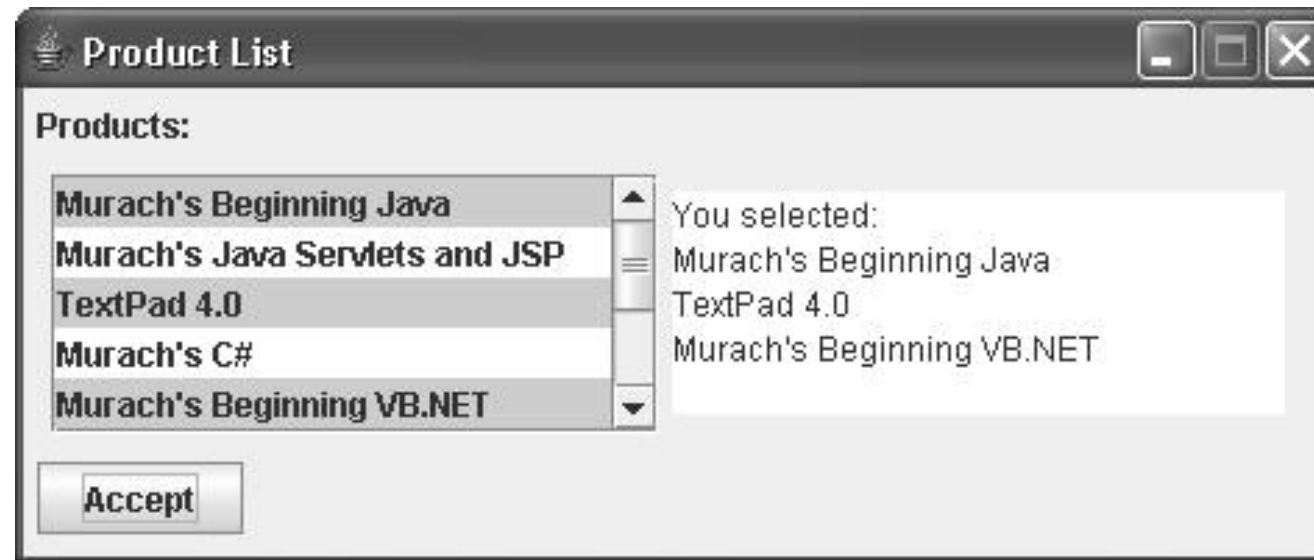
Code that creates a list

```
descriptions = getProductDescriptions();  
                // returns an array of descriptions  
productList = new JList(descriptions);  
productList.setFixedCellWidth(200);  
productList.setVisibleRowCount(5);  
productList.setSelectedIndex(0);  
productList.setSelectionMode(  
    ListSelectionModel.SINGLE_SELECTION);  
add(new JScrollPane(productList));
```

Code that gets the selected item

```
String s = (String)productList.getSelectedValue();
```

A list that allows multiple selections



Fields of the ListSelectionModel interface used to set the selection mode

Field	Description
<code>SINGLE_SELECTION</code>	Allows just one selection.
<code>SINGLE_INTERVAL_SELECTION</code>	Allows a single range of selections.
<code>MULTIPLE_INTERVAL_SELECTION</code>	Allows multiple ranges of selections. This is the default.

Methods of the JList class used to process multiple selections

Method	Description
<code>getSelectedValues()</code>	Returns an array of Object types for the selected items.
<code>getSelectedIndices()</code>	Returns an array of ints corresponding to the indices of the selected items.

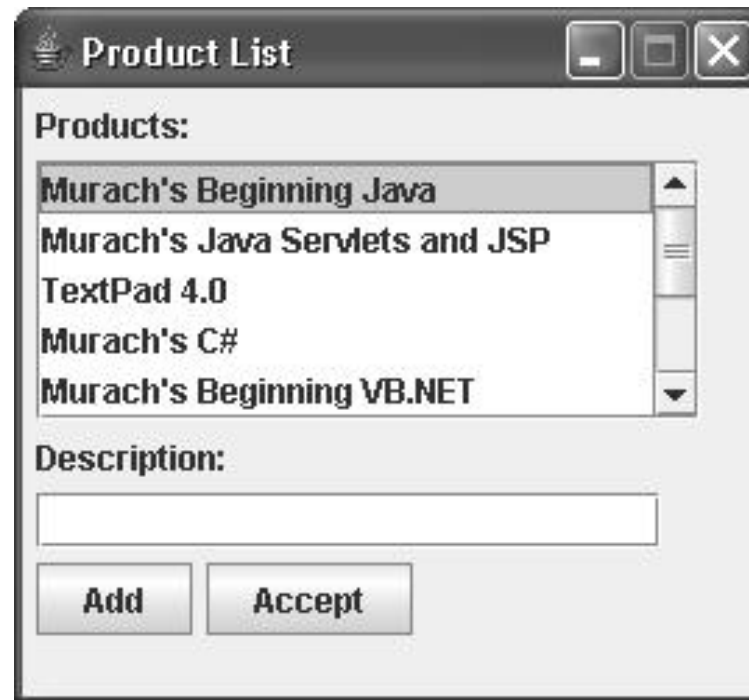
Code that creates a list

```
descriptions = getProductDescriptions();  
                // returns an array of descriptions  
productList = new JList(descriptions);  
productList.setFixedCellWidth(200);  
productList.setVisibleRowCount(5);  
productList.setSelectedIndex(0);  
productList.setSelectionMode(  
    ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);  
add(new JScrollPane(productList));
```

Code that displays the list selections

```
public void actionPerformed(ActionEvent e)
{
    Object source = e.getSource();
    if (source == acceptButton)
    {
        Object[] selections =
            productList.getSelectedValues();
        String s = "";
        for (Object o : selections)
            s += (String)o + "\n";
        productTextArea.setText("You selected:\n" + s);
    }
}
```

A frame that lets you add elements to a list



EVENT HANDLING

- An event is something that happens in the program based on some kind of triggering input.
- Event is typically caused (i.e. generated) by user interaction. For example - mouse click, button press, selecting from a list etc.
- Event can also be generated internally by the program.
- In Java, events are objects, so each type of event is represented by a distinct class (Similar to the way exceptions are distinct classes).

TYPES OF EVENTS IN JAVA

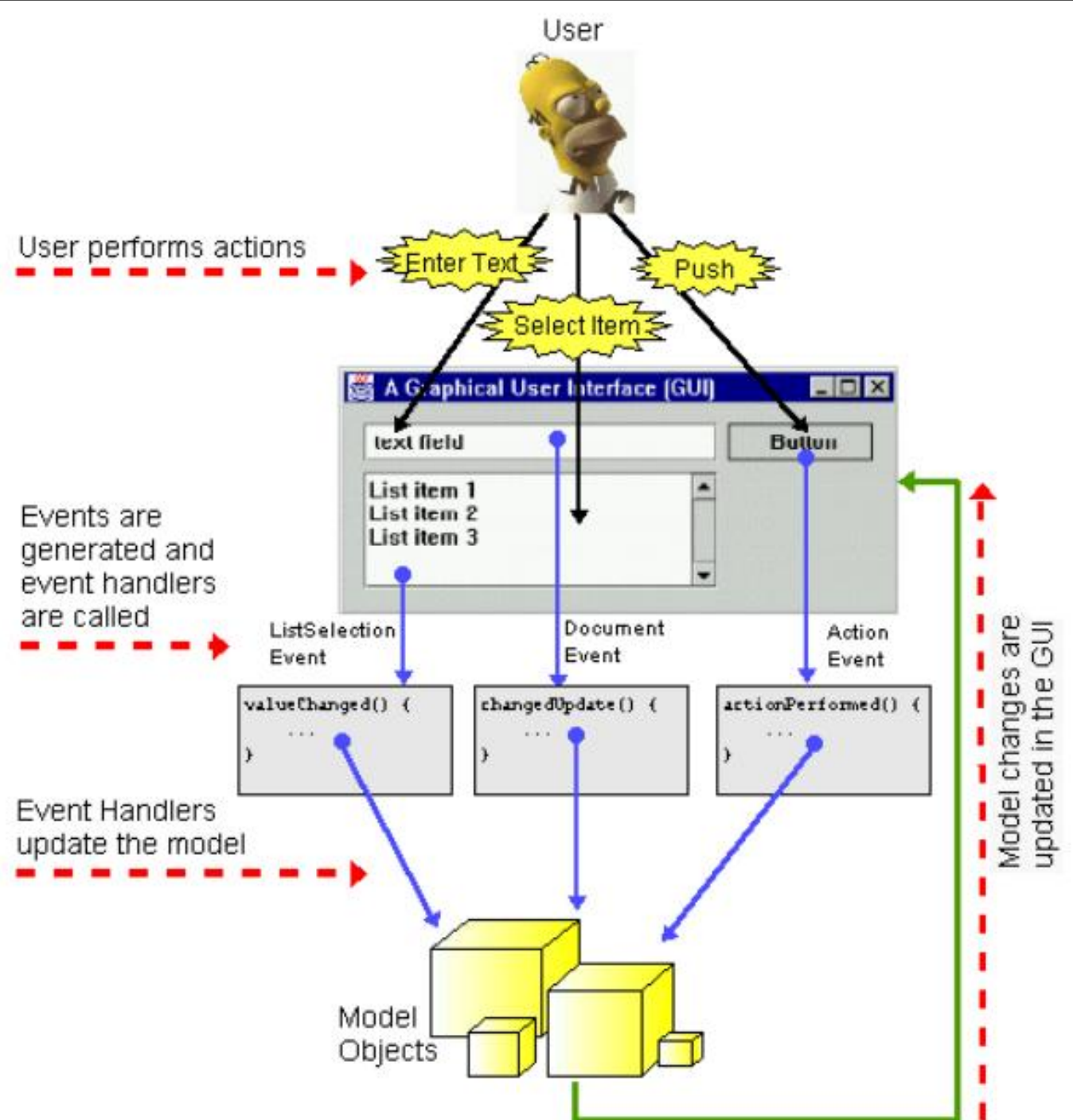
Here is a list of the commonly used types of events in JAVA:

- **Action Events:** clicking buttons, selecting items from lists etc....
- **Component Events:** changes in the component's size, position, or visibility.
- **Focus Events:** gain or lose the ability to receive keyboard input.
- **Key Events:** key presses; generated only by the component that has the current keyboard focus.
- **Mouse Events:** mouse clicks and the user moving the cursor into or out of the component's drawing area.
- **Mouse Motion Events:** changes in the cursor's position over the component.
- **Container Events:** component has been added to or removed from the container.

Here are a couple of the "less used" types of events in JAVA:

- **Ancestor Events:** containment ancestors is added to or removed from a container, hidden, made visible, or moved.
- **Property Change Events:** part of the component has changed (e.g., color, size,...).

EVENT DRIVER GUI



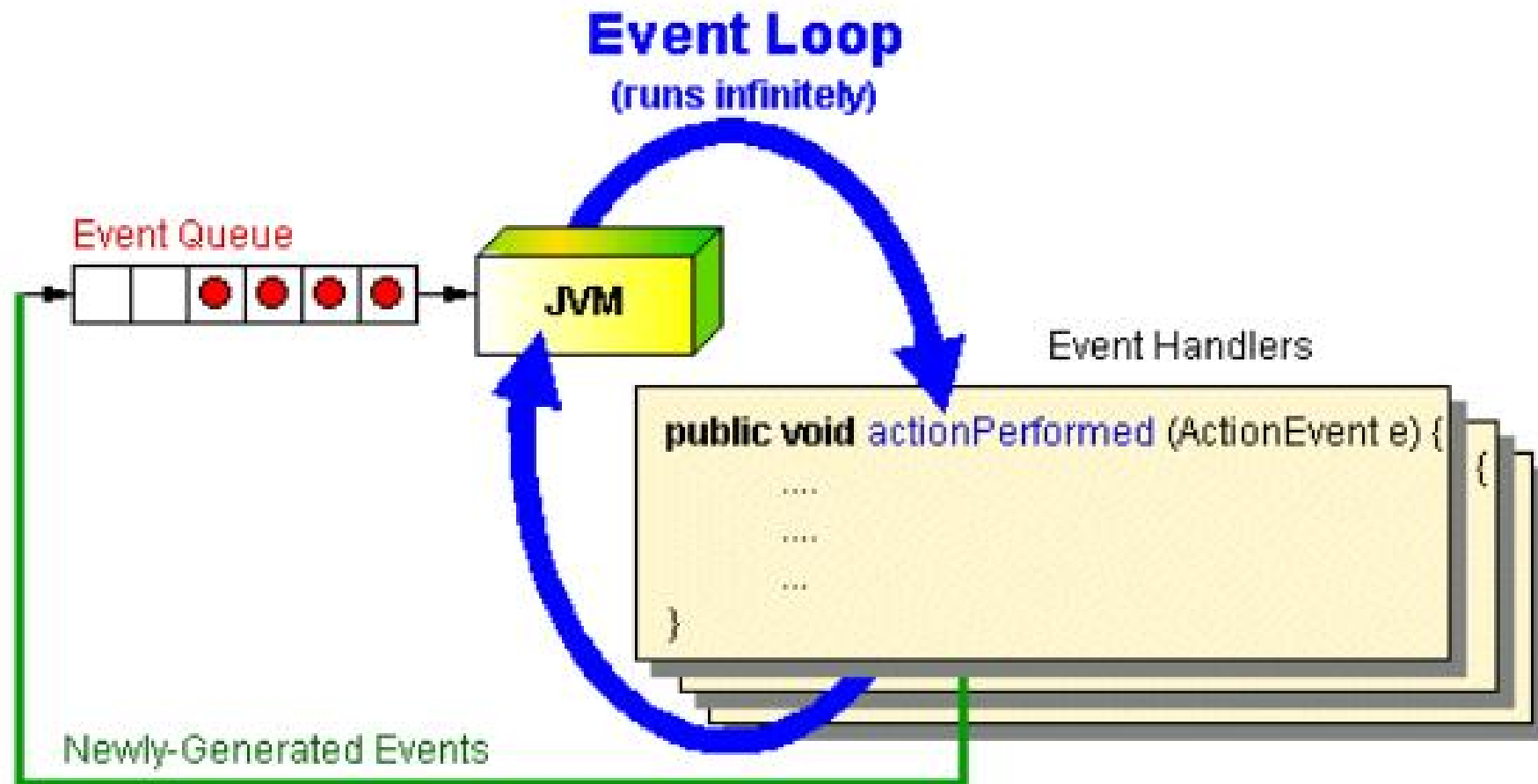
MODEL-VIEW-CONTROLLER ARCHITECTURE

- Java GUI components follows **MVC (Model View Controller)** architecture.
- **Model** - *Model* represents the components state or data related logic that the user works with.
- **View** - *View* represents the UI logic. It generates the user interface for the user.
- **Controller** - *Controller* acts as intermediary between *Model* and *View* and enables interconnection between the views and model.

EVENT LOOP

- An Event Loop is an endless loop that waits for events to occur -
 - Events are queued (lined up on a first-come-first-serve basis) in a buffer.
 - Events are handled one at a time by an event handler
 - The JVM spends all of its time taking an event out of the queue, processing it and then going back to the queue for another.
 - While each event is being handled, JAVA is unable to process any other events. So you must be very careful to make sure that your event handling code does not take too long. Otherwise JVM will not take any more events from the queue. And your program will look like it has been stopped or stuck.

EVENT LOOP



THE DELEGATION EVENT MODEL

- The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a source generates an event and sends it to one or more listeners.
- In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.

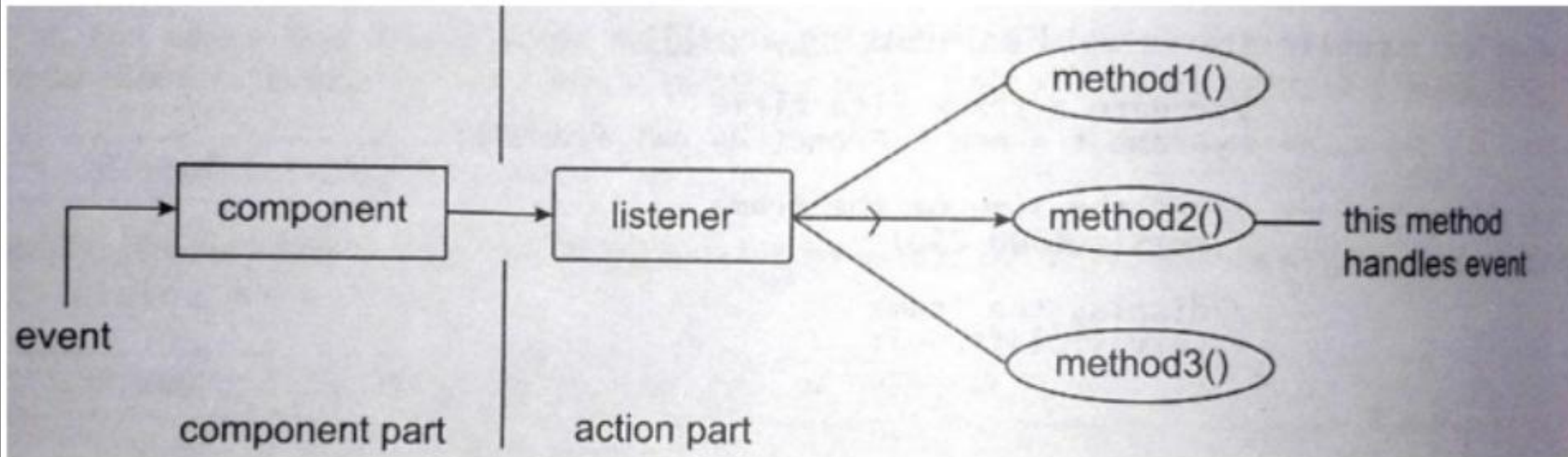
THE DELEGATION EVENT MODEL

- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.
- This is a more efficient way to handle events than the design used by the original Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time.

THE DELEGATION EVENT MODEL

- **Events** - An event is a phenomenon which changes the state of a source. For example, button click, entering characters via the keyboard in TextField, etc.
- **Event Sources** - A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.
- **Event Listeners** - A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.
- When an event occurs, all the registered listeners are notified and receive a copy of the event object. This is known as ***multicasting*** the event.

THE DELEGATION EVENT MODEL



SOURCES OF EVENTS

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked;
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

EVENT CLASSES & LISTENER INTERFACES

- The event classes that represent events are at the core of Java's event handling mechanism.
- The ***java.awt.event*** package provides many event classes and listener interfaces for event handling.
- At the root of the Java event class hierarchy is **EventObject**, which is in *java.util* package. It is the superclass for all events. Constructor for EventObject class is ***EventObject (Object src)*** where *src* is the object that generates this event.
- EventObject class defines two methods: ***getSource()*** and ***toString()***.

Event Classes & Interfaces

Event Class	Description	Listener Interface
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.	ActionListener
AdjustmentEvent	Generated when a scroll bar is manipulated.	AdjustmentListener
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.	ComponentListener
ContainerEvent	Generated when a component is added to or removed from a container.	ContainerListener
FocusEvent	Generated when a component gains or losses keyboard focus.	FocusListener
InputEvent	Abstract super class for all component input event classes.	
ItemEvent	Generated when a check box or list item is clicked	ItemListener
KeyEvent	Generated when input is received from the keyboard.	KeyListener
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.	MouseListener and MouseMotionListener
TextEvent	Generated when the value of a text area or text field is changed.	TextListener
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.	WindowListener

THE ACTION EVENT CLASS

- An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- You can obtain the command name for the invoking ActionEvent object by using the ***String* getActionCommand()** method. For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.
- The method ***long* getWhen()** returns the time at which the event took place. This is called the *event's timestamp*.

THE ITEM EVENT CLASS

- An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.
- The **Object getItem()** method can be used to obtain a reference to the item that changed.
- The **int getStateChange()** method returns the state change (that is, **SELECTED** or **DESELECTED**) for the event.

THE KEY EVENT CLASS

- A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: `KEY_PRESSED`, `KEY_RELEASED`, and `KEY_TYPED`.
- The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing shift does not generate a character.
- The ***char getKeyChar()*** method can be used to get character which has been entered. And ***int getKeyCode()*** method can be used to get key code.

EVENT LISTENER INTERFACES

Interface	Description
ActionListener	Defines the actionPerformed() method to receive and process action events. <i>void actionPerformed(ActionEvent ae)</i>
MouseListener	Defines five methods to receive mouse events, such as when a mouse is clicked, pressed, released, enters, or exits a component <i>void mouseClicked(MouseEvent me)</i> <i>void mouseEntered(MouseEvent me)</i> <i>void mouseExited(MouseEvent me)</i> <i>void mousePressed(MouseEvent me)</i> <i>void mouseReleased(MouseEvent me)</i>
MouseMotionListener	Defines two methods to receive events, such as when a mouse is dragged or moved. <i>void mouseDragged(MouseEvent me)</i> <i>void mouseMoved(MouseEvent me)</i>
AdjustmentListner	Defines the adjustmentValueChanged() method to receive and process the adjustment events. <i>void adjustmentValueChanged(AdjustmentEvent ae)</i>
TextListener	Defines the textValueChanged() method to receive and process an event when the text value changes. <i>void textValueChanged(TextEvent te)</i>

WindowListener	<p>Defines seven window methods to receive events.</p> <pre> void windowActivated(WindowEvent we) void windowClosed(WindowEvent we) void windowClosing(WindowEvent we) void windowDeactivated(WindowEvent we) void windowDeiconified(WindowEvent we) void windowIconified(WindowEvent we) void windowOpened(WindowEvent we) </pre>
ItemListener	<p>Defines the itemStateChanged() method when an item has been</p> <pre> void itemStateChanged(ItemEvent ie) </pre>
WindowFocusListener	<p>This interface defines two methods: windowGainedFocus() and windowLostFocus(). These are called when a window gains or loses input focus. Their general forms are shown here:</p> <pre> void windowGainedFocus(WindowEvent we) void windowLostFocus(WindowEvent we) </pre>
ComponentListener	<p>This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:</p> <pre> void componentResized(ComponentEvent ce) void componentMoved(ComponentEvent ce) void componentShown(ComponentEvent ce) void componentHidden(ComponentEvent ce) </pre>

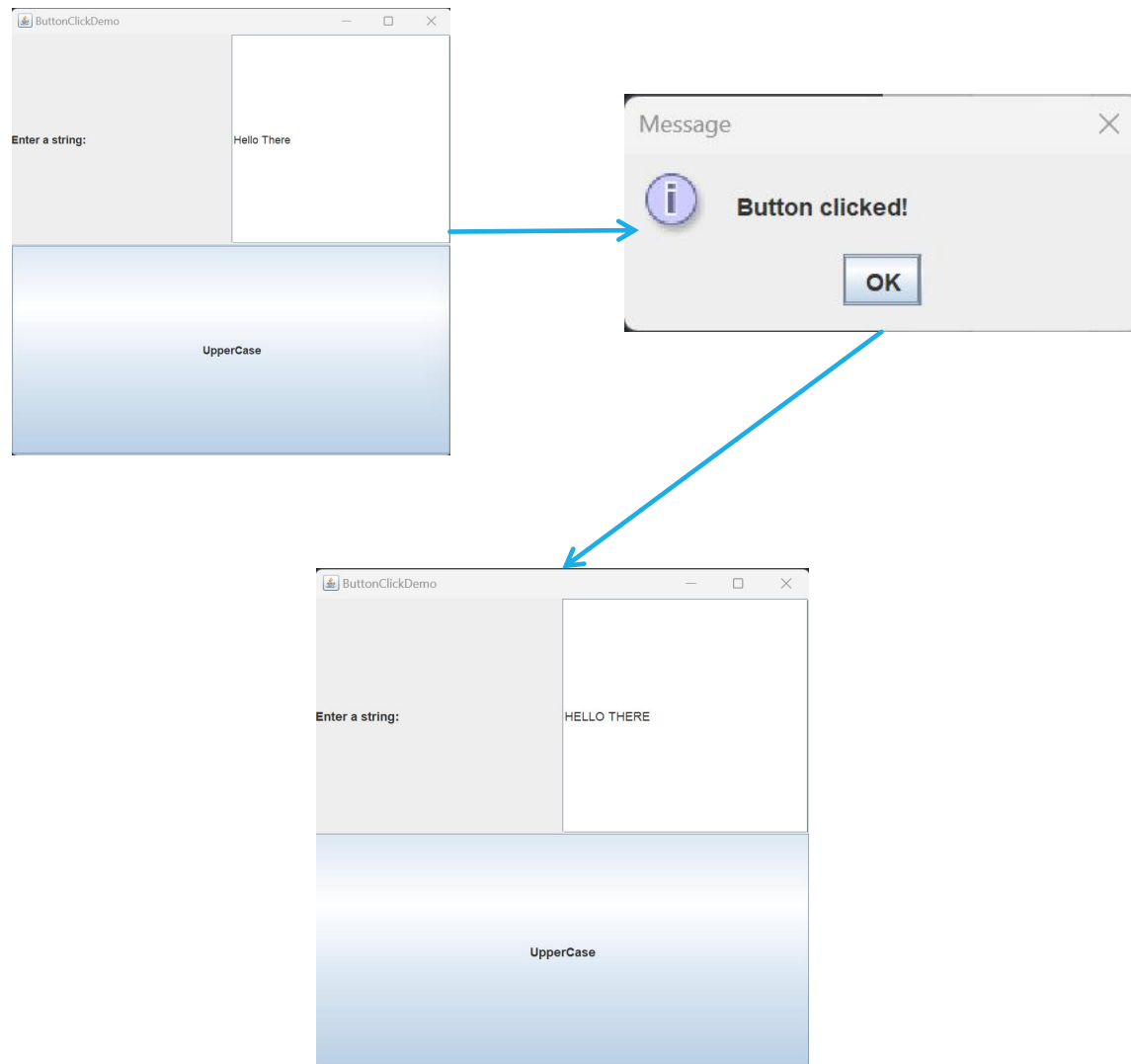
ContainerListener	<p>This interface contains two methods. When a component is added to a container, componentAdded() is invoked. When a component is removed from a container, componentRemoved() is invoked.</p> <p>Their general forms are shown here:</p> <pre><i>void componentAdded(ContainerEvent ce)</i> <i>void componentRemoved(ContainerEvent ce)</i></pre>
FocusListener	<p>This interface defines two methods. When a component obtains keyboard focus, focusGained() is invoked. When a component loses keyboard focus, focusLost() is called. Their general forms are shown here:</p> <pre><i>void focusGained(FocusEvent fe)</i> <i>void focusLost(FocusEvent fe)</i></pre>
KeyListener	<p>This interface defines three methods.</p> <pre><i>void keyPressed(KeyEvent ke)</i> <i>void keyReleased(KeyEvent ke)</i> <i>void keyTyped(KeyEvent ke)</i></pre>

STEPS TO PERFORM EVENT HANDLING

- Following steps are required to perform event handling -
 - Register the component with the listener. Many classes provide the registration methods.
 - Implement the concerned interface i.e. provide the implementation of methods provided by the interface. This can be done using either inner classes, anonymous inner class or even outer class itself.

- **Button**
 - `public void addActionListener(ActionListener a){}`
- **MenuItem**
 - `public void addActionListener(ActionListener a){}`
- **TextField**
 - `public void addActionListener(ActionListener a){}`
 - `public void addTextListener(TextListener a){}`
- **TextArea**
 - `public void addTextListener(TextListener a){}`
- **Checkbox**
 - `public void addItemListener(ItemListener a){}`
- **Choice**
 - `public void addItemListener(ItemListener a){}`
- **List**
 - `public void addActionListener(ActionListener a){}`
 - `public void addItemListener(ItemListener a){}`
- **Mouse**
 - `public void addMouseListener(MouseListener a){}`

EVENT HANDLING DEMO: BUTTON CLICK EVENT



```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  class ButtonClickDemo implements ActionListener{
6      JFrame frame;
7      JTextField tfield;
8      JButton button;
9
10     ButtonClickDemo(){
11         frame = new JFrame("ButtonClickDemo");
12         frame.setSize(500, 500);
13         frame.setLayout(new GridLayout(2, 1));
14
15         /* Input panel and textfield
16         JPanel ipanel = new JPanel();
17         ipanel.setLayout(new GridLayout(1, 2));
18
19         JLabel label = new JLabel("Enter a string: ");
20         tfield = new JTextField();
21
22         ipanel.add(label);
23         ipanel.add(tfield);
24         frame.add(ipanel);
25
26         /* Button
27         button = new JButton("UpperCase");
28
29         /* Adding event listener to the button
30         /* Here 'this' refers to the object of ButtonClickDemo class as it implements ActionListener interface
31         button.addActionListener(this);
32         frame.add(button);
33
34         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
35         frame.setVisible(true);
36     }
37
38     void convertToUpperCase(JTextField tfield){
39         String text = tfield.getText();
40         String utext = text.toUpperCase();
41
42         tfield.setText(utext);
43     }
44
45     public void actionPerformed(ActionEvent e){
46         JOptionPane.showMessageDialog(null, "Button clicked!");
47
48         convertToUpperCase(tfield);
49     }
50     public static void main(String[] args){
51         new ButtonClickDemo();
52     }
53 }
```

```

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

class KeyEventDemo{
    public static void main(String[] args){
        JFrame frame = new JFrame("KeyEventDemo");
        frame.setSize(500, 500);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);

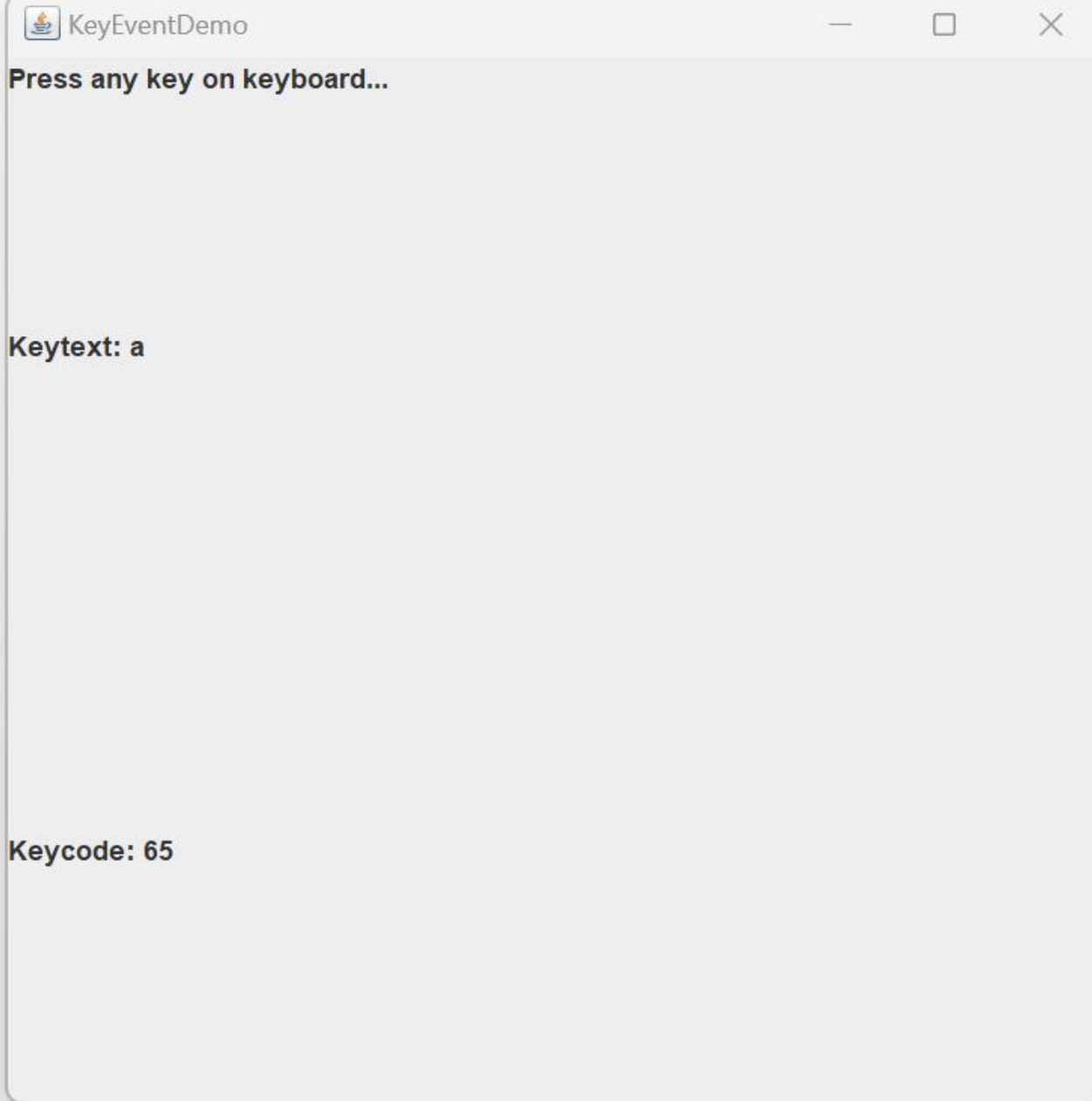
        JLabel label1 = new JLabel("Press any key on keyboard...");
        JLabel label2 = new JLabel("Pressed key will be displayed here...");
        JLabel label3 = new JLabel("key code will be displayed here...");
        JPanel panel = new JPanel(new GridLayout(2, 1));

        frame.add(label1, BorderLayout.NORTH);
        panel.add(label2);
        panel.add(label3);
        frame.add(panel, BorderLayout.CENTER);

        //! Adding KeyEvent
        frame.setFocusable(true);
        frame.addKeyListener(new KeyListener(){
            public void keyPressed(KeyEvent ke){
                char key = ke.getKeyChar();
                int keyCode = ke.getKeyCode();
                label2.setText("Keytext: " + String.valueOf(key));
                label3.setText("Keycode: " + keyCode);
            }
            public void keyReleased(KeyEvent ke){
                char key = ke.getKeyChar();
                int keyCode = ke.getKeyCode();
                label2.setText("Keytext: " + String.valueOf(key));
                label3.setText("Keycode: " + keyCode);
            }
            public void keyTyped(KeyEvent ke){

            }
        });
    }
}

```



PRACTICE EXERCISE: GUESS THE CHARACTER

- Write a simple GUI program in Swing which displays a popular character and user has to guess the name of character.