

# Java Swing

(Week 1-2)

Semester 2 -Spring 2017  
Lecturer: NGUYEN MINH DAO  
FIT -UTE



# SWING - Basic

2

# Overview

3

- **Swing API** is a set of extensible GUI Components to ease the developer's life to create JAVA based Front End/GUI Applications.
- It is build on top of **AWT API** and acts as a replacement of AWT API, since it has almost every control corresponding to AWT controls.
- Unlike AWT, Java Swing provides platform-independent and lightweight components.
- The **javax.swing package** provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.



# Difference between AWT and Swing

4




There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are <b>platform-dependent</b> .	Java swing components are <b>platform-independent</b> .
2)	AWT components are <b>heavyweight</b> .	Swing components are <b>lightweight</b> .
3)	AWT <b>doesn't support pluggable look and feel</b> .	Swing <b>supports pluggable look and feel</b> .
4)	AWT provides <b>less components</b> than Swing.	Swing provides <b>more powerful components</b> such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT <b>doesn't follows MVC</b> (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing <b>follows MVC</b> .

# SWING component class hierarchy

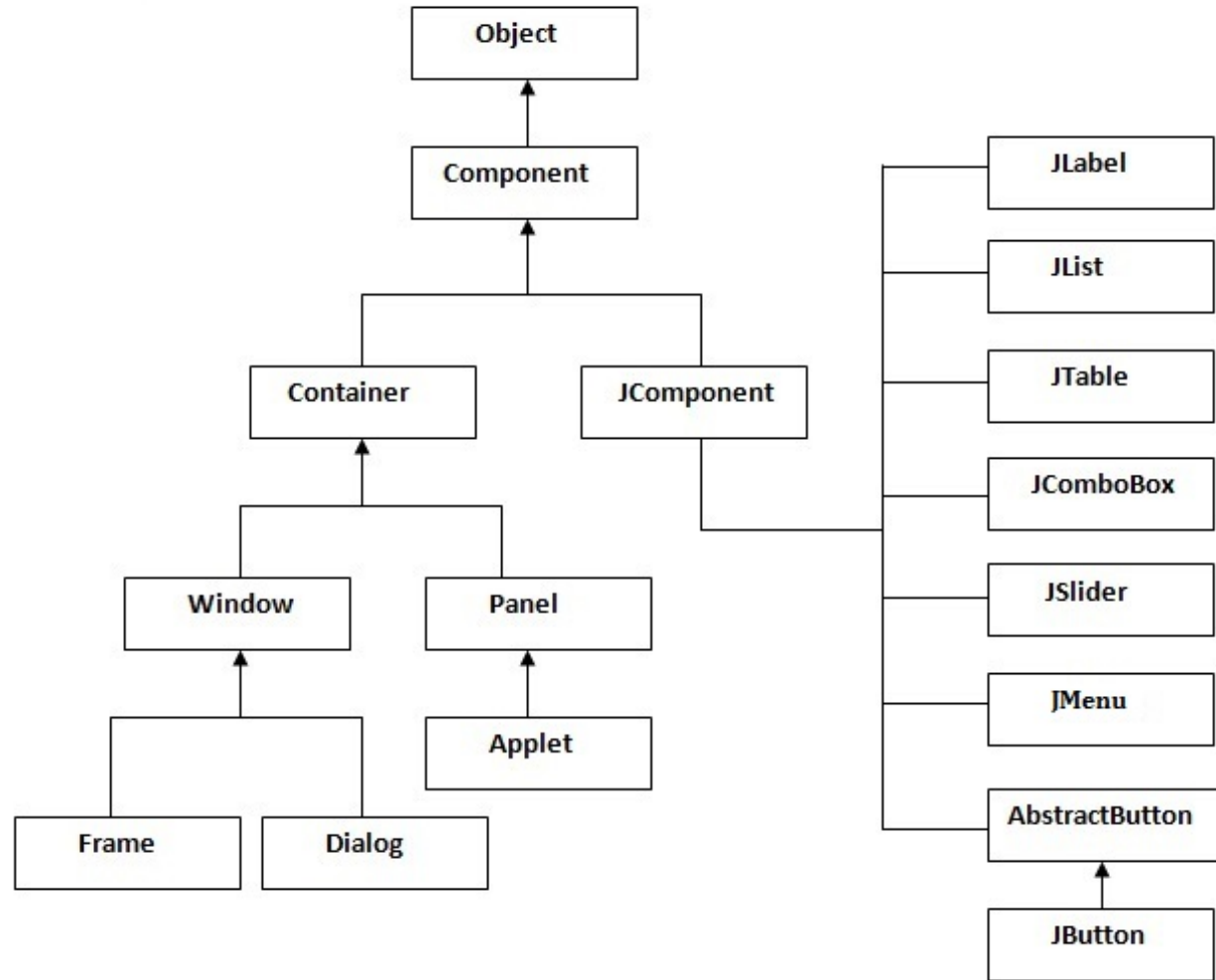
5

- Every SWING controls inherits properties from the following Component class hierarchy.

S.No.	Class & Description
1	<b>Component</b>  A Component is the abstract base class for the non menu user-interface controls of SWING. Component represents an object with graphical representation
2	<b>Container</b>  A Container is a component that can contain other SWING components
3	<b>JComponent</b>  A JComponent is a base class for all SWING UI components. In order to use a SWING component that inherits from JComponent, the component must be in a containment hierarchy whose root is a top-level SWING container

# Hierarchy of **Java Swing** classes

The hierarchy of java swing API is given below.





# Commonly used Methods of Component class

7

The methods of Component class are widely used in java swing that are given below.

Method	Description
<code>public void add(Component c)</code>	add a component on another component.
<code>public void setSize(int width,int height)</code>	sets size of the component.
<code>public void setLayout(LayoutManager m)</code>	sets the layout manager for the component.
<code>public void setVisible(boolean b)</code>	sets the visibility of the component. It is by default false.

# Java Swing Examples

8

- There are two ways to create a frame:
  - By creating the object of Frame class (association)
  - By extending Frame class (inheritance)
- We can write the code of swing inside the main(), constructor or any other method.





```
import javax.swing.*;

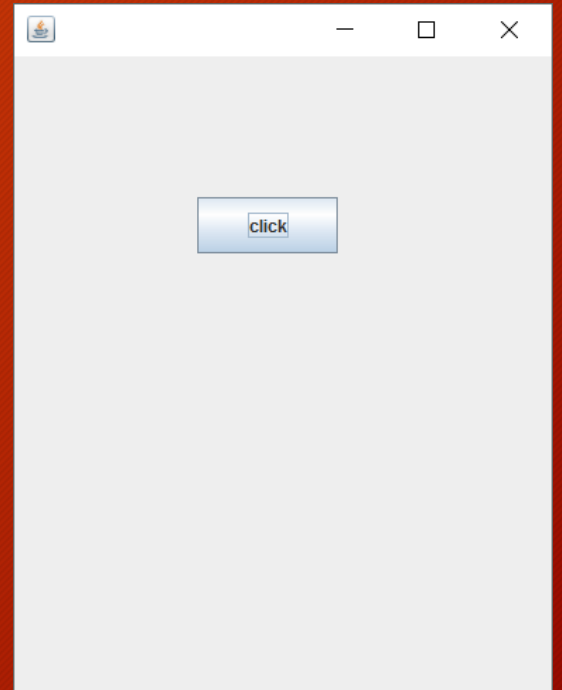
/**
 * @author daonm
 * Simple Java Swing Example
 */
public class FirstSwingExample {

    public static void main(String[] args) {
        JFrame f = new JFrame();//creating instance of JFrame

        JButton b = new JButton("click");//creating instance of JButton
        b.setBounds(130, 100, 100, 40);//x axis, y axis, width, height

        f.add(b);//adding button in JFrame

        f.setSize(400, 500);//400 width and 500 height
        f.setLayout(null);//using no layout managers
        f.setVisible(true);//making the frame visible
    }
}
```



```
package mypack;
import javax.swing.*;

/**
 * @author daonm
 * Example of Swing by Association inside constructor
 */
public class Simple {
    JFrame f;

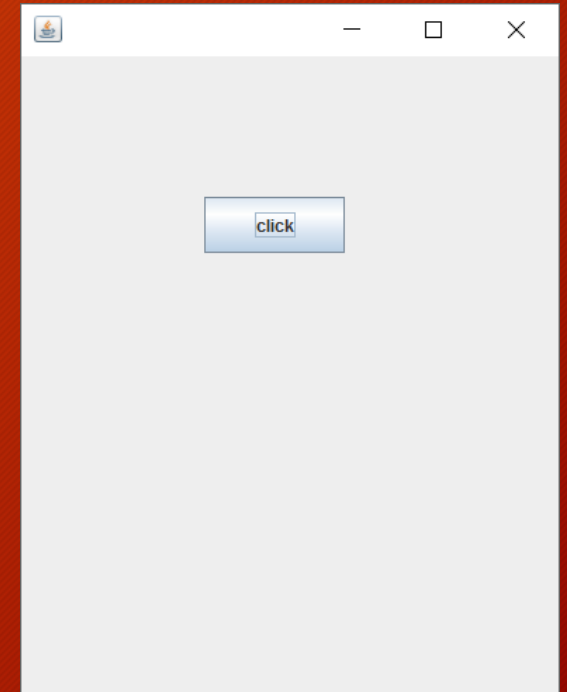
    public Simple() {
        f = new JFrame(); //creating instance of JFrame

        JButton b = new JButton("click"); //creating instance of JButton
        b.setBounds(130, 100, 100, 40);

        f.add(b); //adding button in JFrame

        f.setSize(400, 500); //400 width and 500 height
        f.setLayout(null); //using no layout managers
        f.setVisible(true); //making the frame visible
    }

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



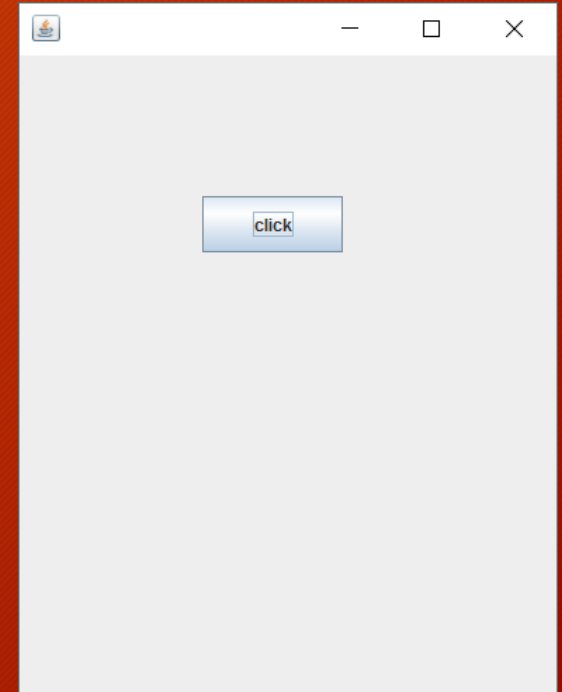


```
package mypack;
import javax.swing.*;

/**
 * @author daonm
 * Simple example of Swing by inheritance
 */
public class Simple2 extends JFrame { //inheriting JFrame
    JFrame f;
    public Simple2() {
        JButton b = new JButton("click"); //create button
        b.setBounds(130, 100, 100, 40);

        add(b); //adding button on frame
        setSize(400, 500);
        setLayout(null);
        setVisible(true);
    }

    public static void main(String[] args) {
        new Simple2();
    }
}
```



# MVC Architecture

13

- Swing API architecture follows loosely based MVC architecture in the following manner.
  - **Model** represents component's data.
  - **View** represents visual representation of the component's data.
  - **Controller** takes the input from the user on the view and reflects the changes in Component's data.
- Swing component has **Model** as a separate element, while the **View** and **Controller** part are clubbed in the User Interface elements. Because of which, Swing has a pluggable look-and-feel architecture.

# SWING - Controls

14



- Every user interface considers the following three main aspects –
  - **UI Elements** – These are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex, which we will cover in this tutorial.
  - **Layouts** – They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in the Layout chapter.
  - **Behavior** – These are the events which occur when the user interacts with UI elements. This part will be covered in the Event Handling chapter.

# SWING UI Elements

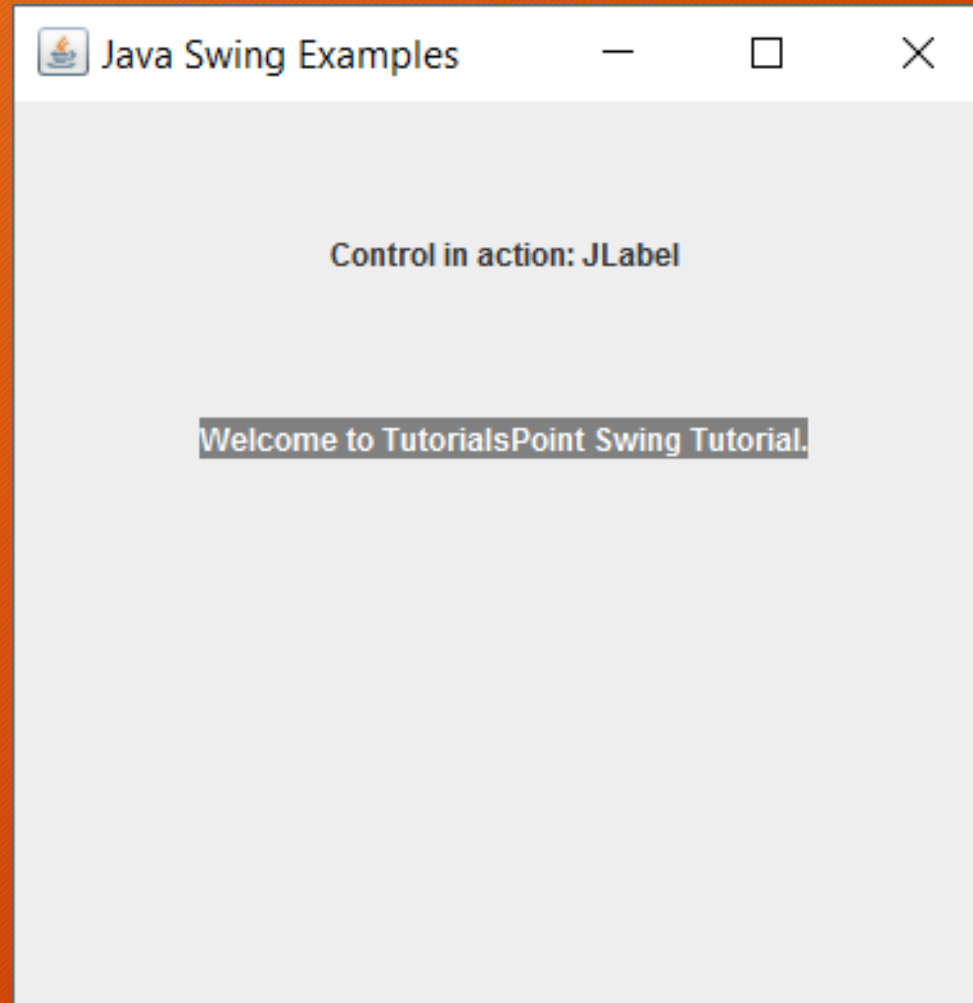
16

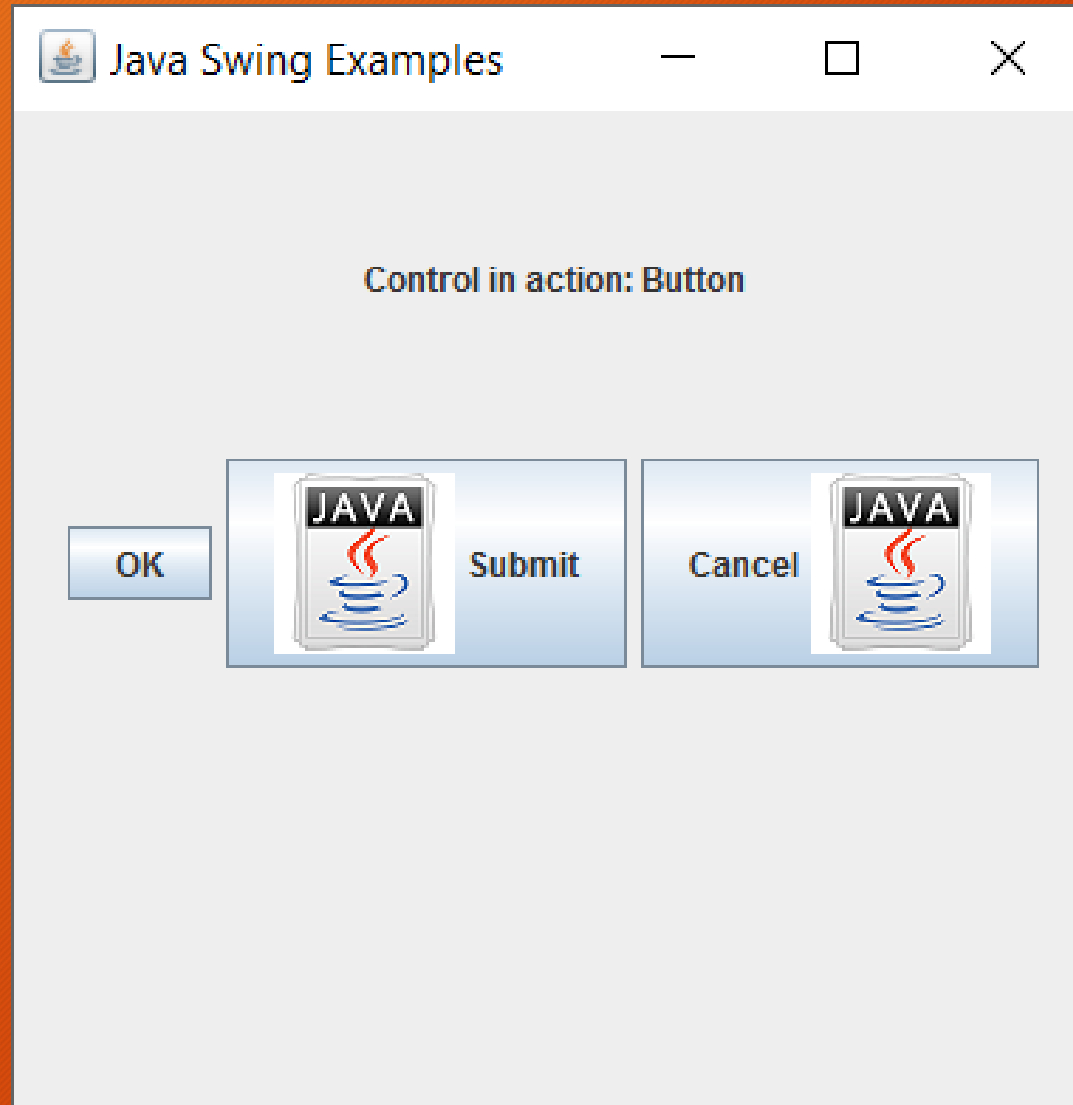
S.No.	Class & Description				
1	<b>JLabel</b> <a href="#">↗</a> A JLabel object is a component for placing text in a container.	6	<b>JList</b> <a href="#">↗</a> A JList component presents the user with a scrolling list of text items.	12	<b>JScrollbar</b> <a href="#">↗</a> A Scrollbar control represents a scroll bar component in order to enable the user to select from range of values.
2	<b>JButton</b> <a href="#">↗</a> This class creates a labeled button.	7	<b>JComboBox</b> <a href="#">↗</a> A JComboBox component presents the user with a to show up menu of choices.	13	<b>JOptionPane</b> <a href="#">↗</a> JOptionPane provides set of standard dialog boxes that prompt users for a value or informs them of something.
3	<b>JColorChooser</b> <a href="#">↗</a> A JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.	8	<b>JTextField</b> <a href="#">↗</a> A JTextField object is a text component that allows for the editing of a single line of text.	14	<b>JFileChooser</b> <a href="#">↗</a> A JFileChooser control represents a dialog window from which the user can select a file.
4	<b>JCheck Box</b> <a href="#">↗</a> A JCheckBox is a graphical component that can be in either an <b>on</b> (true) or <b>off</b> (false) state.	9	<b>JPasswordField</b> <a href="#">↗</a> A JPasswordField object is a text component specialized for password entry.	15	<b>JProgressBar</b> <a href="#">↗</a> As the task progresses towards completion, the progress bar displays the task's percentage of completion.
5	<b>JRadioButton</b> <a href="#">↗</a> The JRadioButton class is a graphical component that can be in either an <b>on</b> (true) or <b>off</b> (false) state. in a group.	10	<b>JTextArea</b> <a href="#">↗</a> A JTextArea object is a text component that allows editing of a multiple lines of text.	16	<b>JSlider</b> <a href="#">↗</a> A JSlider lets the user graphically select a value by sliding a knob within a bounded interval.
		11	<b>ImageIcon</b> <a href="#">↗</a> A ImageIcon control is an implementation of the Icon interface that paints Icons from Images	17	<b>JSpinner</b> <a href="#">↗</a> A JSpinner is a single line input field that lets the user select a number or an object value from an ordered sequence.

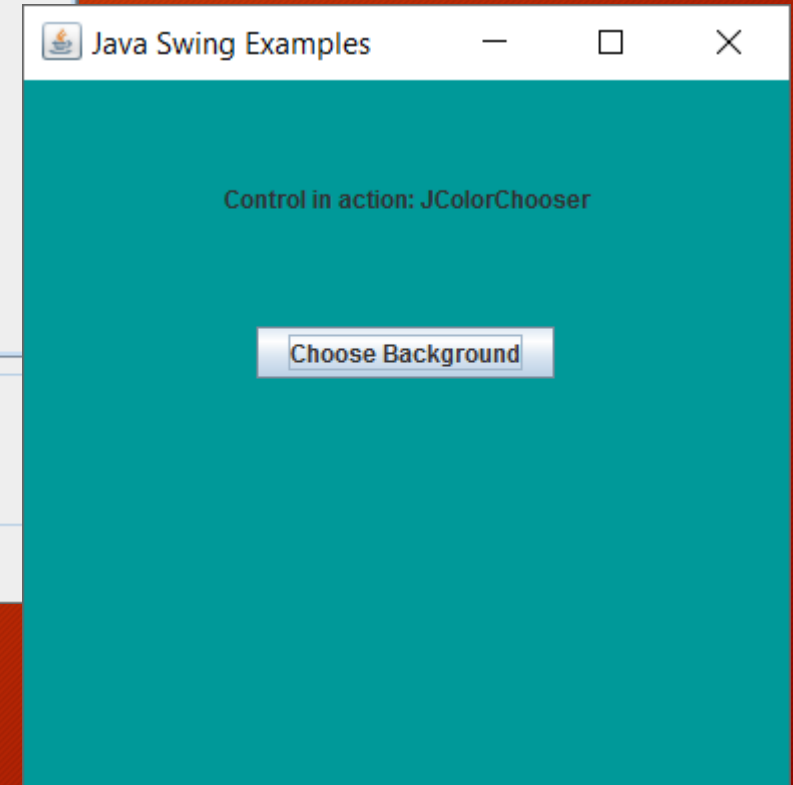
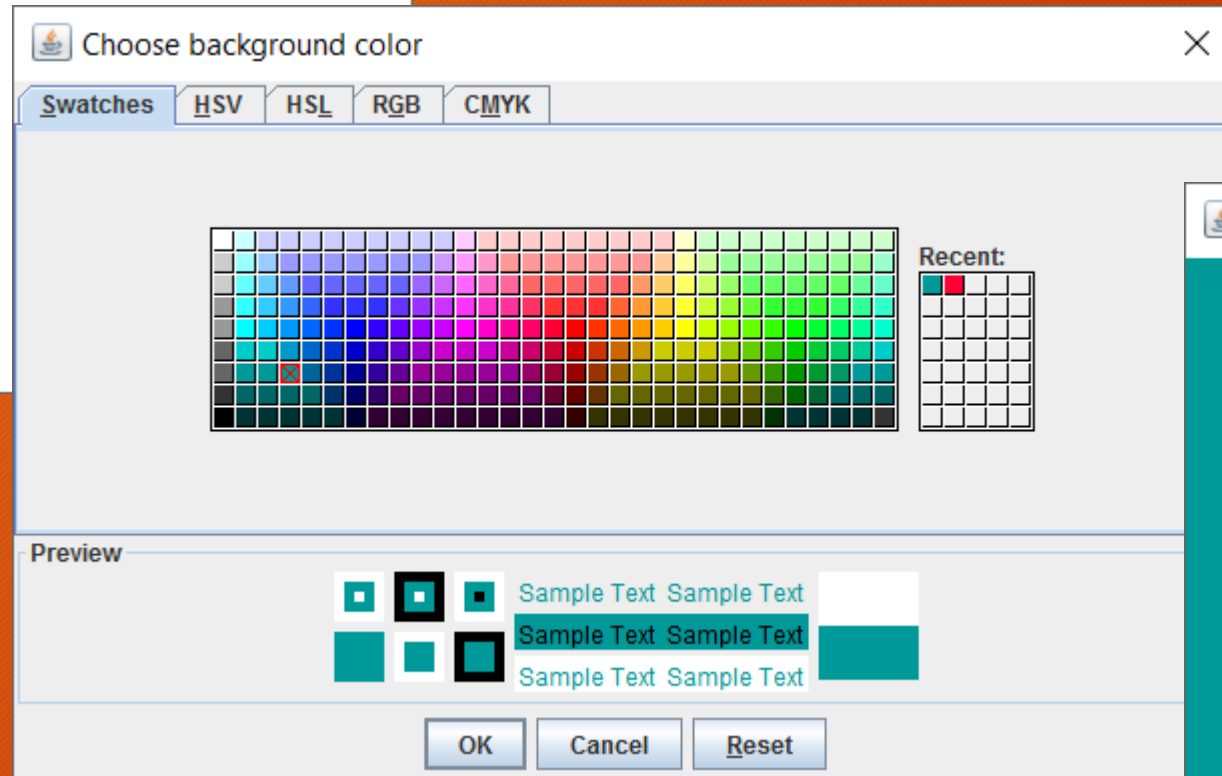
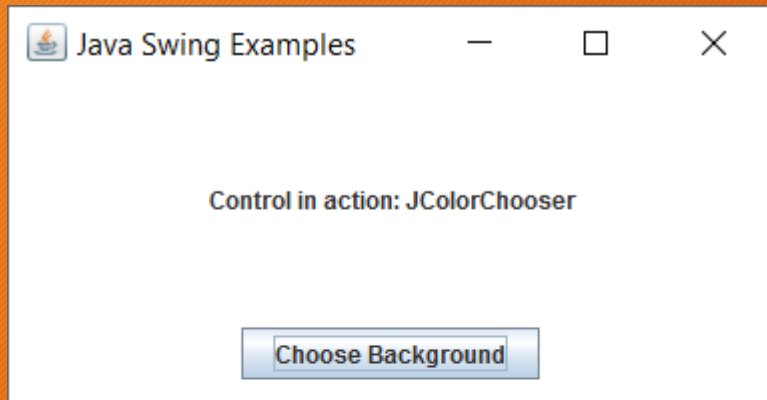
Lecturer: NGUYEN MINH DAO



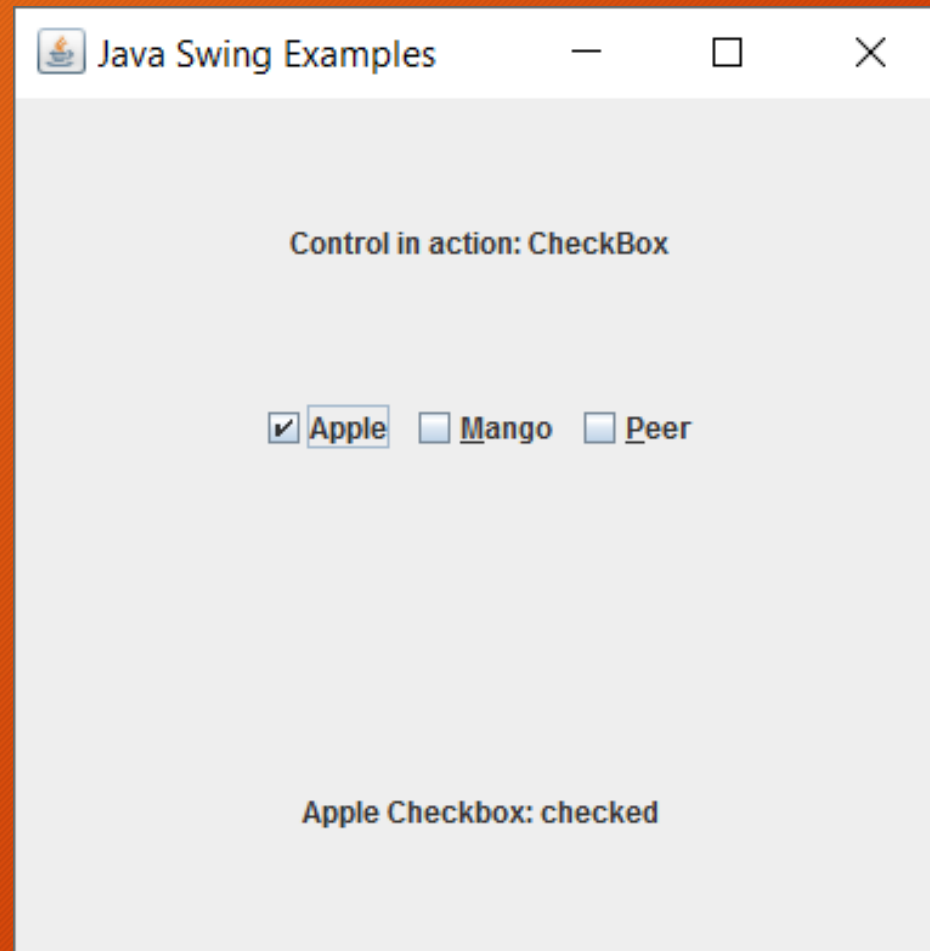


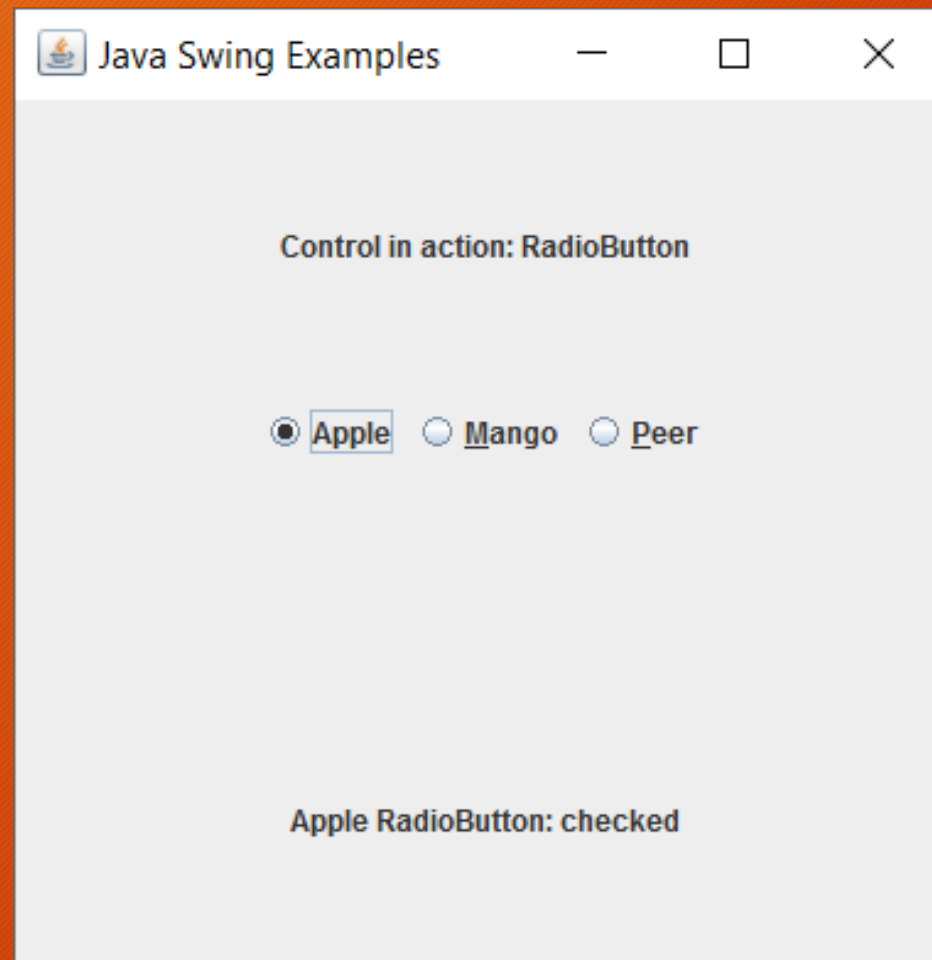


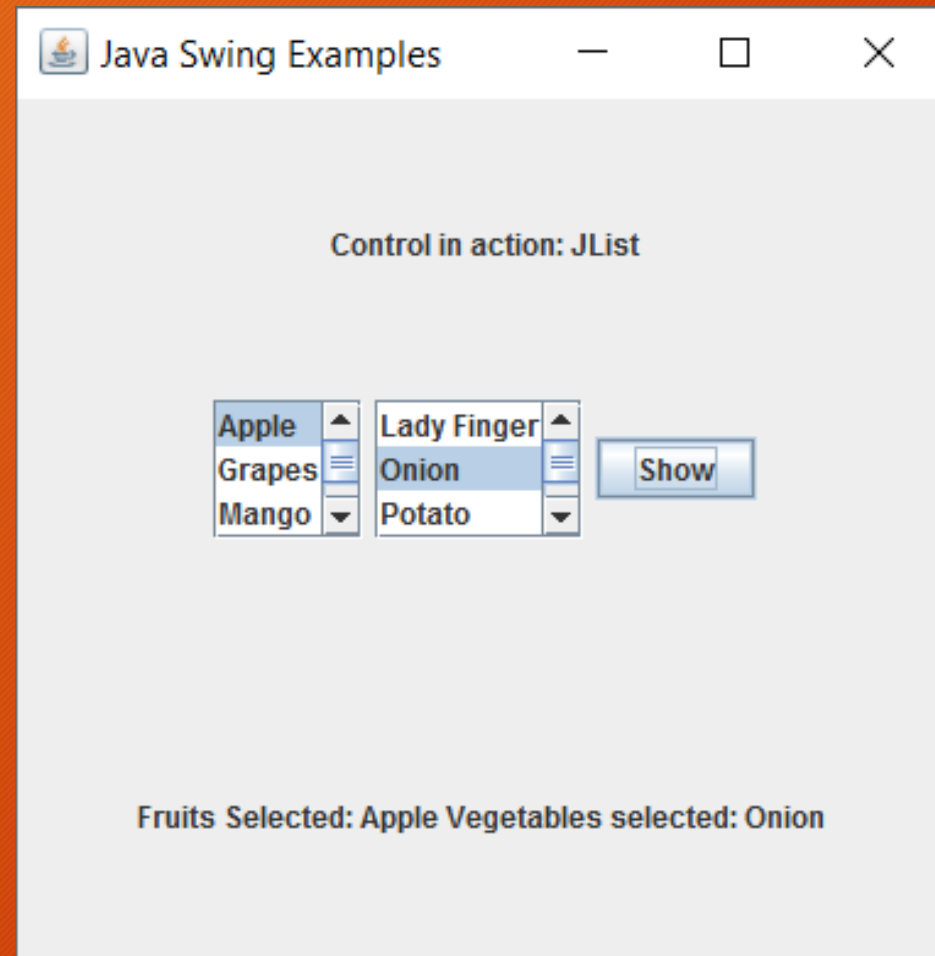




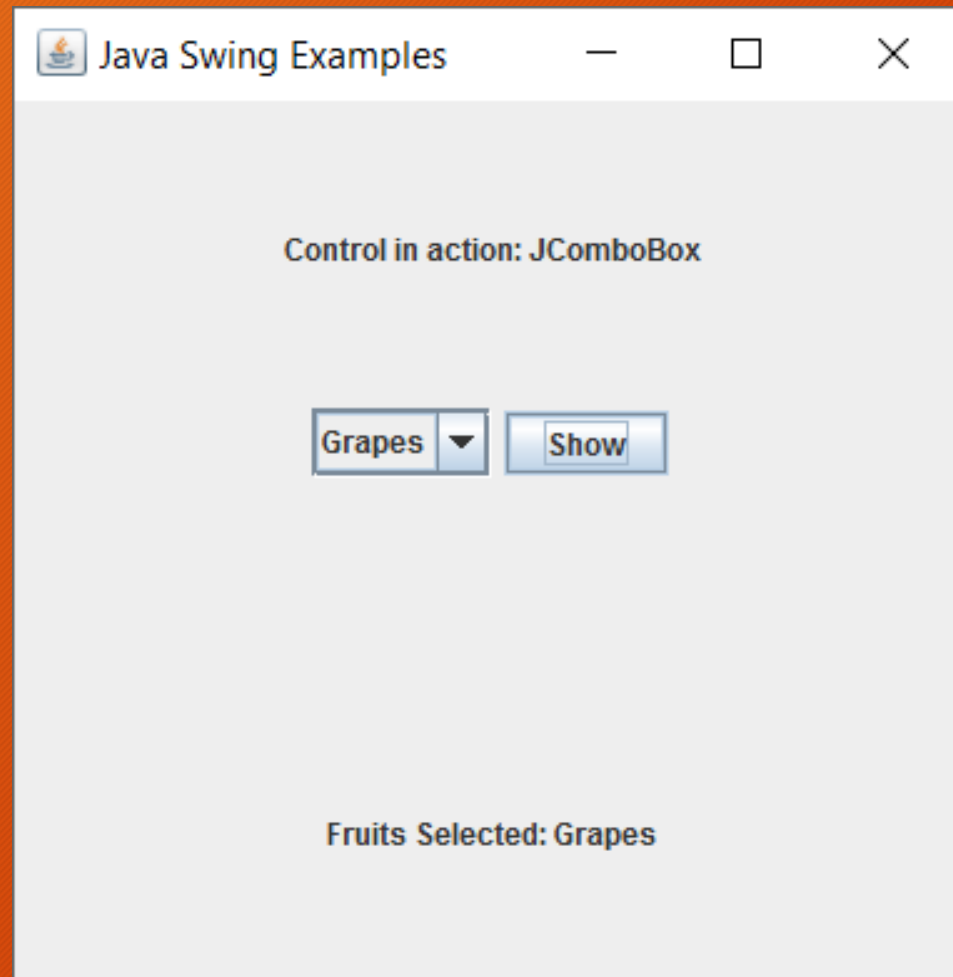












Java Swing Examples

Control in action: JTextField

User ID:  Password:

Username daonm, Password: ãâêô

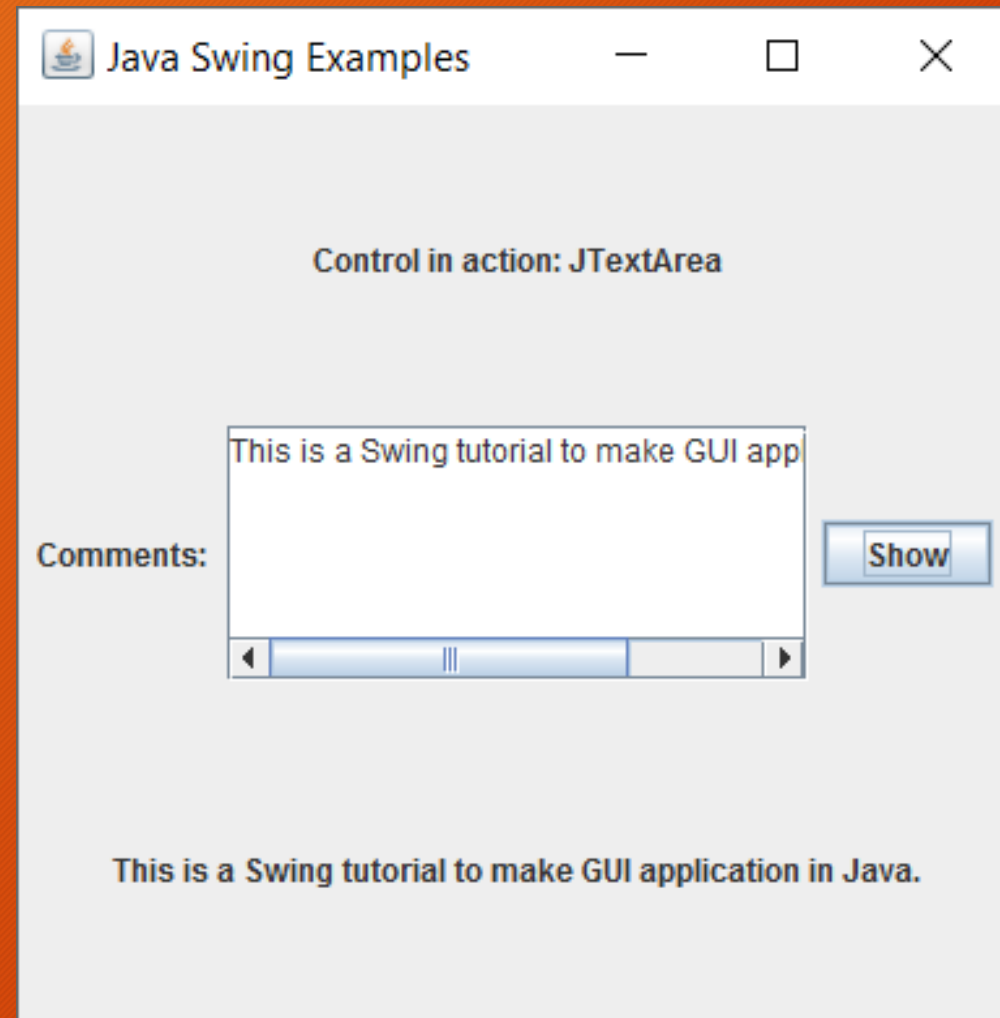
Java Swing Examples

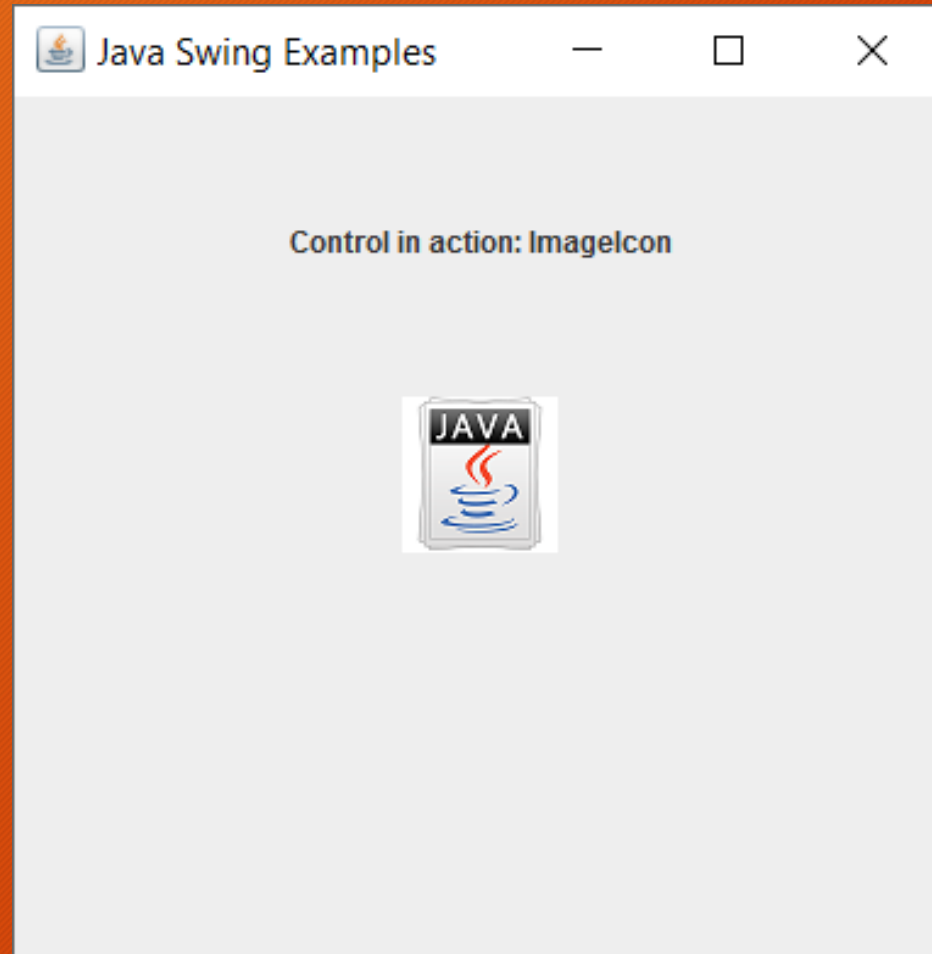
Control in action: JPasswordField

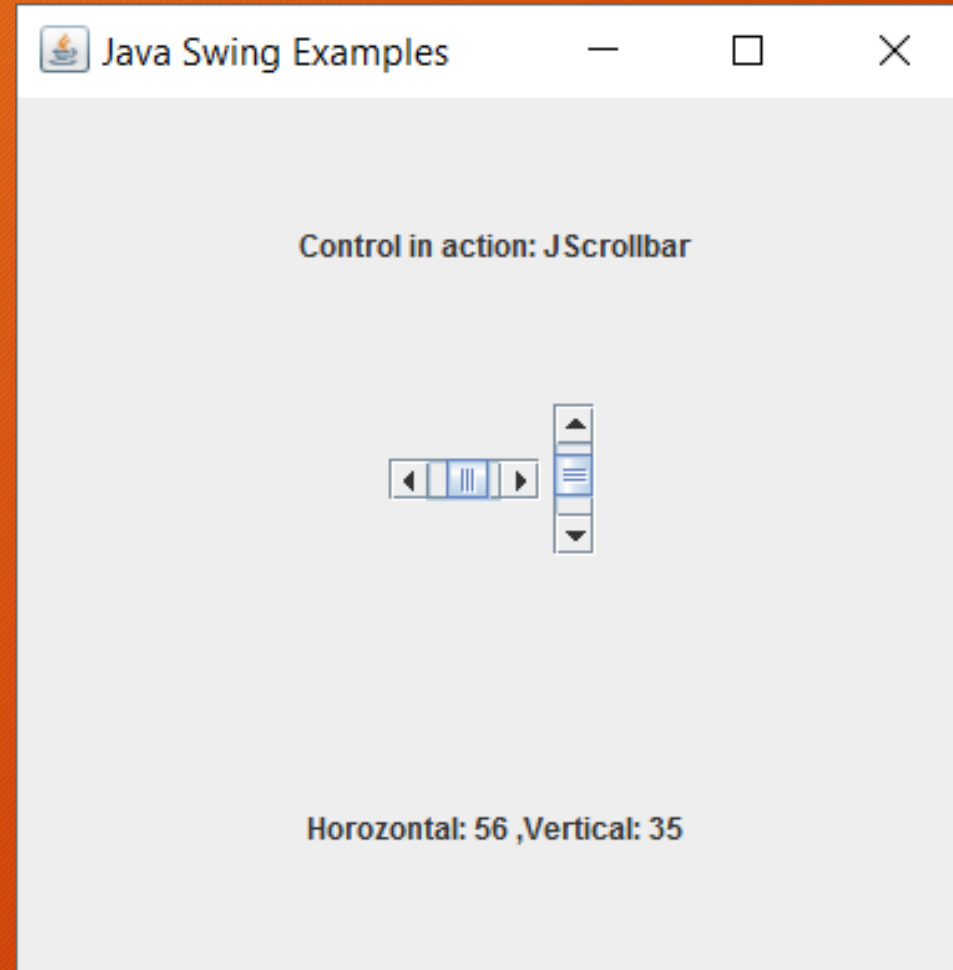
User ID:  Password:

Username daonm, Password: ăăêô

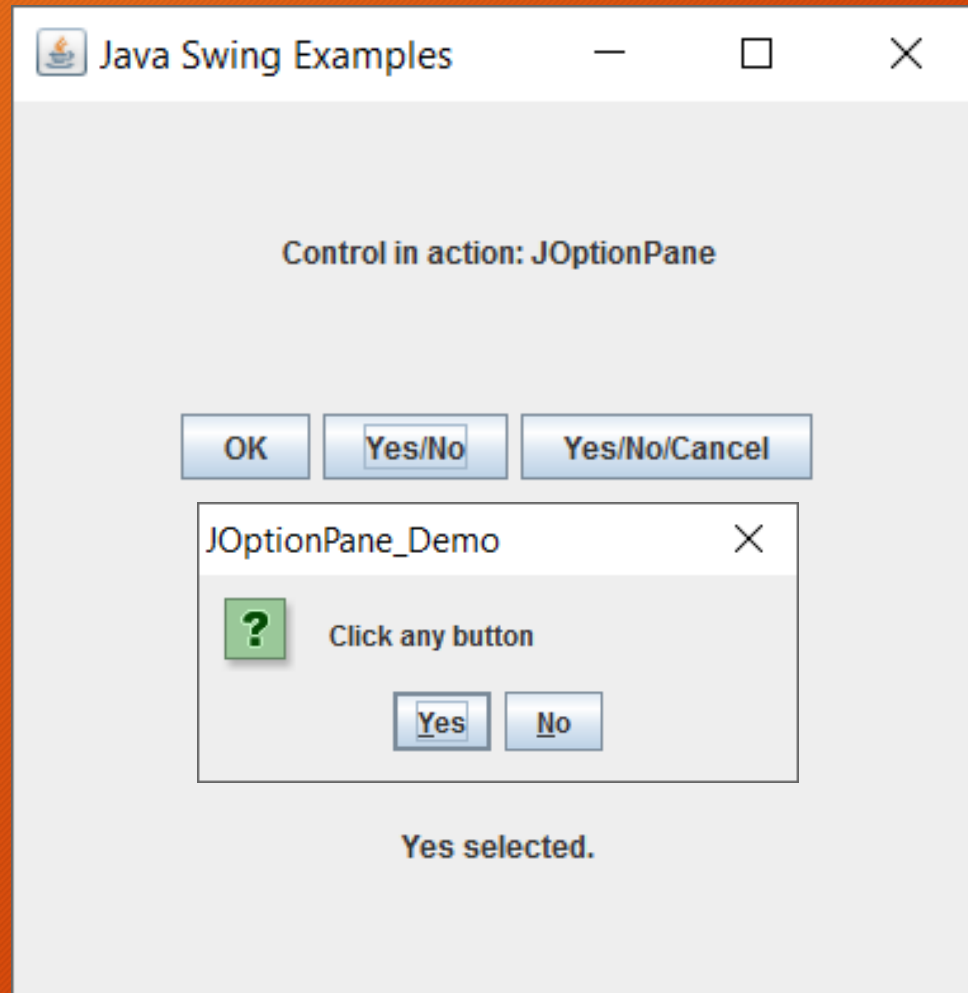


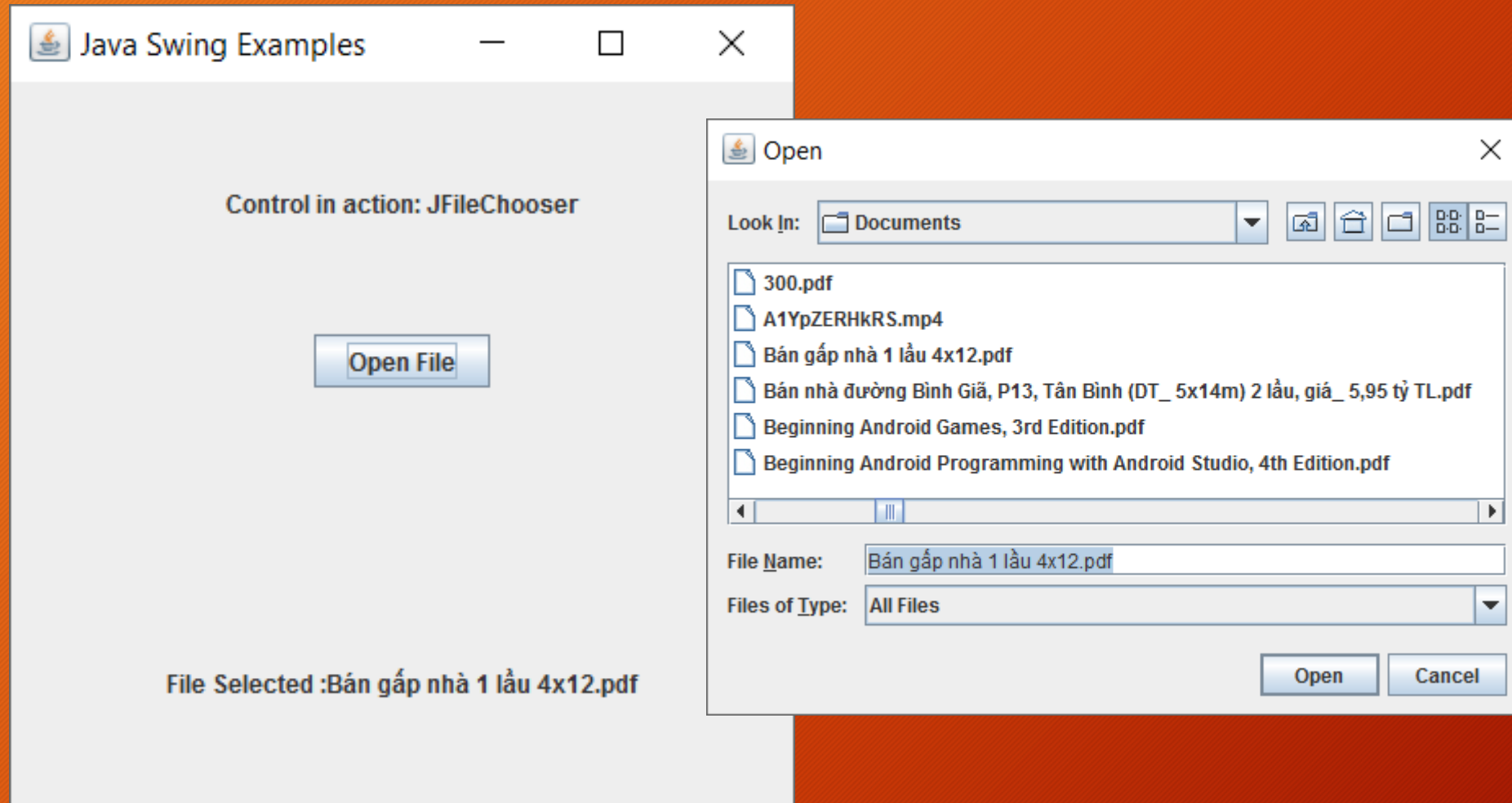


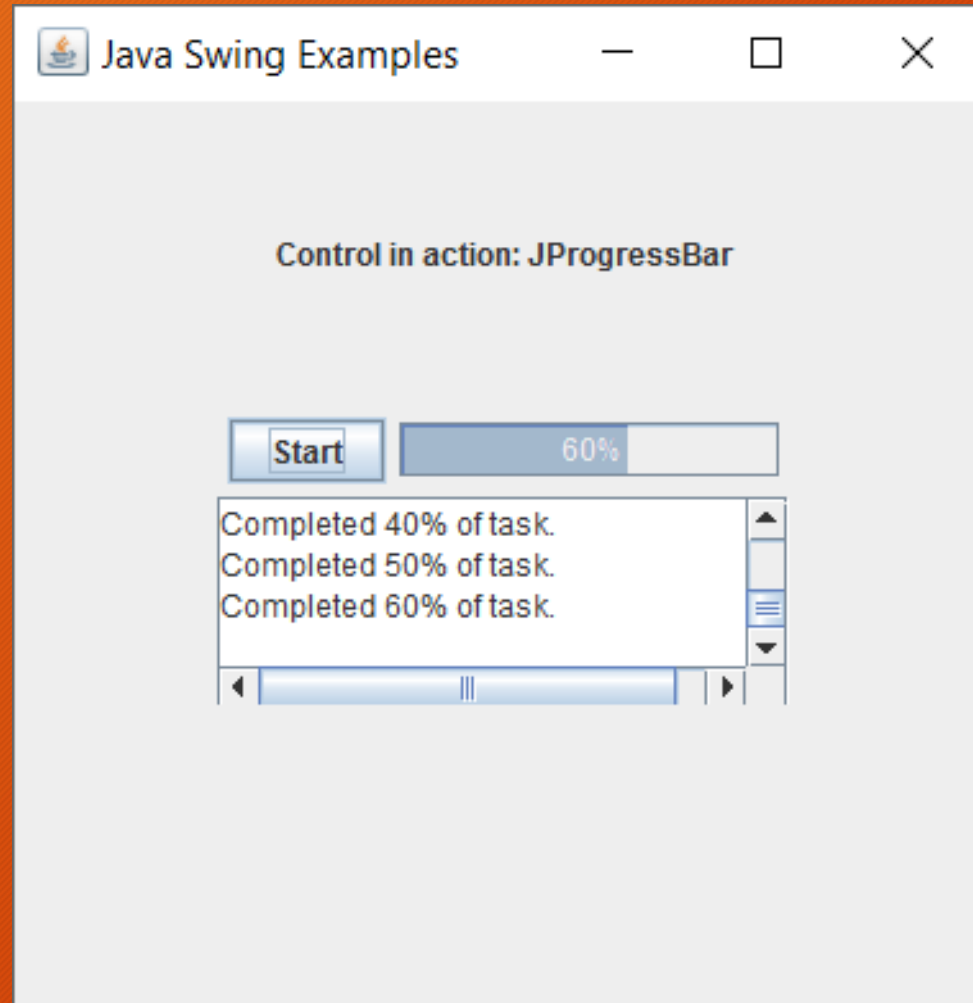




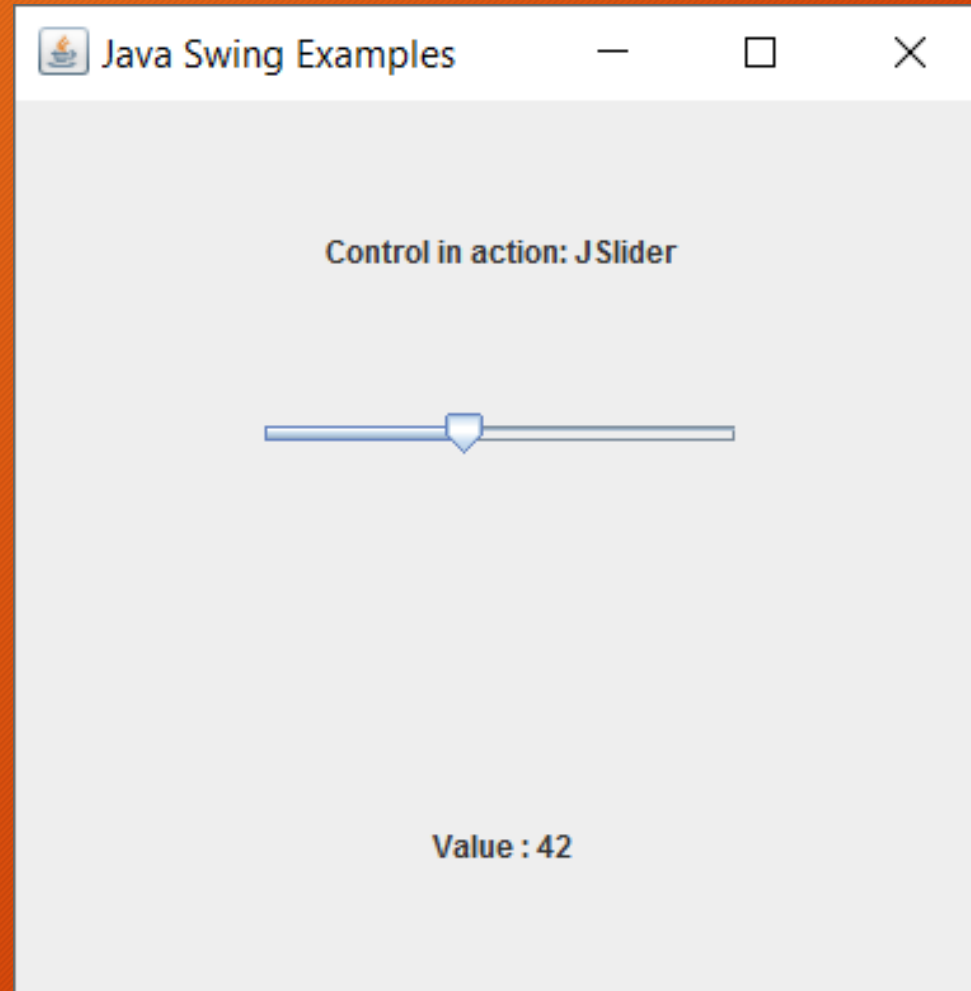


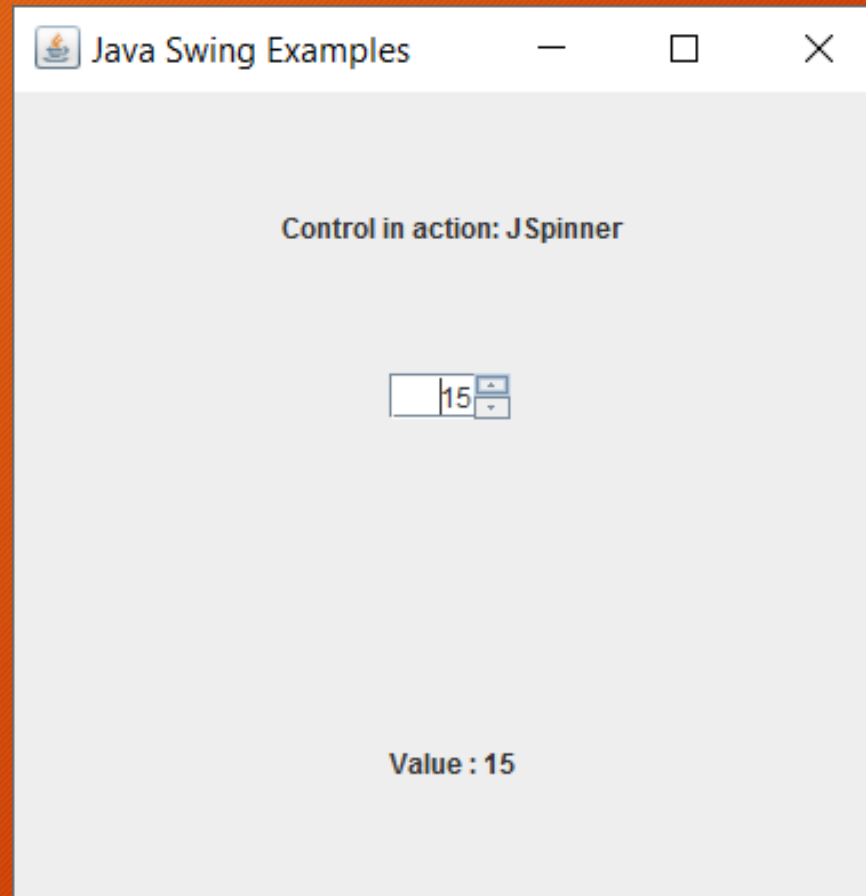












# SWING-Events

35



# What is an **Event**?

36

- Change in the state of an object is known as **Event**, i.e., event describes the change in the state of the source.
- Events are generated as a result of user interaction with the graphical user interface components.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from the list, and scrolling the page are the activities that causes an event to occur.

# Types of Event

37

- The events can be broadly classified into two categories –
  - **Foreground Events** – These events require direct interaction of the user. They are generated as consequences of a person interacting with the graphical components in the Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page, etc.
  - **Background Events** – These events require the interaction of the end user. Operating system interrupts, hardware or software failure, timer expiration, and operation completion are some examples of background events.



# What is **Event Handling**?

38

- **Event Handling** is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has a code which is known as an event handler, that is executed when an event occurs.
- Java uses the **Delegation Event Model** to handle the events. This model defines the standard mechanism to generate and handle the events.



# What is Event Handling?

39

- The **Delegation Event Model** has the following key participants.
  - **Source** – The source is an object on which the event occurs. Source is responsible for providing information of the occurred event to its handler. Java provides us with classes for the source object.
  - **Listener** – It is also known as event handler. The listener is responsible for generating a response to an event. From the point of view of Java implementation, the listener is also an object. The listener waits till it receives an event. Once the event is received, the listener processes the event and then returns.

# Steps Involved in Event Handling

40

- **Step 1** – The user clicks the button and the event is generated.
- **Step 2** – The object of concerned event class is created automatically and information about the source and the event get populated within the same object.
- **Step 3** – Event object is forwarded to the method of the registered listener class.
- **Step 4** – The method is gets executed and returns.



# Points to Remember About the **Listener**

41

- In order to design a listener class, you have to develop some **listener interfaces**. These **Listener interfaces** forecast some public abstract callback methods, which must be implemented by the listener class.
- If you do not implement any of the predefined interfaces, then your class cannot act as a listener class for a source object.



# Callback Methods

42

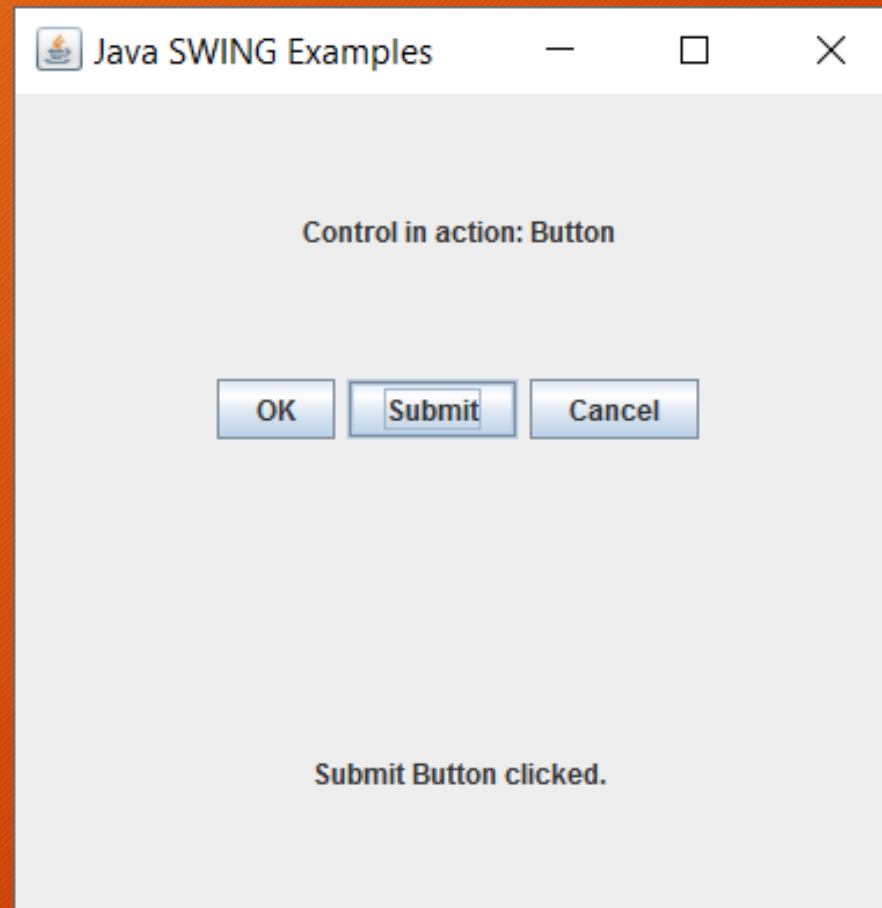
- These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represent an event method. In response to an event, java jre will fire callback method. All such callback methods are provided in listener interfaces.
- If a component wants some listener to listen ot its events, the source must register itself to the listener.





# Event Handling Example

44





```
private class ButtonClickListener implements ActionListener{
    @Override
    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();

        if( command.equals( "OK" )) {
            statusLabel.setText("Ok Button clicked.");
        } else if( command.equals( "Submit" ) ) {
            statusLabel.setText("Submit Button clicked.");
        } else {
            statusLabel.setText("Cancel Button clicked.");
        }
    }
}
```

```
private void showEventDemo() {  
    headerLabel.setText("Control in action: Button");  
  
    JButton okButton = new JButton("OK");  
    JButton submitButton = new JButton("Submit");  
    JButton cancelButton = new JButton("Cancel");  
  
    okButton.setActionCommand("OK");  
    submitButton.setActionCommand("Submit");  
    cancelButton.setActionCommand("Cancel");  
  
    okButton.addActionListener(new ButtonClickListener());  
    submitButton.addActionListener(new ButtonClickListener());  
    cancelButton.addActionListener(new ButtonClickListener());  
  
    controlPanel.add(okButton);  
    controlPanel.add(submitButton);  
    controlPanel.add(cancelButton);  
  
    mainFrame.setVisible(true);  
}
```

# SWING - Event Classes

47

- **Event classes represent the event.** Java provides various Event classes, however, only those which are more frequently used will be discussed.
- **EventObject Class**
  - It is the root class from which all event state objects shall be derived. All Events are constructed with a reference to the object, the **source**, that is logically deemed to be the object upon which the Event in question initially occurred upon. This class is defined in `java.util` package.



# SWING Event Classes

48

Sr.No.	Class & Description
1	<b>AWTEvent</b> <a href="#">↗</a> It is the root event class for all SWING events. This class and its subclasses supercede the original <b>java.awt.Event</b> class.
2	<b>ActionEvent</b> <a href="#">↗</a> The ActionEvent is generated when the button is clicked or the item of a list is double-clicked.
3	<b>InputEvent</b> <a href="#">↗</a> The InputEvent class is the root event class for all component-level input events.
4	<b>KeyEvent</b> <a href="#">↗</a> On entering the character the Key event is generated.
5	<b>MouseEvent</b> <a href="#">↗</a> This event indicates a mouse action occurred in a component.

6	<b>WindowEvent</b> <a href="#">↗</a> The object of this class represents the change in the state of a window.
7	<b>AdjustmentEvent</b> <a href="#">↗</a> The object of this class represents the adjustment event emitted by Adjustable objects.
8	<b>ComponentEvent</b> <a href="#">↗</a> The object of this class represents the change in the state of a window.
9	<b>ContainerEvent</b> <a href="#">↗</a> The object of this class represents the change in the state of a window.
10	<b>MouseMotionEvent</b> <a href="#">↗</a> The object of this class represents the change in the state of a window.
11	<b>PaintEvent</b> <a href="#">↗</a> The object of this class represents the change in the state of a window.






# SWING - Event Listeners






49

- Event listeners represent the interfaces responsible to handle events. Java provides various Event listener classes, however, only those which are more frequently used will be discussed. Every method of an event listener method has a single argument as an object which is the subclass of **EventObject** class.
- For example, mouse event listener methods will accept instance of **MouseEvent**, where **MouseEvent** derives from **EventObject**.

# SWING Event Listener Interfaces

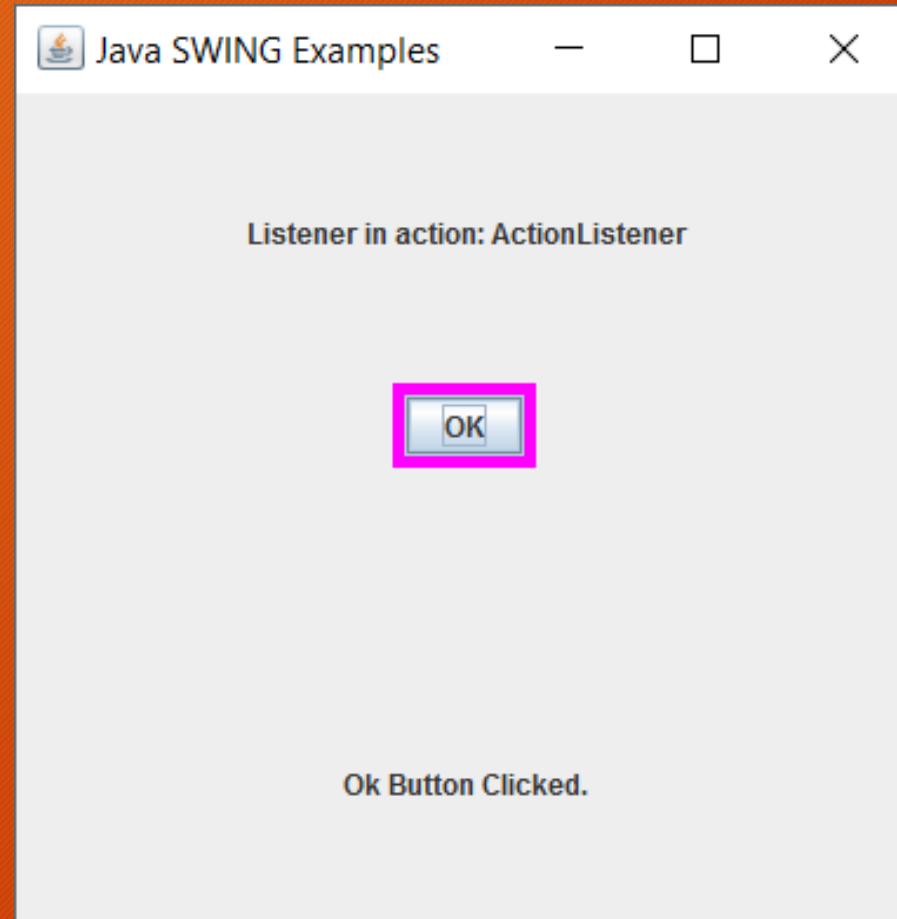
50

Sr.No.	Class & Description
1	<b>ActionListener</b>  This interface is used for receiving the action events.
2	<b>ComponentListener</b>  This interface is used for receiving the component events.
3	<b>ItemListener</b>  This interface is used for receiving the item events.
4	<b>KeyListener</b>  This interface is used for receiving the key events.
5	<b>MouseListener</b>  This interface is used for receiving the mouse events.

6	<b>WindowListener</b>  This interface is used for receiving the window events.
7	<b>AdjustmentListener</b>  This interface is used for receiving the adjustment events.
8	<b>ContainerListener</b>  This interface is used for receiving the container events.
9	<b>MouseMotionListener</b>  This interface is used for receiving the mouse motion events.
10	<b>FocusListener</b>  This interface is used for receiving the focus events.

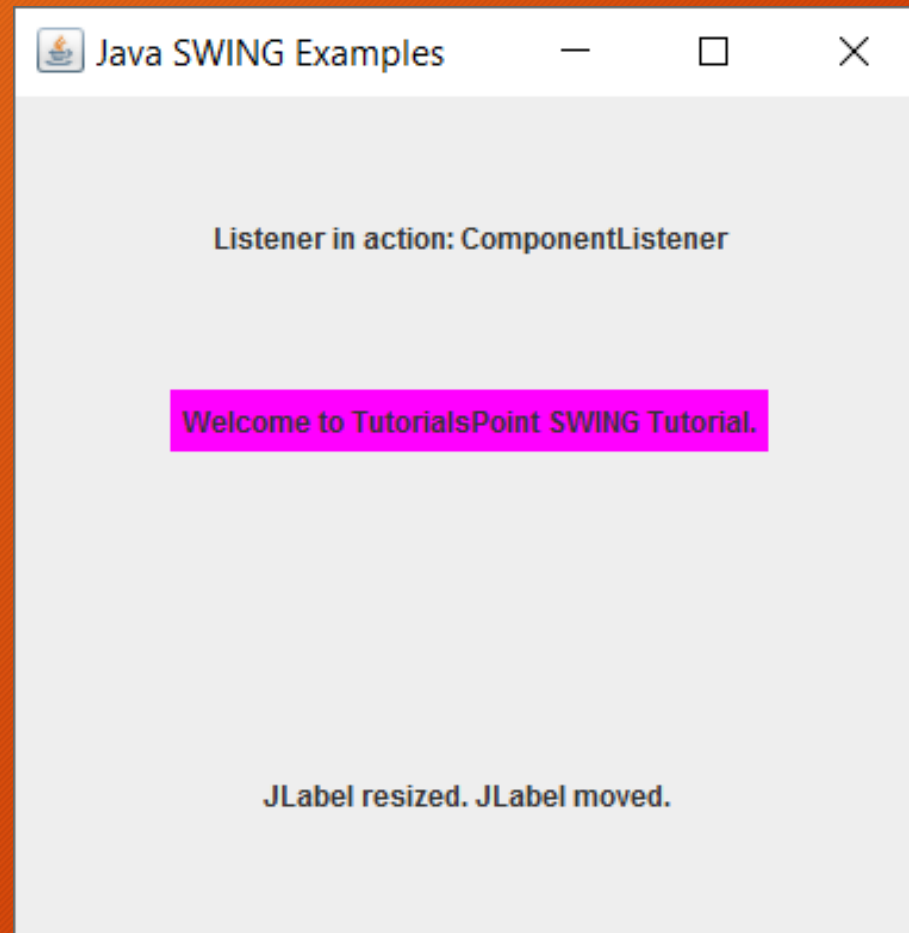




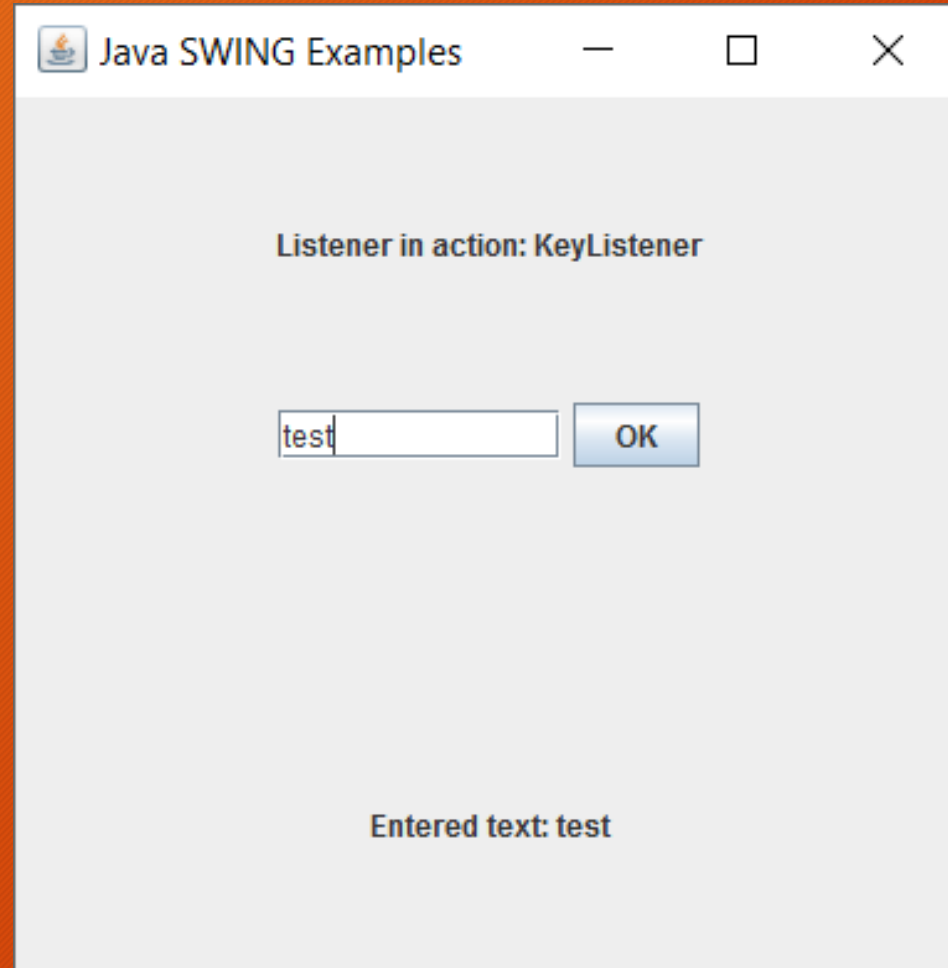


```
private void showActionListenerDemo() {  
    headerLabel.setText("Listener in action: ActionListener");  
  
    JPanel panel = new JPanel();  
    panel.setBackground(Color.magenta);  
    JButton okButton = new JButton("OK");  
  
    okButton.addActionListener(new CustomActionListener());  
    panel.add(okButton);  
    controlPanel.add(panel);  
    mainFrame.setVisible(true);  
}  
  
class CustomActionListener implements ActionListener{  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        statusLabel.setText("Ok Button Clicked.");  
    }  
}
```





```
class CustomComponentListener implements ComponentListener {  
    @Override  
    public void componentResized(ComponentEvent e) {  
        statusLabel.setText(statusLabel.getText()  
            + e.getComponent().getClass().getSimpleName() + " resized. ");  
    }  
    @Override  
    public void componentMoved(ComponentEvent e) {  
        statusLabel.setText(statusLabel.getText()  
            + e.getComponent().getClass().getSimpleName() + " moved. ");  
    }  
    @Override  
    public void componentShown(ComponentEvent e) {  
        statusLabel.setText(statusLabel.getText()  
            + e.getComponent().getClass().getSimpleName() + " shown. ");  
    }  
    @Override  
    public void componentHidden(ComponentEvent e) {  
        statusLabel.setText(statusLabel.getText()  
            + e.getComponent().getClass().getSimpleName() + " hidden. ");  
    }  
}
```





```
class CustomKeyListener implements KeyListener{  
    @Override  
    public void keyTyped(KeyEvent e) {  
    }  
    @Override  
    public void keyPressed(KeyEvent e) {  
        if(e.getKeyCode() == KeyEvent.VK_ENTER) {  
            statusLabel.setText("Entered text: " + textField.getText());  
        }  
    }  
    @Override  
    public void keyReleased(KeyEvent e) {  
    }  
}
```






# SWING - Event Adapters

58

- Adapters are abstract classes for receiving various events.
- The methods in these classes are empty.
- These classes exist as convenience for creating listener objects.

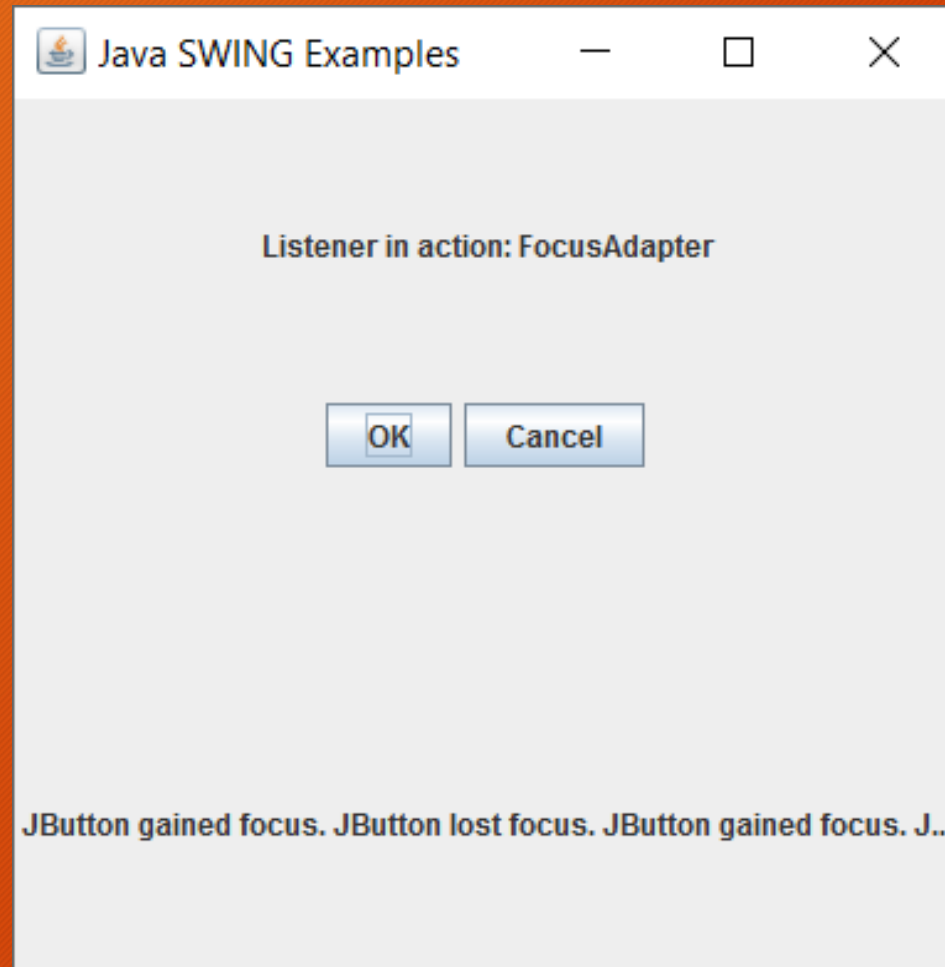
# SWING - Event Adapters

59

Sr.No.	Adapter & Description
1	<b>FocusAdapter</b>  An abstract adapter class for receiving focus events.
2	<b>KeyAdapter</b>  An abstract adapter class for receiving key events.
3	<b>MouseAdapter</b>  An abstract adapter class for receiving mouse events.
4	<b>MouseMotionAdapter</b>  An abstract adapter class for receiving mouse motion events.
5	<b>WindowAdapter</b>  An abstract adapter class for receiving window events.



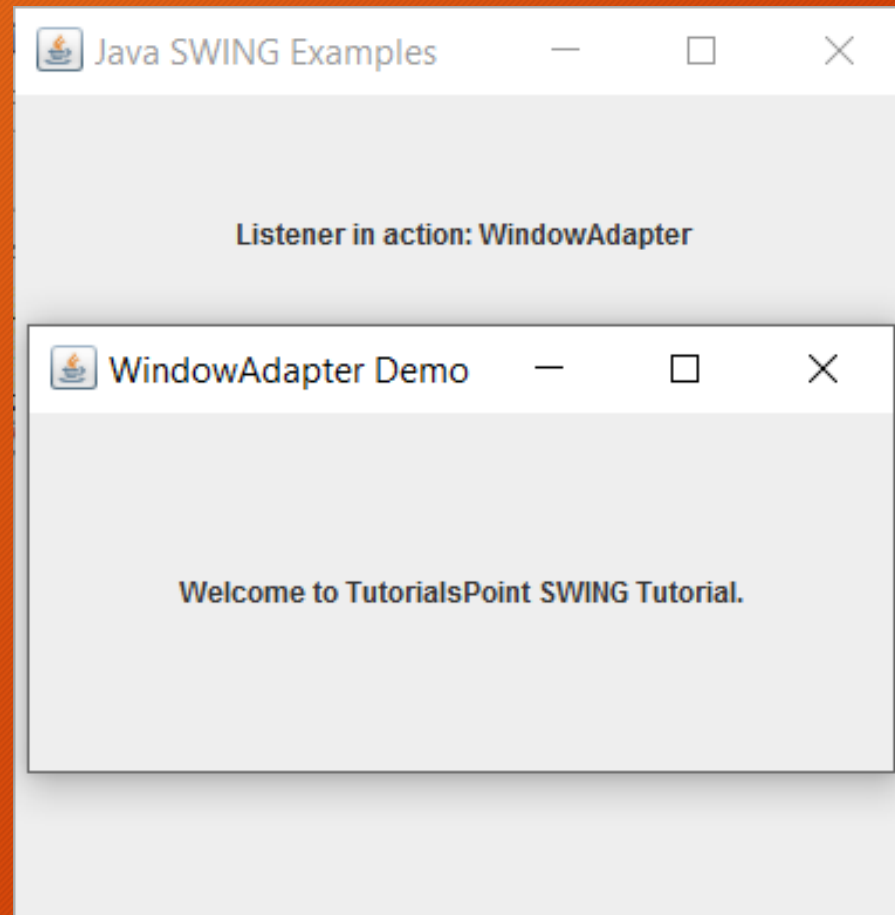




```
private void showFocusAdapterDemo() {  
    headerLabel.setText("Listener in action: FocusAdapter");  
    JButton okButton = new JButton("OK");  
    JButton cancelButton = new JButton("Cancel");  
  
    okButton.addFocusListener(new FocusAdapter() {  
        @Override  
        public void focusGained(FocusEvent e) {  
            statusLabel.setText(statusLabel.getText()  
                + e.getComponent().getClass().getSimpleName()  
                + " gained focus. ");  
        }  
    });  
}
```



```
cancelButton.addFocusListener(new FocusAdapter() {  
    @Override  
    public void focusLost(FocusEvent e) {  
        statusLabel.setText(statusLabel.getText()  
            + e.getComponent().getClass().getSimpleName()  
            + " lost focus. ");  
    }  
});  
controlPanel.add(okButton);  
controlPanel.add(cancelButton);  
mainFrame.setVisible(true);  
}
```



```
private void showWindowAdapterDemo() {  
    headerLabel.setText("Listener in action: WindowAdapter");  
    JButton okButton = new JButton("OK");  
    final JFrame aboutFrame = new JFrame();  
    aboutFrame.setSize(300,200);  
    aboutFrame.setTitle("WindowAdapter Demo");  
  
    aboutFrame.addWindowListener(new WindowAdapter() {  
        @Override  
        public void windowClosing(WindowEvent windowEvent){  
            aboutFrame.dispose();  
        }  
    });  
    JLabel msgLabel  
        = new JLabel("Welcome to Tutorialspoint SWING Tutorial.",JLabel.CENTER);  
    aboutFrame.add(msgLabel);  
    aboutFrame.setVisible(true);  
}
```



# SWING - Layouts

66

- Layout refers to the arrangement of components within the container.
- In another way, it could be said that layout is placing the components at a particular position within the container.
- The task of laying out the controls is done automatically by the **Layout Manager**.

# Layout Manager

68



- Java provides various layout managers to position the controls.
- Properties like size, shape, and arrangement varies from one layout manager to the other.
- When the size of the applet or the application window changes, the size, shape, and arrangement of the components also changes in response, i.e. the layout managers adapt to the dimensions of the appletviewer or the application window.



# Layout Manager

69

- The layout manager is associated with every Container object.
- Each layout manager is an object of the class that implements the **LayoutManager** interface.
- Following are the interfaces defining the functionalities of Layout Managers.

Sr.No.	Interface & Description
1	<b>LayoutManager</b> <a href="#"></a> The LayoutManager interface declares those methods which need to be implemented by the class, whose object will act as a layout manager.
2	<b>LayoutManager2</b> <a href="#"></a> The LayoutManager2 is the sub-interface of the LayoutManager. This interface is for those classes that know how to layout containers based on layout constraint object.

# AWT Layout Manager Classes

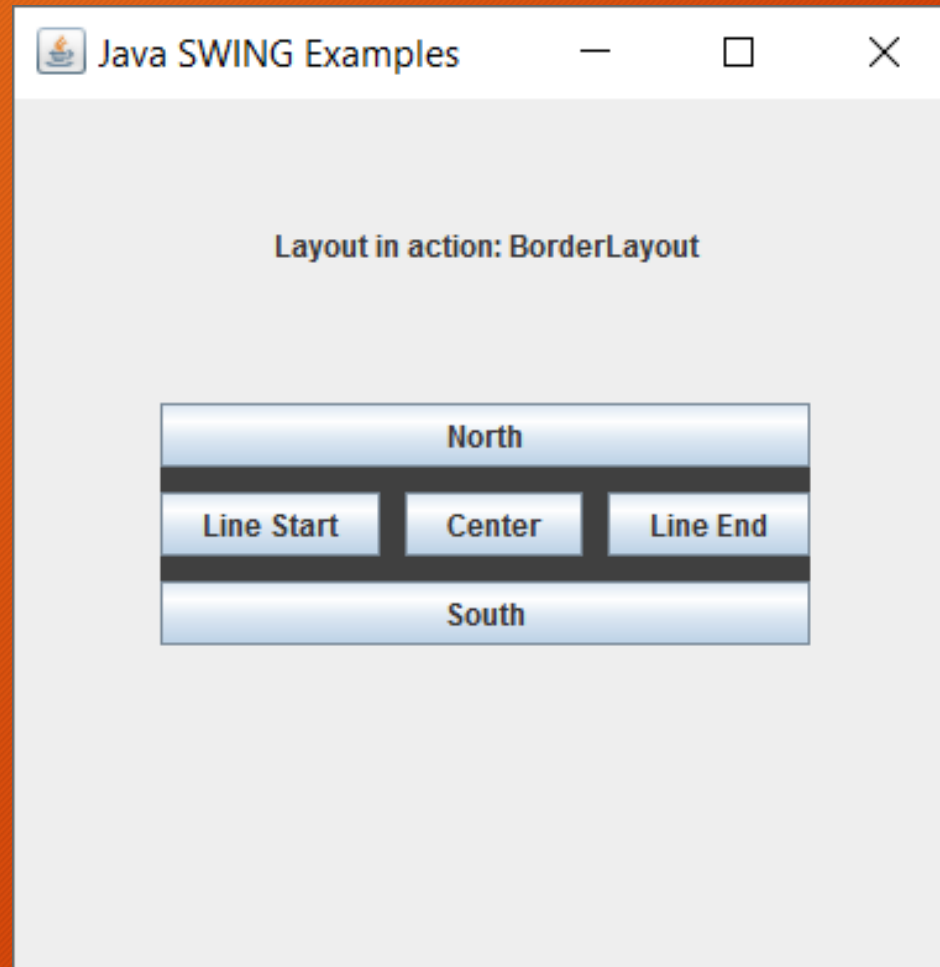
70

Sr.No.	LayoutManager & Description
1	<b>BorderLayout</b> <a href="#">↗</a> The BorderLayout arranges the components to fit in the five regions: east, west, north, south, and center.
2	<b>CardLayout</b> <a href="#">↗</a> The CardLayout object treats each component in the container as a card. Only one card is visible at a time.
3	<b>FlowLayout</b> <a href="#">↗</a> The FlowLayout is the default layout. It layout the components in a directional flow.

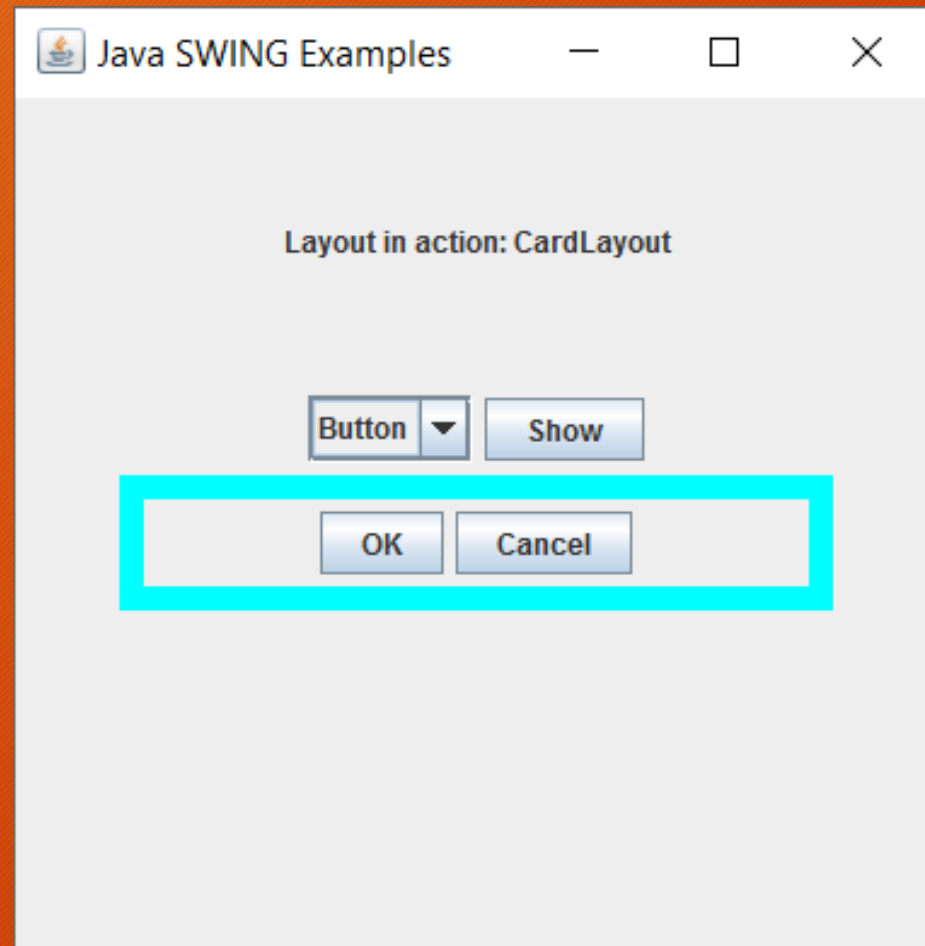
4	<b>GridLayout</b> <a href="#">↗</a> The GridLayout manages the components in the form of a rectangular grid.
5	<b>GridBagLayout</b> <a href="#">↗</a> This is the most flexible layout manager class. The object of GridBagLayout aligns the component vertically, horizontally, or along their baseline without requiring the components of the same size.
6	<b>GroupLayout</b> <a href="#">↗</a> The GroupLayout hierarchically groups the components in order to position them in a Container.
7	<b>SpringLayout</b> <a href="#">↗</a> A SpringLayout positions the children of its associated container according to a set of constraints.





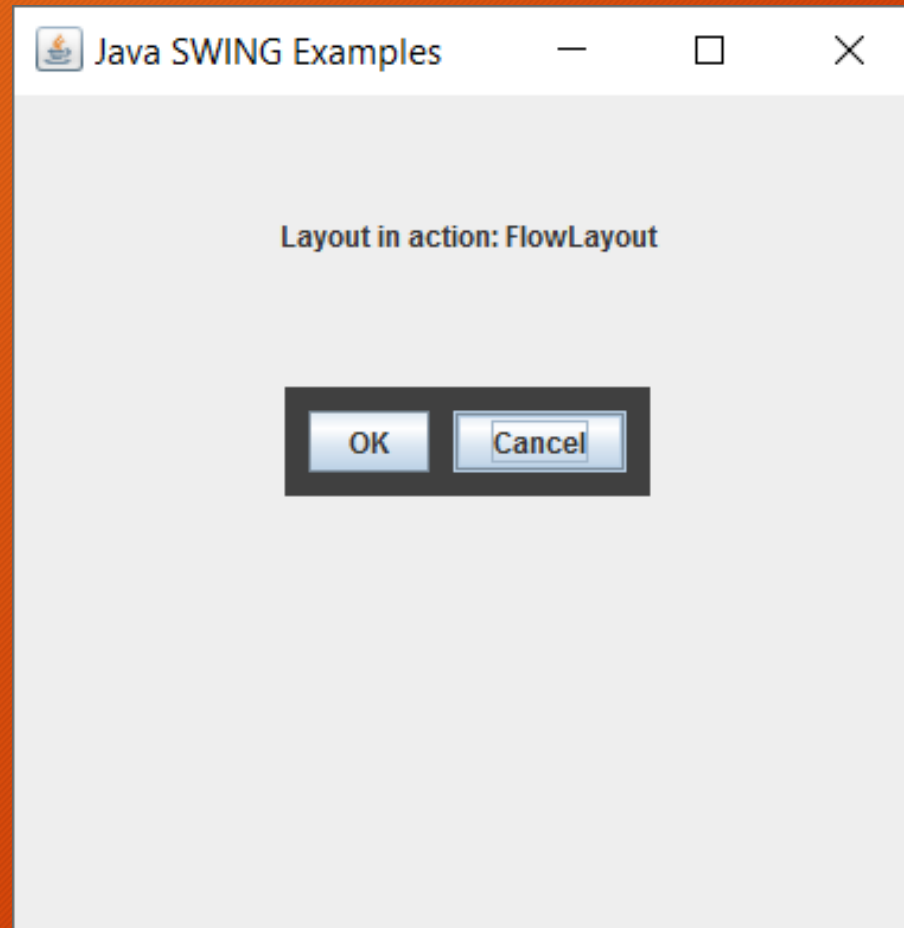


```
private void showBorderLayoutDemo() {  
    headerLabel.setText("Layout in action: BorderLayout");  
  
    JPanel panel = new JPanel();  
    panel.setBackground(Color.darkGray);  
    panel.setSize(300,300);  
    BorderLayout layout = new BorderLayout();  
    layout.setHgap(10);  
    layout.setVgap(10);  
  
    panel.setLayout(layout);  
    panel.add(new JButton("Center"),BorderLayout.CENTER);  
    panel.add(new JButton("Line Start"),BorderLayout.LINE_START);  
    panel.add(new JButton("Line End"),BorderLayout.LINE_END);  
    panel.add(new JButton("East"),BorderLayout.EAST);  
    panel.add(new JButton("West"),BorderLayout.WEST);  
    panel.add(new JButton("North"),BorderLayout.NORTH);  
    panel.add(new JButton("South"),BorderLayout.SOUTH);  
  
    controlPanel.add(panel);  
    mainFrame.setVisible(true);  
}
```



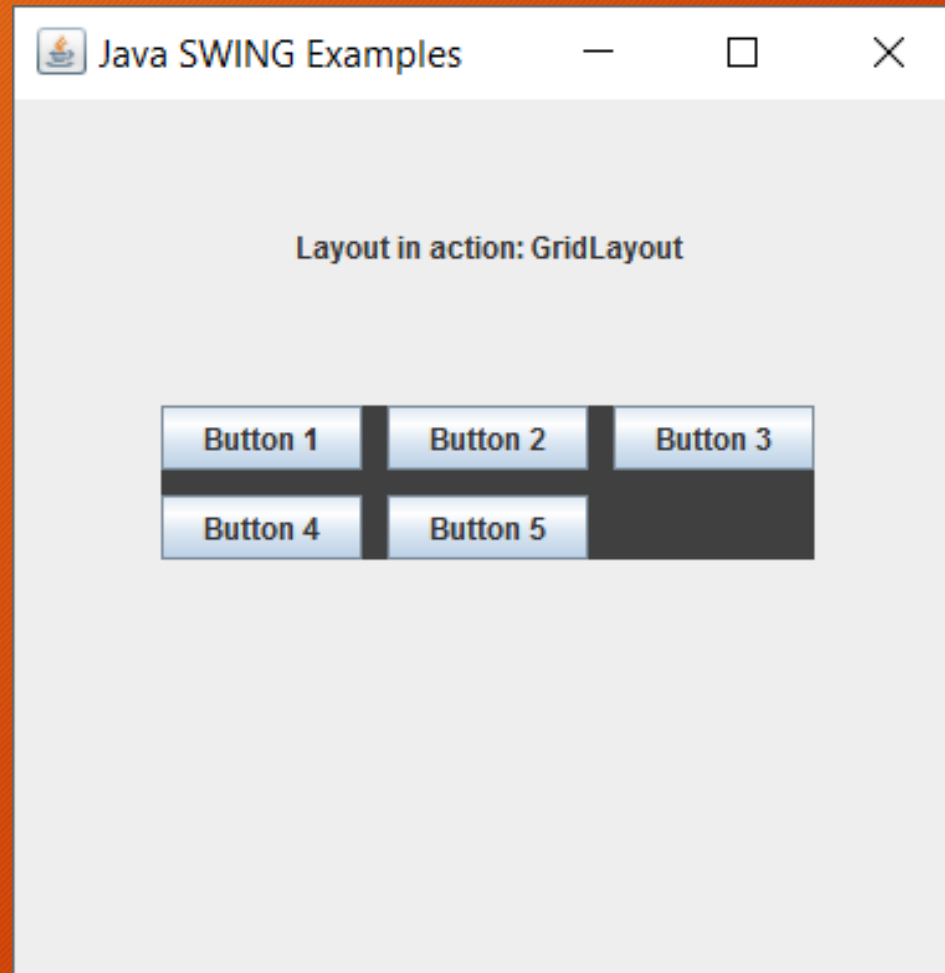


```
private void showCardLayoutDemo() {  
    headerLabel.setText("Layout in action: CardLayout");  
    final JPanel panel = new JPanel();  
    panel.setBackground(Color.CYAN);  
    panel.setSize(300,300);  
    CardLayout layout = new CardLayout();  
    layout.setHgap(10);  
    layout.setVgap(10);  
    panel.setLayout(layout);  
    JPanel buttonPanel = new JPanel(new FlowLayout());  
    buttonPanel.add(new JButton("OK"));  
    buttonPanel.add(new JButton("Cancel"));  
    JPanel textBoxPanel = new JPanel(new FlowLayout());  
    textBoxPanel.add(new JLabel("Name:"));  
    textBoxPanel.add(new JTextField(20));  
    panel.add("Button", buttonPanel);  
    panel.add("Text", textBoxPanel);  
    final DefaultComboBoxModel panelName = new DefaultComboBoxModel();  
    panelName.addElement("Button");  
    panelName.addElement("Text");  
    final JComboBox listCombo = new JComboBox(panelName);  
    listCombo.setSelectedIndex(0);  
    JScrollPane listComboScrollPane = new JScrollPane(listCombo);  
    JButton showButton = new JButton("Show");  
}
```

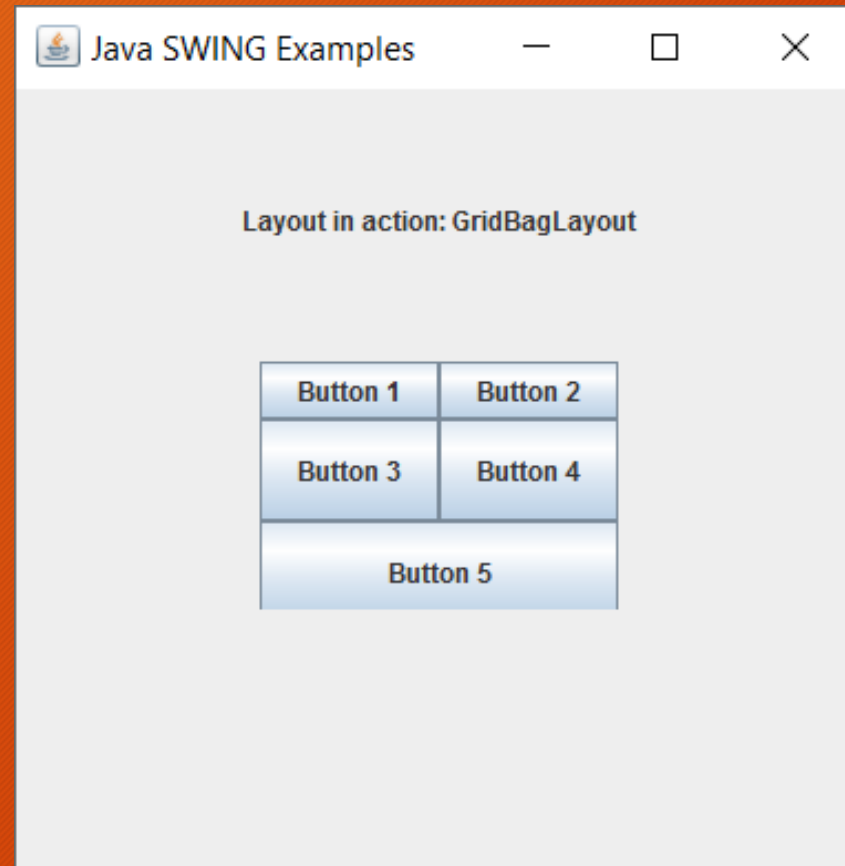


```
private void showFlowLayoutDemo() {  
    headerLabel.setText("Layout in action: FlowLayout");  
  
    JPanel panel = new JPanel();  
    panel.setBackground(Color.darkGray);  
    panel.setSize(200,200);  
    FlowLayout layout = new FlowLayout();  
    layout.setHgap(10);  
    layout.setVgap(10);  
  
    panel.setLayout(layout);  
    panel.add(new JButton("OK"));  
    panel.add(new JButton("Cancel"));  
    controlPanel.add(panel);  
    mainFrame.setVisible(true);  
}  
}
```





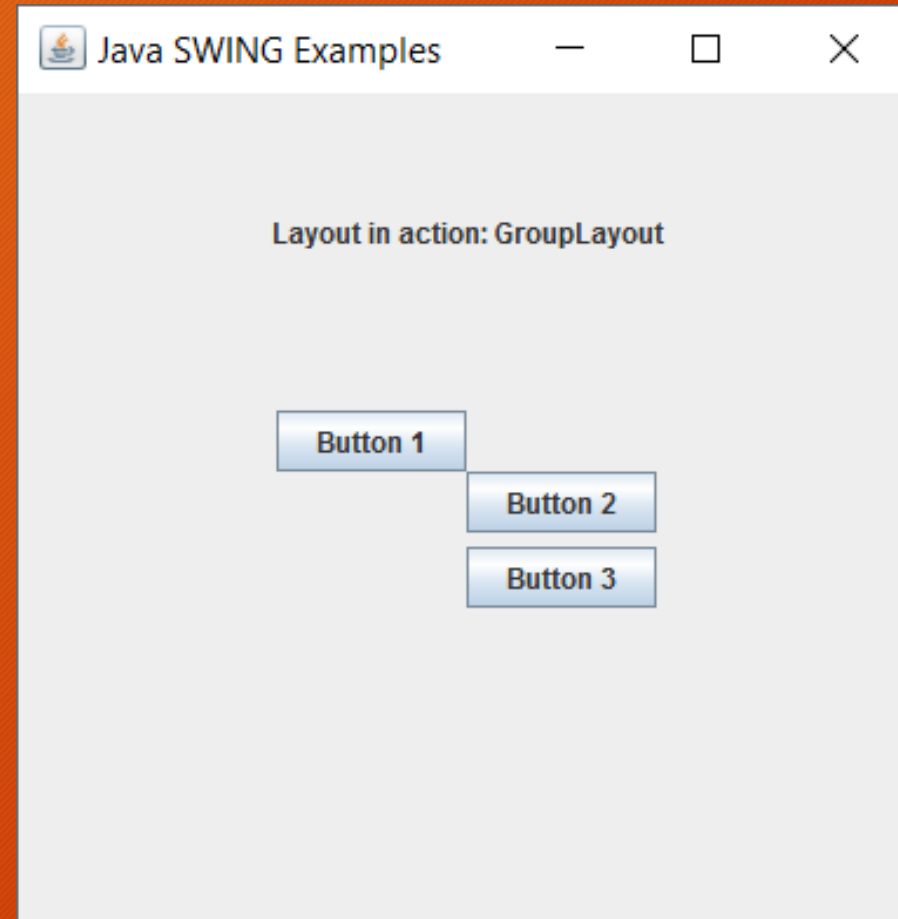
```
private void showGridLayoutDemo() {  
    headerLabel.setText("Layout in action: GridLayout");  
  
    JPanel panel = new JPanel();  
    panel.setBackground(Color.darkGray);  
    panel.setSize(300,300);  
    GridLayout layout = new GridLayout(0,3);  
    layout.setHgap(10);  
    layout.setVgap(10);  
  
    panel.setLayout(layout);  
    panel.add(new JButton("Button 1"));  
    panel.add(new JButton("Button 2"));  
    panel.add(new JButton("Button 3"));  
    panel.add(new JButton("Button 4"));  
    panel.add(new JButton("Button 5"));  
    controlPanel.add(panel);  
    mainFrame.setVisible(true);  
}
```





```
private void showGridBagLayoutDemo() {  
    headerLabel.setText("Layout in action: GridBagLayout");  
    JPanel panel = new JPanel();  
    panel.setBackground(Color.darkGray);  
    panel.setSize(300,300);  
    GridBagLayout layout = new GridBagLayout();  
    panel.setLayout(layout);  
    GridBagConstraints gbc = new GridBagConstraints();  
    gbc.fill = GridBagConstraints.HORIZONTAL;  
    gbc.gridx = 0;  
    gbc.gridy = 0;  
    panel.add(new JButton("Button 1"), gbc);  
    gbc.gridx = 1;  
    gbc.gridy = 0;  
    panel.add(new JButton("Button 2"), gbc);  
    gbc.fill = GridBagConstraints.HORIZONTAL;  
    gbc.ipady = 20;  
    gbc.gridx = 0;  
    gbc.gridy = 1;  
}
```

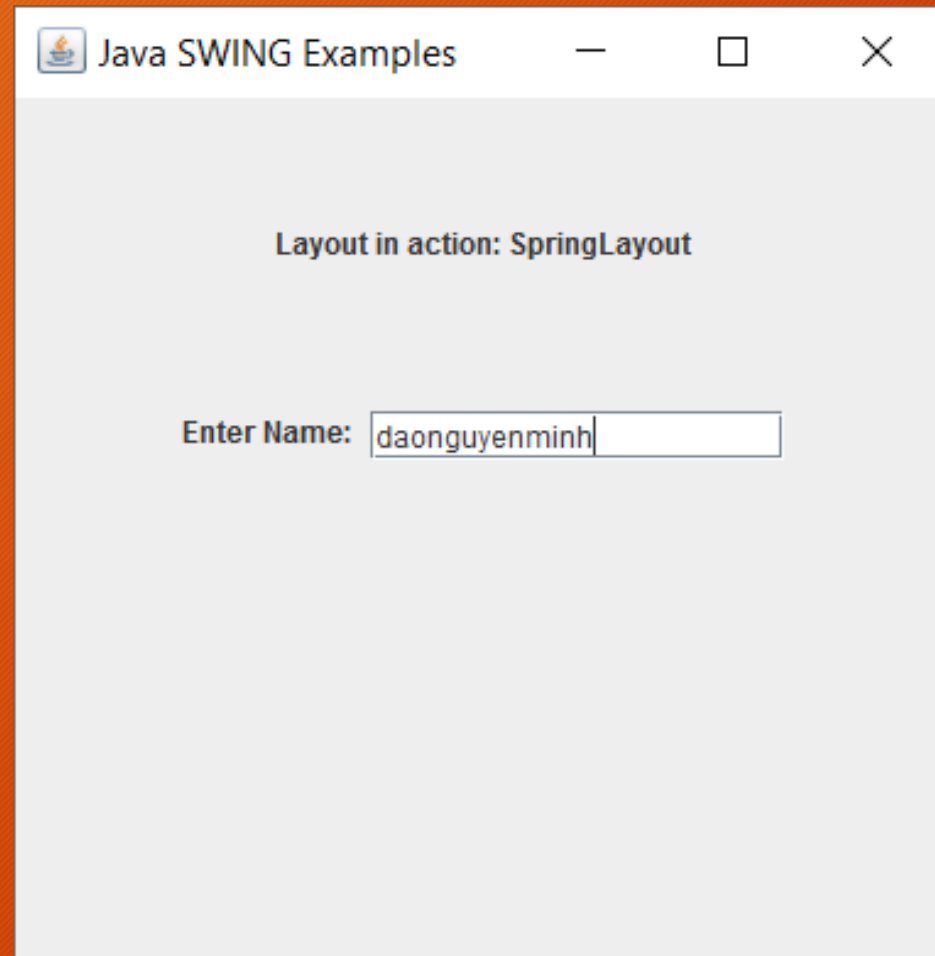
```
panel.add(new JButton("Button 3"), gbc);  
gbc.gridx = 1;  
gbc.gridy = 1;  
panel.add(new JButton("Button 4"), gbc);  
gbc.gridx = 0;  
gbc.gridy = 2;  
gbc.fill = GridBagConstraints.HORIZONTAL;  
gbc.gridwidth = 2;  
panel.add(new JButton("Button 5"), gbc);  
controlPanel.add(panel);  
mainFrame.setVisible(true);  
}
```





```
private void showGroupLayoutDemo () {  
    headerLabel.setText("Layout in action: GroupLayout");  
    JPanel panel = new JPanel();  
  
    // panel.setBackground(Color.darkGray);  
    panel.setSize(200,200);  
    GroupLayout layout = new GroupLayout(panel);  
    layout.setAutoCreateGaps(true);  
    layout.setAutoCreateContainerGaps(true);  
  
    JButton btn1 = new JButton("Button 1");  
    JButton btn2 = new JButton("Button 2");  
    JButton btn3 = new JButton("Button 3");  
  
    layout.setHorizontalGroup(layout.createSequentialGroup()  
        .addComponent(btn1)  
        .addGroup(layout.createSequentialGroup()  
            .addGroup(layout.createParallelGroup(  
                GroupLayout.Alignment.LEADING)  
                .addComponent(btn2)  
                .addComponent(btn3)))  
        );  
}
```

```
        layout.setVerticalGroup(layout.createSequentialGroup()  
            .addComponent(btn1)  
            .addComponent(btn2)  
            .addComponent(btn3) );  
  
        panel.setLayout(layout);  
        controlPanel.add(panel);  
        mainFrame.setVisible(true);  
    }
```





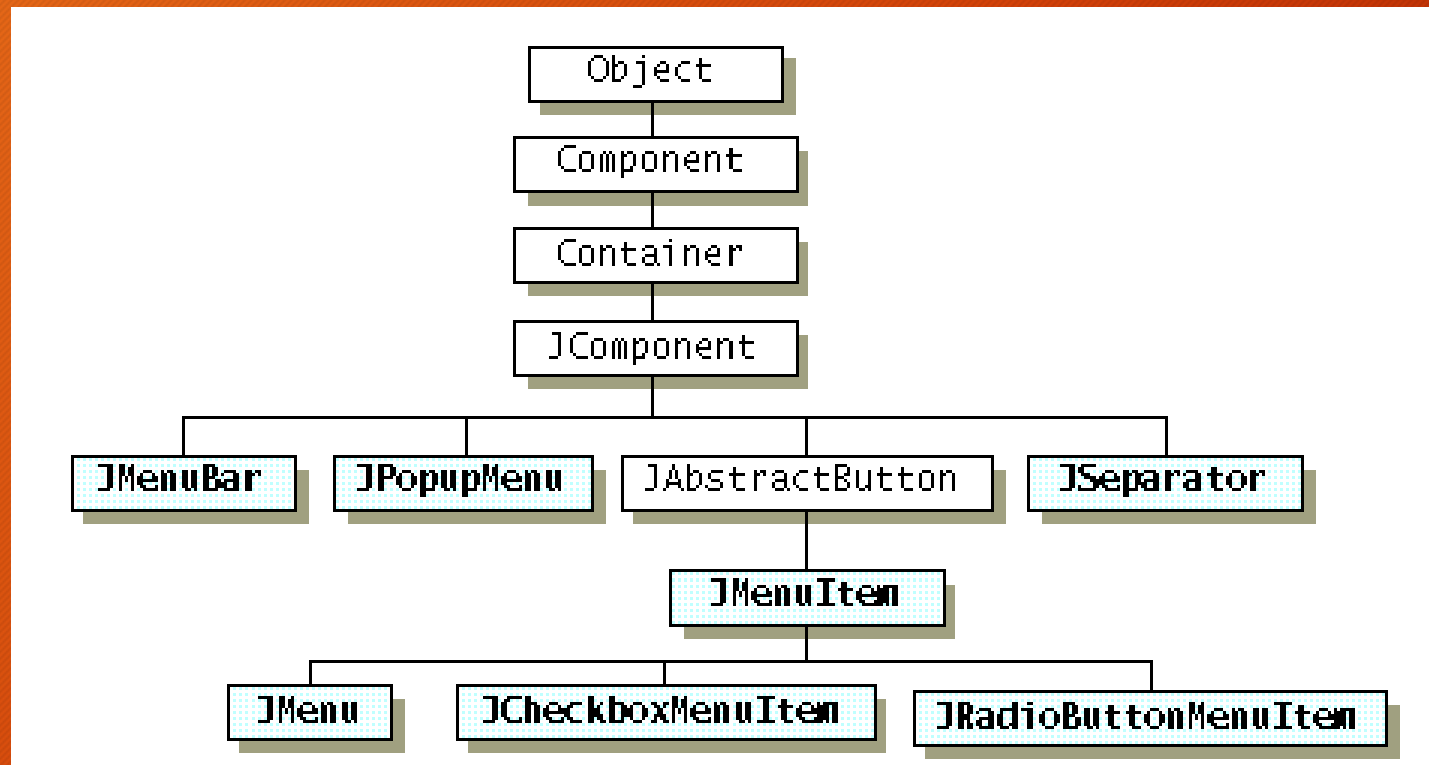
```
private void showSpringLayoutDemo() {  
    headerLabel.setText("Layout in action: SpringLayout");  
    SpringLayout layout = new SpringLayout();  
  
    JPanel panel = new JPanel();  
    panel.setLayout(layout);  
    JLabel label = new JLabel("Enter Name: ");  
    JTextField textField = new JTextField("", 15);  
    panel.add(label);  
    panel.add(textField);  
  
    layout.putConstraint(SpringLayout.WEST, label, 5, SpringLayout.WEST, controlPanel);  
    layout.putConstraint(SpringLayout.NORTH, label, 5, SpringLayout.NORTH, controlPanel);  
    layout.putConstraint(SpringLayout.WEST, textField, 5, SpringLayout.EAST, label);  
    layout.putConstraint(SpringLayout.NORTH, textField, 5, SpringLayout.NORTH,  
        controlPanel);  
  
    layout.putConstraint(SpringLayout.EAST, panel, 5, SpringLayout.EAST, textField);  
    layout.putConstraint(SpringLayout.SOUTH, panel, 5, SpringLayout.SOUTH, textField);  
    controlPanel.add(panel);  
    mainFrame.setVisible(true);  
}
```

# SWING - Menu Classes

88

# Menu Hierarchy

89





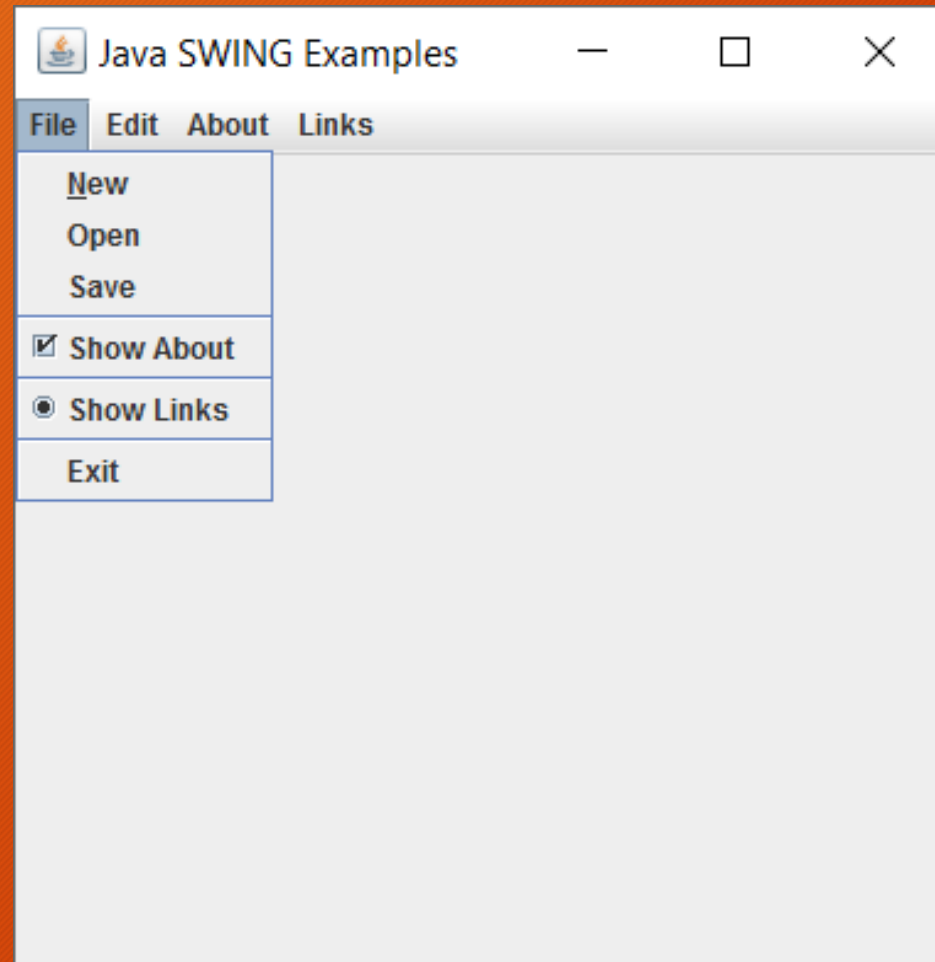
# Menu Controls

90

Sr.No.	Class & Description
1	<b>JMenuBar</b> <a href="#">↗</a> The JMenuBar object is associated with the top-level window.
2	<b>JMenuItem</b> <a href="#">↗</a> The items in the menu must belong to the JMenuItem or any of its subclass.
3	<b>JMenu</b> <a href="#">↗</a> The JMenu object is a pull-down menu component which is displayed from the menu bar.

4	<b>JCheckboxMenuItem</b> <a href="#">↗</a> JCheckboxMenuItem is the subclass of JMenuItem.
5	<b>JRadioButtonMenuItem</b> <a href="#">↗</a> JRadioButtonMenuItem is the subclass of JMenuItem.
6	<b>JPopupMenu</b> <a href="#">↗</a> JPopupMenu can be dynamically popped up at a specified position within a component.







```
//create a menu bar
final JMenuBar menuBar = new JMenuBar();
//create menus
JMenu fileMenu = new JMenu("File");
JMenu editMenu = new JMenu("Edit");
final JMenu aboutMenu = new JMenu("About");
final JMenu linkMenu = new JMenu("Links");
//create menu items
JMenuItem newItem = new JMenuItem("New");
newItem.setMnemonic(KeyEvent.VK_N);
newItem.setActionCommand("New");
JMenuItem openMenuItem = new JMenuItem("Open");
openMenuItem.setActionCommand("Open");
JMenuItem saveMenuItem = new JMenuItem("Save");
saveMenuItem.setActionCommand("Save");
JMenuItem exitMenuItem = new JMenuItem("Exit");
exitMenuItem.setActionCommand("Exit");
JMenuItem cutMenuItem = new JMenuItem("Cut");
cutMenuItem.setActionCommand("Cut");
JMenuItem copyMenuItem = new JMenuItem("Copy");
copyMenuItem.setActionCommand("Copy");
JMenuItem pasteMenuItem = new JMenuItem("Paste");
pasteMenuItem.setActionCommand("Paste");
```

```
MenuItemListener menuItemListener = new MenuItemListener();

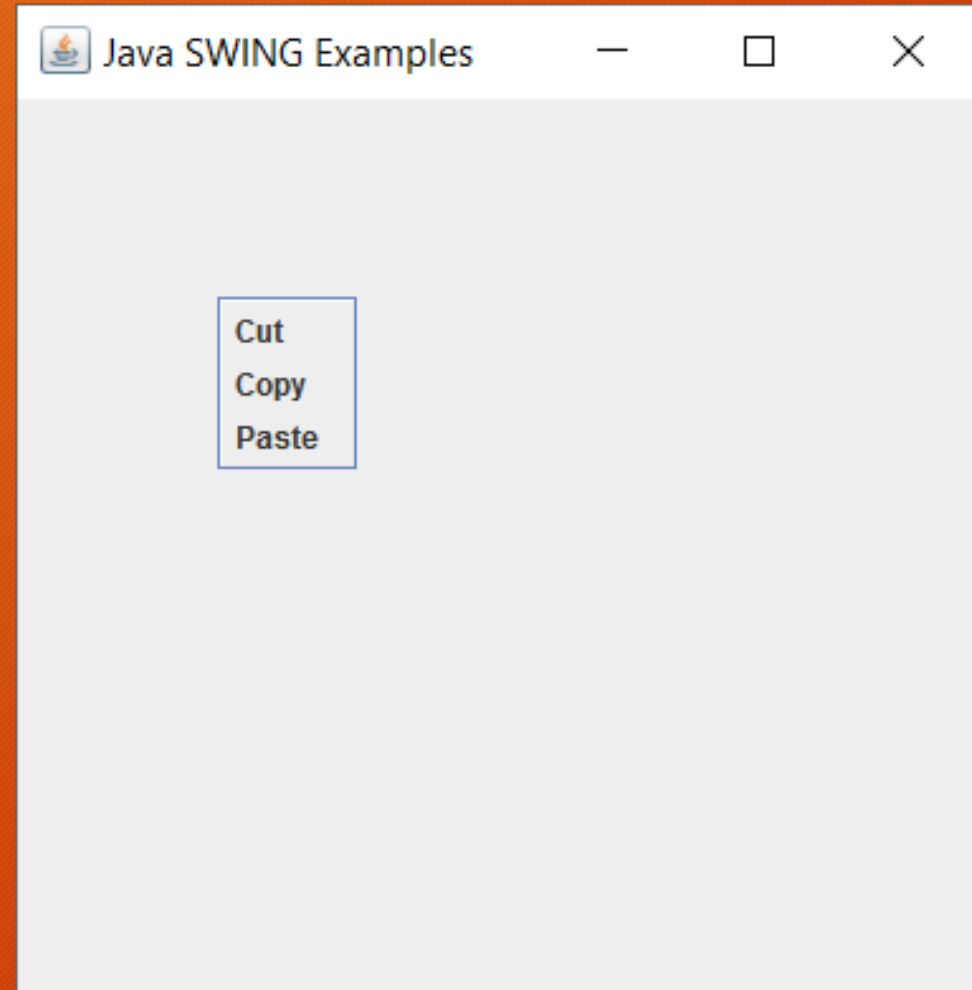
newMenuItem.addActionListener(menuItemListener);
openMenuItem.addActionListener(menuItemListener);
saveMenuItem.addActionListener(menuItemListener);
exitMenuItem.addActionListener(menuItemListener);
cutMenuItem.addActionListener(menuItemListener);
copyMenuItem.addActionListener(menuItemListener);
pasteMenuItem.addActionListener(menuItemListener);

final JCheckBoxMenuItem showWindowMenu = new JCheckBoxMenuItem("Show About", true);
showWindowMenu.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {

        if(showWindowMenu.getState()) {
            menuBar.add(aboutMenu);
        } else {
            menuBar.remove(aboutMenu);
        }
    }
});
final JRadioButtonMenuItem showLinksMenu = new JRadioButtonMenuItem(
    "Show Links", true);
```

```
showLinksMenu.addItemListener(new ItemListener() {  
    @Override  
    public void itemStateChanged(ItemEvent e) {  
  
        if (menuBar.getMenu(3) != null) {  
            menuBar.remove(linkMenu);  
            mainFrame.repaint();  
        } else {  
            menuBar.add(linkMenu);  
            mainFrame.repaint();  
        }  
    }  
});  
  
//add menu items to menus  
fileMenu.add(newMenuItem);  
fileMenu.add(openMenuItem);  
fileMenu.add(saveMenuItem);  
fileMenu.addSeparator();
```





```
private void showPopupMenuDemo() {  
    final JPopupMenu editMenu = new JPopupMenu("Edit");  
    JMenuItem cutMenuItem = new JMenuItem("Cut");  
    cutMenuItem.setActionCommand("Cut");  
    JMenuItem copyMenuItem = new JMenuItem("Copy");  
    copyMenuItem.setActionCommand("Copy");  
    JMenuItem pasteMenuItem = new JMenuItem("Paste");  
    pasteMenuItem.setActionCommand("Paste");  
    MenuItemListener menuItemListener = new MenuItemListener();  
    cutMenuItem.addActionListener(menuItemListener);  
    copyMenuItem.addActionListener(menuItemListener);  
    pasteMenuItem.addActionListener(menuItemListener);  
    editMenu.add(cutMenuItem);  
    editMenu.add(copyMenuItem);  
    editMenu.add(pasteMenuItem);  
  
    mainFrame.addMouseListener(new MouseAdapter() {  
        @Override  
        public void mouseClicked(MouseEvent e) {  
            editMenu.show(mainFrame, e.getX(), e.getY());  
        }  
    });  
    mainFrame.add(editMenu);  
    mainFrame.setVisible(true);  
}
```

# SWING - Containers




98



- Containers are an integral part of SWING GUI components. A container provides a space where a component can be located.
- A Container in AWT is a component itself and it provides the capability to add a component to itself. Following are certain noticable points to be considered.
  - Sub classes of Container are called as Container. For example, **JPanel**, **JFrame** and **JWindow**.
  - Container can add only a Component to itself.
  - A default layout is present in each container which can be overridden using **setLayout** method.

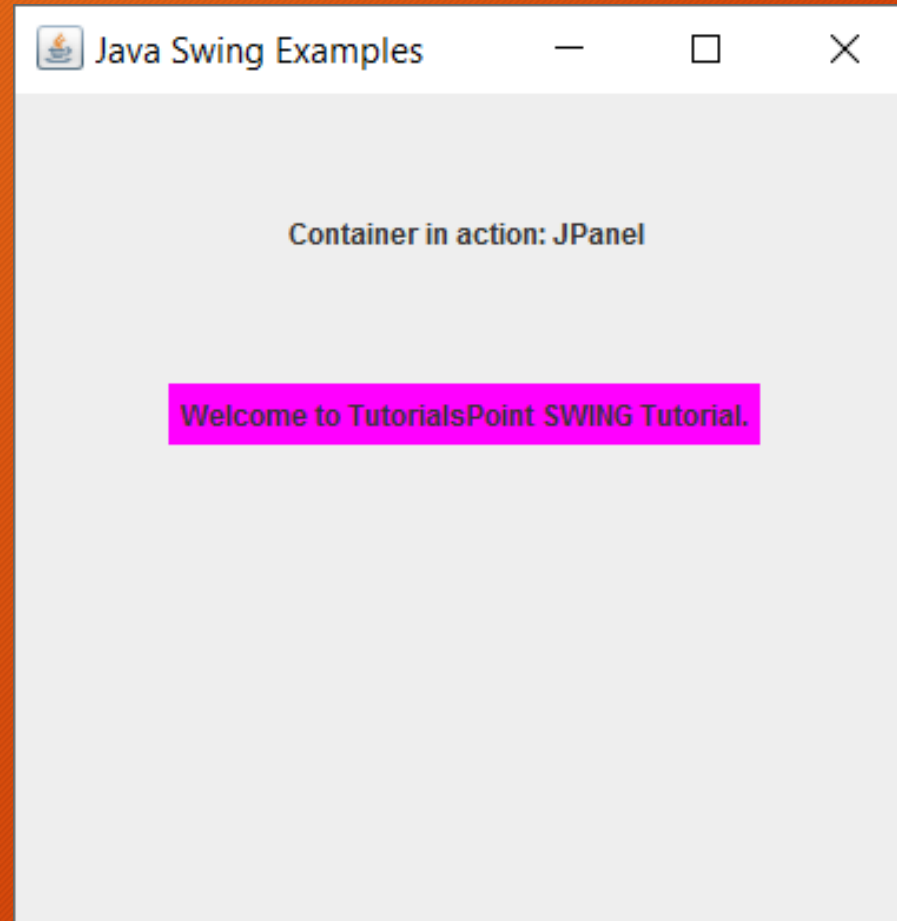
# SWING Containers

100

Sr.No.	Container & Description
1	<b>Panel</b>  JPanel is the simplest container. It provides space in which any other component can be placed, including other panels.
2	<b>Frame</b>  A JFrame is a top-level window with a title and a border.
3	<b>Window</b>  A JWindow object is a top-level window with no borders and no menubar.

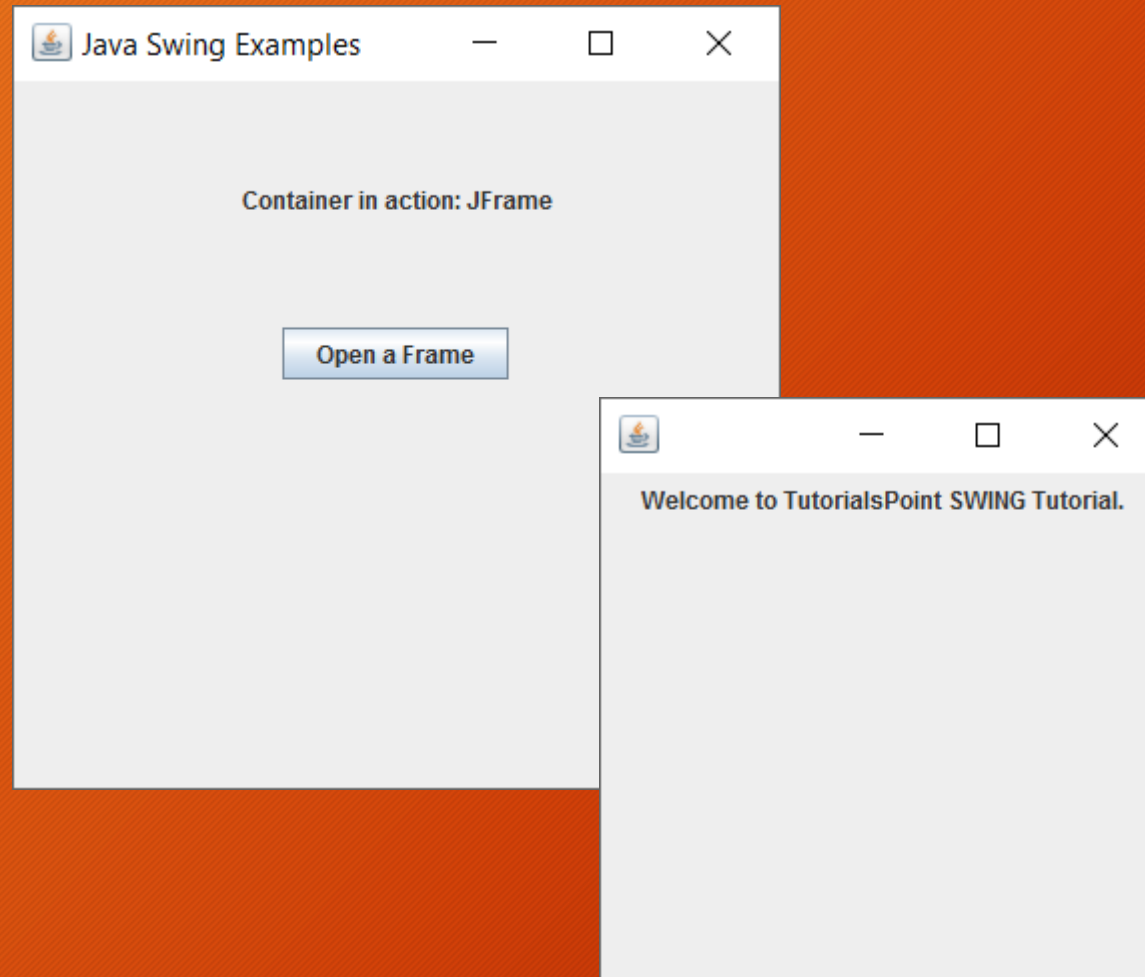






```
headerLabel = new JLabel("", JLabel.CENTER);
statusLabel = new JLabel("", JLabel.CENTER);
statusLabel.setSize(350,100);
msglabel = new JLabel("Welcome to Tutorialspoint SWING Tutorial.", JLabel.CENTER);
controlPanel = new JPanel();
controlPanel.setLayout(new FlowLayout());

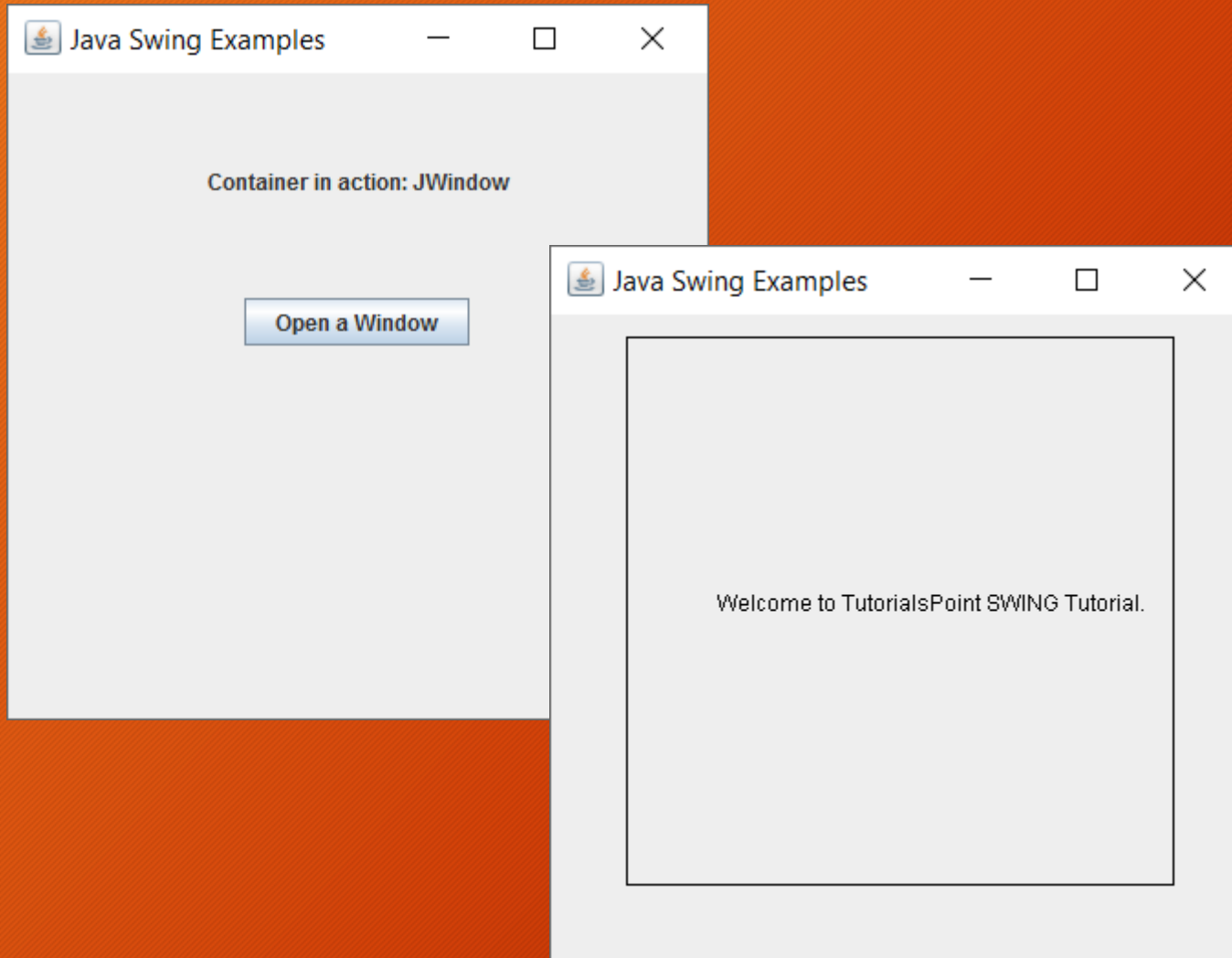
mainFrame.add(headerLabel);
mainFrame.add(controlPanel);
mainFrame.add(statusLabel);
mainFrame.setVisible(true);
```





```
private void showJFrameDemo() {
    headerLabel.setText("Container in action: JFrame");
    final JFrame frame = new JFrame();
    frame.setSize(300, 300);
    frame.setLayout(new FlowLayout());
    frame.add(msgLabel);

    frame.addWindowListener(new WindowAdapter() {
        @Override
        public void windowClosing(WindowEvent windowEvent) {
            frame.dispose();
        }
    });
    JButton okButton = new JButton("Open a Frame");
    okButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            statusLabel.setText("A Frame shown to the user.");
            frame.setVisible(true);
        }
    });
    controlPanel.add(okButton);
    mainFrame.setVisible(true);
}
```



```
class MessageWindow extends JWindow{
    private String message;
    public MessageWindow(JFrame parent, String message) {
        super(parent);
        this.message = message;
        setSize(300, 300);
        setLocationRelativeTo(parent);
    }
    @Override
    public void paint(Graphics g) {
        super.paint(g);
        g.drawRect(0,0,getSize().width - 1,getSize().height - 1);
        g.drawString(message,50,150);
    }
}
```



```
private void showJWindowDemo() {  
    headerLabel.setText("Container in action: JWindow");  
    final MessageWindow window = new MessageWindow(  
        mainFrame, "Welcome to TutorialsPoint SWING Tutorial.");  
  
    JButton okButton = new JButton("Open a Window");  
    okButton.addActionListener(new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            window.setVisible(true);  
            statusLabel.setText("A Window shown to the user.");  
        }  
    });  
    controlPanel.add(okButton);  
    mainFrame.setVisible(true);  
}
```





