UNIT-1

1. Event handling and AWT

1.1 Introduction to Event Handling

In any interactive environment, the program should be able to respond to actions performed by the user. The action can be mouse click, key press or selection of menu items. The basis of event driven programming is typing events to code that responds to those events.

Java's Abstract Windowing Toolkit (AWT) communicates these actions to the program using events. When user initiates an action, the AWT generates an event and communicates it to event handlers. These event handlers can respond to the events appropriately.

Event handling is fundamental to Java programming because it is used to create event driven programs like:

- Applets
- GUI based windows application
- Web Application

It is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Event handling is at the core of successful AWT programming. Most events to which the applet will respond are generated by the user. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button.

1.2 The Delegation Event Model

In java the delegation event model defines a standard mechanism to generate and process events.The Delegation Event Model is a programming pattern used in Java for handling events in graphical user interfaces (GUIs). In this model, when an event is generated by a GUI component (such as a button press), it is delegated to one or more event listeners registered with that component. These listeners then handle the event by executing the appropriate code.

It provides a uniform and interoperable method for creating and handling events. In this model, a source generates an event and forwards it to one or more listeners. After receiving an event, the listener waits. When the event is received, the listener processes it and sends it back.

The key advantage of the Delegation Event Model is that the application logic is completely separated from the interface logic. In this model, the listener must be connected with a source to receive the event notifications. Thus, the events will only be received by the listeners who wish to receive them. So, this approach is more convenient than the inheritance-based event model. Basically, an Event Model is based on the following three components: Events, Event Sources and Events Listeners

a. **Events:** An event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a GUI. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

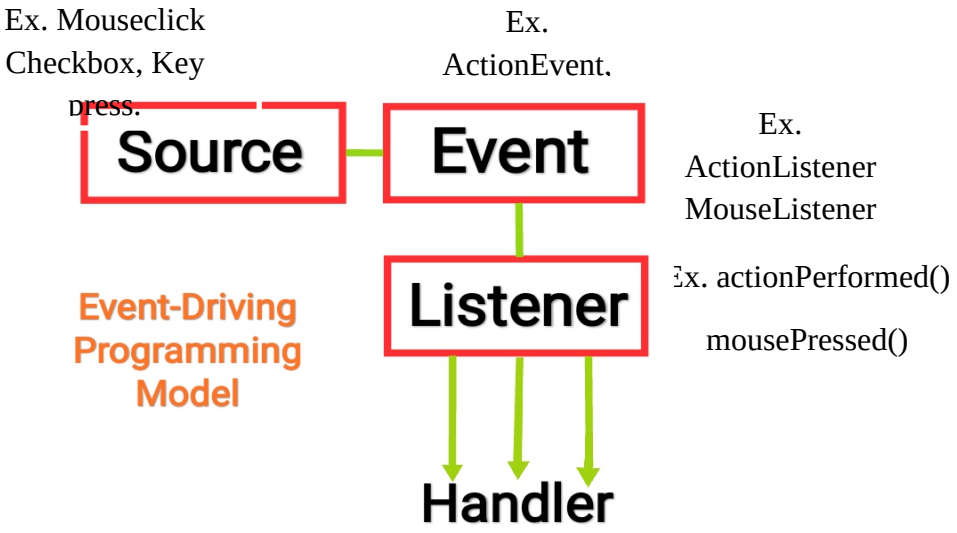


Figure 1.1: - Event Delegation Model

➢ **Types of Events in Java:**

- **Foreground Events**: Those events require the direct interaction of the user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example
  - Clicking on a button
  - Moving the mouse
  - Entering a character through the keyboard
  - Selecting an item from the list
  - Scrolling the page, etc

- **Background Events:** Those events that require the interaction of the end-user are known as background events. Operating system interrupts, hardware or software failure, the timer expires, an operation completion is the example of background events.

b. Event Sources: A source is an object that generates an event. Generally, sources are components like Button, Checkbox, RadioButton etc. 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. Here is the general form:

public void addTypeListener (TypeListener e)

Here, TypeListener is the name of the event, and e is a reference to the event listener.

For e.g., the method that registers a Mouse event listener is called addMouseListener( ). A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeTypeListener(TypeListener e)

Here, TypeListener is the name of the event and 'e' is a reference to the event listener

c. **Event classes and Event Listeners:** A listener is an object that is notified when an event occurs. A listener has two major requirements. First is, listener must be registered with a source to receive notifications and second is, it must implement methods to receive and process these notifications.

1.3 Event Classes

In Java, events and sources are maintained as classes, listeners are maintained as interfaces. Most of them are available in java.awt.event package. Examples of listeners are ActionListener, ItemListener, MouseListener, Key Listener etc.

java.awt.event Package

The java.awt.event package contains many event classes which can be used for event handling. Root class for all the event classes in Java is EventObject which is available in java.util package. Root class for all AWT event classes is AWTEvent which is available in java.awt package and is a sub class of EventObject class.

Some of the frequently used event classes available in **java.awt.event** package are listed below:

Control	Description
AWTEvent	For all AWT events, it is the root event class. The original java.awt.Event class is superseded by this class and its subclasses.
ActionEvent	When a button is pressed or a list item is double clicked, an ActionEvent is created.
InputEvent	All component level input events belong to the InputEvent class, which is the root event class
KeyEvent	On entering the character, the Key event is generated.
MouseEvent	This event indicates a mouse action occurred in a component.
TextEvent	The object of this class represents the text events.
WindowEvent	This class's object depicts how a window's status has changed.
AdjustmentEvent	The adjustment event that Adjustable objects emit is represented by t his class's objec
ComponentEvent	The object of this class represents the change in state of a window.
ContainerEvent	The object of this class represents the change in state of a window.
MouseMotionEvent	The object of this class represents the change in state of a window.
PaintEvent	The object of this class represents the change in state of a window

Table 1: Event Classes

1.4 Event Listener Interfaces

Following is the list of commonly used **event listeners**.

Control	Description
ActionListener	This interface is used for receiving the action events.
ComponentListener	This interface is used for receiving the component events.
ItemListener	This interface is used for receiving the item events.
KeyListener	This interface is used for receiving the key events.
MouseListener	This interface is used for receiving the mouse events.
TextListener	This interface is used for receiving the text events.
WindowListener	This interface is used for receiving the window events.
AdjustmentListener	This interface is used for receiving the adjustment events.
ContainerListener	This interface is used for receiving the container events.
MouseMotionListener	This interface is used for receiving the mouse motion events.
FocusListener	This interface is used for receiving the focus events.

Table 2: Event Listeners Interfaces

All together Event Classes, Event Listeners, Event methods and action it performs.

Source	Action by Event Source	Event Listener	Event Handling Method	Event Class
Button	Clicking on button	ActionListener	actionPerformted()	ActionEvent
Checkbox	Selecting checkbox items	ItemListener	itemStateChanged()	ItemEvent
Choice	Selecting option from choice	ItemListener	itemStateChanged()	ItemEvent
CheckboxGroup	Select checkbox from given option	ItemListener	itemStateChanged()	ItemEvent
List	Selecting option from list	ActionListener	actionPerformed()	ActionEvent
TextField	Enter value in textfield	ActionListener	actionPerformed()	ActionEvent
Scrollbar	Scrolling scrollbar	AdjustmentListener	adjustmentValueChanged()	Adjustment Event

Methods of Component Class:

Method	Description
Public void add(Component c)	Inserts a component
Public void setSize(int width, int height)	Sets the width and height of the component
Public void setLayout(LayoutManager m)	Defines the layout manager for the component
Public void setVisible(boolean v)	Changes the visibility of the component. By default it is false.

1.5 Handling Mouse Event and Keyboard Events

The term "MouseEvent" refers to an event that shows a mouse click occurred within a component. When the mouse pointer is over the visible portion of a component's boundaries, the action is supposed to have taken place in that specific component. A mouse event type is enabled by adding the appropriate mouse-based EventListener to the component (Mouse Listener or Mouse Motion Listener), or by invoking Component.enableEvents (long) with the appropriate mask parameter

(AWTEvent.MOUSE_EVENT_MASK or AWTEvent.MOUSE_MOTION_EVENT_MASK).

In the event that the mouse event type is not enabled on the component, the corresponding mouse events are sent to the first ancestor who has enabled the mouse event type. In the event that a MouseListener is added to a component or that enable Events (AWTEvent.MOUSE_EVENT_MASK) is called, then all of the events defined by the MouseListener are sent to the component.

In the event that neither of these actions is taken, however—that is, if MouseMotionListener has not been added, nor if enableEvents has not been called with AWTEvent.MOUSE_MOTION_EVENT_MASK—then the corresponding mouse events are sent to the first ancestor that has enabled mouse motion events.

Some methods of MouseEvent Class:

Method	Description
public Point getLocationOnScreen()	Gets the absolute x,y position of the MouseEvent.
public int getX()	Gets the x-coordinate of the MouseEvent
public int getY()	Gets the y-coordinate of the MouseEvent

Mouse Listeners

To catch mouse events we import java.awt.event.MouseListener and the class we want to handle events should implement the MouseListener interface. This interfaces requires that we implement the following methods:

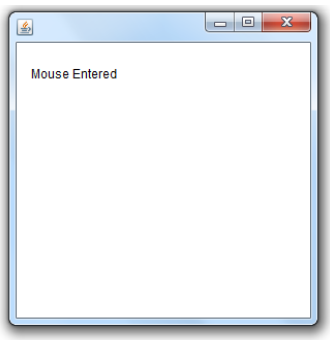
- void mouseClicked(MouseEvent me)
- void mouseEntered(MouseEvent me)
- void mousePressed(MouseEvent me)
- void mouseReleased(MouseEvent me)
- void mouseExited(MouseEvent me)

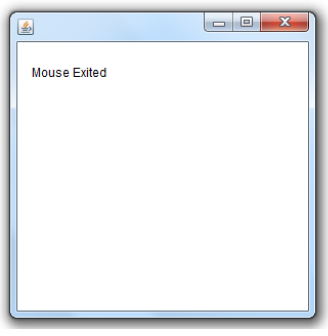
Each method takes a MouseEvent parameter which gives us the X and Y coordinates of the mouse. The component to handle the mouse events calls the addMouseListener(component) method.

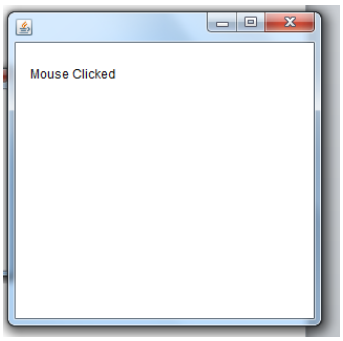
//The following program is an example for MouseEvent.

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener
{
Label l;
```

```
MouseListenerExample(){
addMouseListener(this);
l=new Label();
l.setBounds(20,50,100,20);
add(l);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void mouseClicked(MouseEvent e) {
l.setText("Mouse Clicked");
}
public void mouseEntered(MouseEvent e) {
l.setText("Mouse Entered");
}
public void mouseExited(MouseEvent e) {
l.setText("Mouse Exited");
}
public void mousePressed(MouseEvent e) {
l.setText("Mouse Pressed");
}
public void mouseReleased(MouseEvent e) {
l.setText("Mouse Released");
}
public static void main(String[] args) {
new MouseListenerExample();
} }
```







- **Mouse Motion Listener**

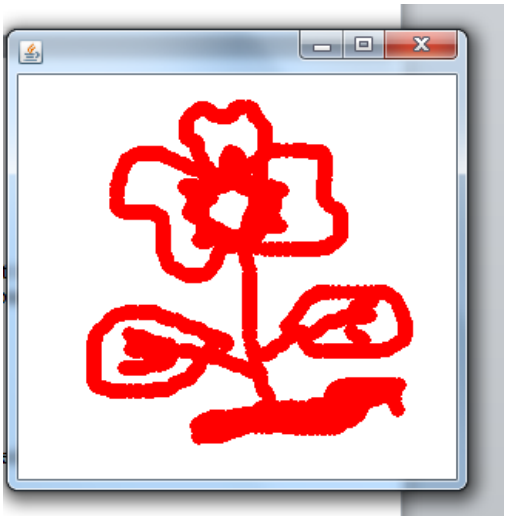
The MouseListener interface is geared around mouse clicks. If you want to capture mouse motion then there is a separate interface, the MouseMotionListener. This is also found in java.awt.event. The MouseMotionListener requires that we implement following methods:

- mouseMoved()
- mouseDragged()

As you can infer, the methods are invoked when the mouse is either moved or dragged. Both take a MouseEvent as an input parameter.

//Example of Mouse Motion Listener

```
import java.awt.*;
import java.awt.event.*;
public class MouseMotionListenerExample extends Frame implements
MouseMotionListener{
   MouseMotionListenerExample(){
      addMouseMotionListener(this);

      setSize(300,300);
      setLayout(null);
      setVisible(true);
   }
public void mouseDragged(MouseEvent e) {
   Graphics g=getGraphics();
   g.setColor(Color.RED);
   g.fillOval(e.getX(),e.getY(),10,10);
}
public void mouseMoved(MouseEvent e) {}

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



- **Keyboard Event : KeyEvent Class**

Keyboard input results in the creation of a KeyEvent. KeyEvent classes and listeners are available in java.awt.event.

There are three different key events that are denoted by integer constants:

- KEY_PRESSED - When a key is pressed, this is event generated.
- KEY_RELEASED - When a key is released, this is event generated.
- KEY_TYPED - When a character is typed on the keyboard, that is a key is pressed and then released, this is event.

Every keystroke does not produces a letter, For instance, shifting does not produce a character.

There are many other integer constants that are defined by KeyEvent.

For example,

VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters.

Methods of KeyEvent class

Method	Description

int getKeyCode()	It is used for getting the integer code associated with a key. It is used for KEY_PRESSED and KEY_RELEASED events. The keycodes are defined as constants in KeyEvent class.
char getKeyChar()	This method is used to get the Unicode character of the key pressed. It works with the KEY_TYPED events

- **Key Listener**

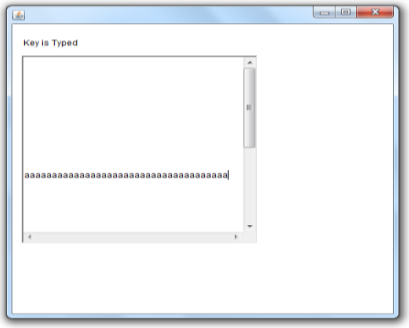
Key events are monitored by Java KeyListener. The java.awt.event package contains objects that implement the KeyListener interface, which serve as their representation. Three methods specified by KeyListener must be carried out by a KeyListener's class. These methods have the following syntax:

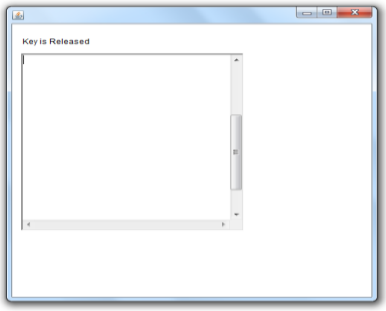
- public abstract void keyPressed (KeyEvent e)
- public abstract void keyReleased (KeyEvent e)
- public abstract void keyTyped (KeyEvent e)

/* Example of Key Event*/

```
import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener {
 Label l;
   TextArea area;
     KeyListenerExample() {
     l = new Label();
        l.setBounds (20, 50, 100, 20);
       area = new TextArea();
       area.setBounds (20, 80, 300, 300);
       area.addKeyListener(this);
       add(l);
      add(area);
       setSize (500, 500);
       setLayout (null);
       setVisible (true);
   }
   public void keyPressed (KeyEvent e) {
      l.setText ("Key is Pressed");
   }
   public void keyReleased (KeyEvent e) {
      l.setText ("Key is Released");
   }
   public void keyTyped (KeyEvent e) {
      l.setText ("Key is Typed");
   }
   public static void main(String[] args) {
      new KeyListenerExample();
```

}}





1.6 Adapter Classes

When you want to receive and process only a portion of the events handled by a specific event listener interface, Java offers a special feature called an adapter class that can make creating event handlers easier in specific scenarios. An adapter class offers an empty implementation of all the methods in an event listener interface.

MouseListener	MouseAdapter
void mouseClicked(MouseEvent me) void mouseEntered(MouseEvent me) void mouseExited(MouseEvent me) void mousePressed(MouseEvent me) void mouseReleased(MouseEvent me)	void mouseClicked(MouseEvent me){ } void mouseEntered(MouseEvent me) { } void mouseExited(MouseEvent me) { } void mousePressed(MouseEvent me) { } void mouseReleased(MouseEvent me) { }

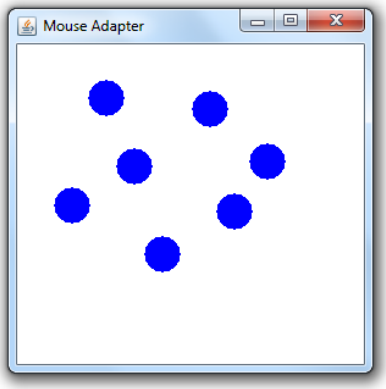
Table: Commonly used Listener Interfaces implemented by Adapter Classes

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

//Example of Adapter Class using Mouse Adapter Class

```
import java.awt.*;
import java.awt.event.*;
 public class MouseAdapterExample extends MouseAdapter {
    Frame f;
    MouseAdapterExample() {
        f = new Frame ("Mouse Adapter");
        f.addMouseListener(this);
        f.setSize (300, 300);
        f.setLayout (null);
```

```
        f.setVisible (true);
    }

    public void mouseClicked (MouseEvent e) {
        Graphics g = f.getGraphics();
        g.setColor (Color.BLUE);
        g.fillOval (e.getX(), e.getY(), 30, 30);
    }
public static void main(String[] args) {
    new MouseAdapterExample();
}}
```



1.7 AWT Class Hierarchy

Java AWT (Abstract Window Toolkit) is an API to develop Graphical User Interface (GUI) or windows-based applications in Java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

The AWT tutorial will help the user to understand Java GUI programming in simple and easy steps.

The hierarchy of Java AWT classes are given below.

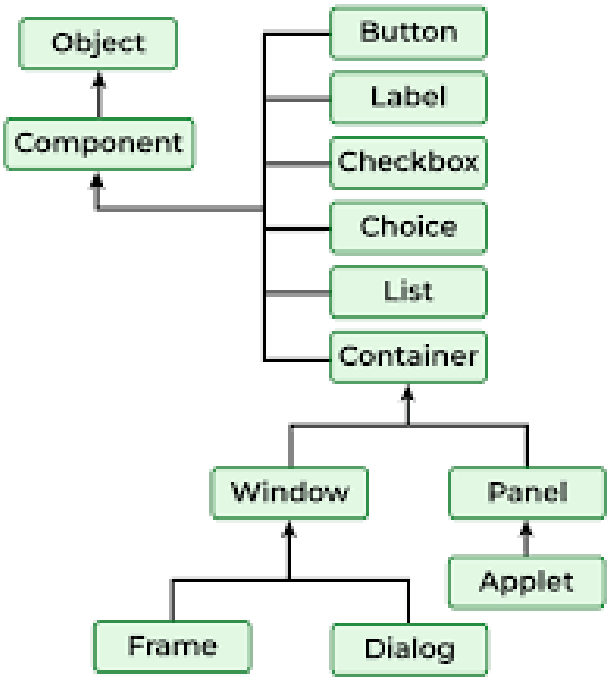


Fig: Java AWT Hierarchy

- **Components**: AWT provides various components such as buttons, labels, text fields, checkboxes, etc used for creating GUI elements for Java Applications.
- **Containers**: AWT provides containers like panels, frames, and dialogues to organize and group components in the Application.
- **Layout Managers:** Layout Managers are responsible for arranging data in the containers some of the layout managers are BorderLayout, FlowLayout, etc.
- **Event Handling:** AWT allows the user to handle the events like mouse clicks, key presses, etc. using event listeners and adapters.
- **Graphics and Drawing:** It is the feature of AWT that helps to draw shapes, insert images and write text in the components of a Java Application.

1.8 Types of Containers in Java AWT

There are four types of containers in Java AWT:

- **Window**: Window is a top-level container that represents a graphical window or dialog box. The Window class extends the Container class, which means it can contain other components, such as buttons, labels, and text fields.
- **Panel**: Panel is a container class in Java. It is a lightweight container that can be used for grouping other components together within a window or a frame.
- **Frame**: The Frame is the container that contains the title bar and border and can have menu bars.
- **Dialog**: A dialog box is a temporary window an application creates to retrieve user input**.**

1.9 AWT Controls (Components):

The AWT supports the following types of controls:

Label,	Button,	TextField,
TextArea,	Choice,	List,
CheckBox,	CheckBoxGroup,	Scrollbar,

These controls are subclasses of Component. To use these controls, before its use you need to add it in the in container using ***Component add(Component c)*** method and to remove it you can use ***void remove(Component c)*** method of component class.
Here, c is an instance of the control that you want to add or remove.
ou can remove all controls by calling **removeAll( )**.

a. Labels:

A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

- Label( )
- Label(String *str*)
- Label(String *str*, int *align*)

The first constructor creates a blank label. The second one creates a label that contains the string specified by *str*. And the third one creates a label that contains the string specified by *str* using the alignment specified by *align*. The value of *align* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

b. Button

A button is basically a control component with a label that generates an event when pushed. The Button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.
When we press a button and release it, AWT sends an instance of ActionEvent to that button by calling processEvent on the button. The processEvent method of the button receives the all the events, then it passes an action event by calling its own method processActionEvent. This method passes the action event on to action listeners that are interested in the action events generated by the button.

To perform an action on a button being pressed and released, the ActionListener interface needs to be implemented. The registered new listener can receive events from the button by calling addActionListener method of the button. The Java application can use the button's action command as a messaging protocol

Syantx:

public class Button extends Component implements Accessible

Button defines the following constructors:

Button( )

Button (String text)

Button Class Methods:

Method	Description
void setText (String text)	It sets the string message on the button
String getText()	It fetches the String message on the button.
void setLabel (String label)	It sets the label of button with the specified string.
String getLabel()	It fetches the label of the button.

```
//Example of Button AWT Control
import java.awt.*;
import java.awt.event.*;
class ButtonFrame extends Frame implements ActionListener
{
Button b1,b2,b3;
MyFrame()
{
b1=new Button("Red");
b2=new Button("Green");
b3=new Button("Blue");
add(b1); add(b2); add(b3);
b1.addActionListener(this);
```

```
b2.addActionListener(this);
b3.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{
if(ae.getActionCommand()=="Red")
setBackground(new Color(255,0,0));
if(ae.getActionCommand()=="Green")
setBackground(new Color(0,255,0));
if(ae.getActionCommand()=="Blue")
setBackground(new Color(0,0,255));
}
}
class ButtonExample
{
public static void main(String arg[])
{
ButtonFrame f=new ButtonFrame();
f.setSize(400,400);
f.setTitle("Event Handling Programs...");
f.setVisible(true);
f.setLayout(new FlowLayout());
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent we)
{
System.exit(0);
}
});
}
}
```

c. Check Boxes:

A CheckBox is a AWT control that is used to create checkboxes which allows to select particular or all options from given list of options. There is a label associated with each check box that describes what option the box represents.

Constructors of Checkbox class are:

- Checkbox( ) throws HeadlessException
- Checkbox(String *str*) throws HeadlessException
- Checkbox(String *str*, boolean *on*) throws HeadlessException
- Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*) throws HeadlessException
- Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*) throws HeadlessException

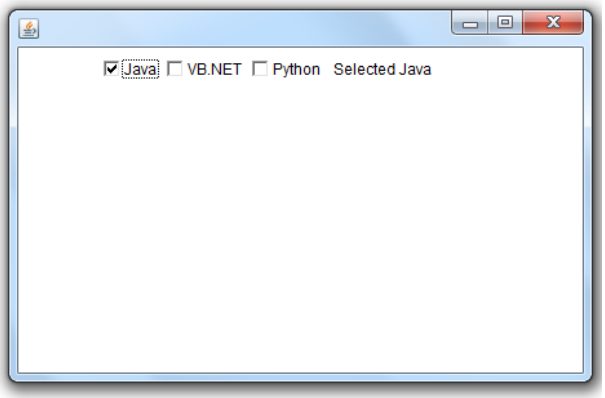
The first form of constructor creates a check box whose label is initially blank. The default state of the check box is unchecked. All other constructors are having 'str' as string and Boolean value either true or false to indicate checkbox is checked or unchecked.

Methods of Checkbox class:

Method	Description
boolean getState( )	To retrieve the current state of a check box
void setState(boolean *on*)	to set the state of a check box
String getLabel( )	returns the label associated with check box
void setLabel(String *str*)	to set the label

```
/* Example of Checkbox
import java.awt.*;
import java.awt.event.*;

public class CheckboxExample extends Frame implements ItemListener{
      // Create checkboxes
Checkbox ch1, ch2,ch3;
 Label lbl1,lbl2,lbl3;
       CheckboxExample(){
      ch1 = new Checkbox("Java");
      ch2 = new Checkbox("VB.NET");
      ch3 = new Checkbox("Python");
       lbl1=new Label();
       lbl2=new Label();
       lbl3=new Label();
      add(ch1);
      add(ch2);
      add(ch3);
      add(lbl1);
      add(lbl2);
      add(lbl3);

      setVisible(true);
      setLayout(new FlowLayout());
      setSize(300,300);

      ch1.addItemListener(this);
      ch2.addItemListener(this);
      ch3.addItemListener(this);
   }

 public void itemStateChanged(ItemEvent ie)
   {
    if(ch1.getState()==true)
          lbl1.setText("Selected Java");
  if(ch2.getState()==true)
```

```
        lbl2.setText("Selected VB.NET");
  if(ch3.getState()==true)
         lbl3.setText("Selected Python");
  }
   public static void main(String[] args) {
      new CheckboxExample();
   }
}
```



d. CheckboxGroup:

A collection of mutually exclusive check boxes can be made, where only one check box in the group can be selected at a time. One button may be selected at a time from these check boxes, which are also known as **radio buttons.**

Defining the group to which the check boxes will belong and then specifying that group during check box construction are the initial steps in creating a set of mutually exclusive check boxes.

The default constructor is the only one defined, and it generates an empty group.

CheckboxGroup()

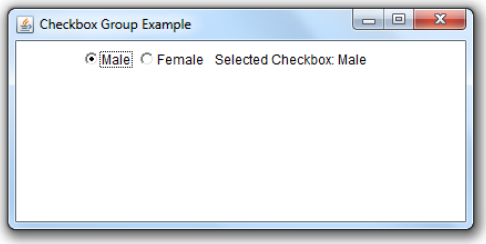
Methods:

Method	Description
Checkbox getSelectedCheckbox( )	which check box in a group is currently selected
void setSelectedCheckbox(Checkbox *c)*	*'c'* is the check box that you want to be selected. The previously selected check box will be turned off

/* Example of CheckboxGroup

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

public class CheckboxGroupDemo extends Frame implements ItemListener {
   private CheckboxGroup checkboxGroup;
   private Checkbox ch1, ch2;
   Label lbl;
```

```
    public CheckboxGroupDemo() {
        checkboxGroup = new CheckboxGroup();
        lbl= new Label();

        ch1 = new Checkbox("Male", checkboxGroup, false);
        ch2 = new Checkbox("Female", checkboxGroup, false);

        ch1.addItemListener(this);
        ch2.addItemListener(this);

        add(ch1);  add(ch2);     add(lbl);

        setTitle("Checkbox Group Example");
        setSize(300, 200);
        setLayout(new FlowLayout());
        setVisible(true);
    }
    public void itemStateChanged(ItemEvent e) {
        Checkbox selectedCheckbox = checkboxGroup.getSelectedCheckbox();
        lbl.setText("Selected Checkbox: " + selectedCheckbox.getLabel());
    }
    public static void main(String[] args) {
        new CheckboxGroupDemo();
    }
}
```



e. TextField

The TextField class is used to create a textfield control that enables editing and entry of a single line of text by the user. An ActionEvent is created in a TextField whenever the enter key is hit. Implementing the ActionListener interface is necessary to handle such an event.

Constructors of TextField

Constructor	Description
public TextField)	Creates a TextField..
public TextField(String text)	Creates a TextField with a specified default text.
public TextField(int width)	Creates a TextField with a specified width.
public TextField(String text, int width)	Creates a TextField with a specified default text and width.

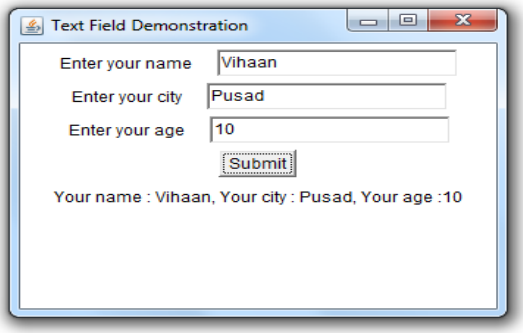
Some methods of TextField class

Methods	Description
public void setText(String text)	Sets a String message on the TextField.
public String getText()	Gets a String message of TextField.
public void setEditable(boolean b)	Sets a *Sets a TextField to editable or uneditable.*
public void setFont(Font f)	Sets a font type to the TextField
void setForeground(Color c)	Sets a foreground color, i.e. color of text in TextField.

/* Example of TextField*/

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

public class TextFieldExample implements ActionListener
{
Button button;
Label label1, label2, label3, label4;
TextField field1, field2, field3;
Frame f;

TextFieldExample()
{
f = new Frame("Handling TextField Event");
f.setSize(340,260);

label1= new Label("Enter your name");
label2= new Label("Enter your city");
label3= new Label("Enter your age");

field1 = new TextField(20);
field2 = new TextField(20);
field3 = new TextField(20);

button = new Button("Submit");

f.setLayout(new FlowLayout());

f.add(label1);
f.add(field1);
f.add(label2);
f.add(field2);
f.add(label3);
f.add(field3);
f.add(button);

button.addActionListener(this); /
```

```
f.setVisible(true);
}

public void actionPerformed(ActionEvent ae)
{
if(ae.getActionCommand().equals("Submit"))
{
label4= new Label("", Label.CENTER);
label4.setText("Your name : " + field1.getText()+ ", Your city : " + field2.getText()+ ", Your
age :" +

field3.getText());
f.add(label4);
f.setVisible(true);
}
}

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



f. TextArea

The user can enter text in multiple lines by using the TextArea class to build a multi-line text area.

Constructors of TextArea

Constructor	Description
public TextArea()	Creates a new TextArea..
public TextArea(String text)	Creates a new TextArea with specified text.
public TextArea(int rows, int columns)	Creates a new TextArea with specified number of rows and columns.
public TextArea(String text, int rows, int columns)	Creates a new TextArea with a text and a number of rows and columns.

Methods of TextArea class

Methods	Description
public void setText(String text)	Sets a String message on the TextArea.
public String getText()	Gets a String message of TextArea.
public void append(String text)	Appends the text to the TextArea.
public int getRows()	Gets the total number of rows in TextArea.
public int getColumns()	Gets the total number of columns in TextArea.

```
import java.awt.*;
public class TextAreaExample
{
Frame f;
TextArea ta1, ta2;

public TextAreaExample()
{
f= new Frame("TextArea");
ta1 = new TextArea("B.C.A.", 10,20);
ta2 = new TextArea("SGBAU, ", 10,20);
ta1.append("SEM IV");
ta2.append("Amravati");

f.add(ta1);
f.add(ta2);

f.setLayout(new FlowLayout());
f.setSize(500,400);
f.setVisible(true);
}

public static void main(String... ar)
{
new TextAreaExample();} }
```

g. List class

A list with many values can be created using the List class, enabling the user to choose any value. Implementing the ItemListener interface handles the ItemEvent that is generated when a value is chosen from the List.

Constructors of List

Constructor	Description
public List()	Creates a scrolling list.
public List(int *rows*)	Creates a List with a number of *rows*.

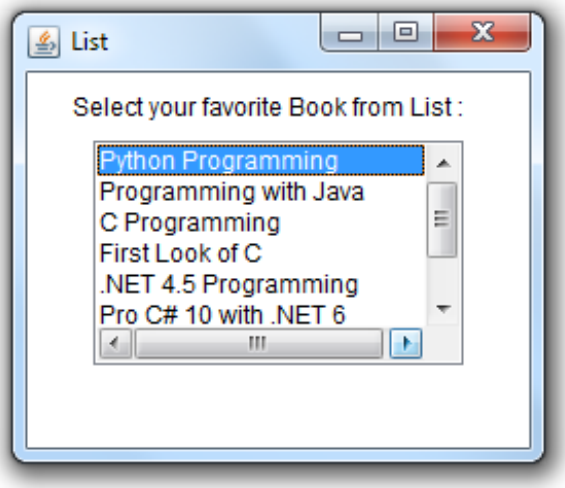
public List(int rows, boolean *mode*)	Creates a List with a number of *rows*, allowing us to select multiple values from list if *mode* is true.

Methods of List class

Methods	**Description**
public void add(String *item*)	Adds the *item* to the end of List.
public void add(String *item*, int *index*)	Adds the *item* at a specific *index* in the list.
public void addActionListener(ActionListener al)	Adds a specific ActionListener to listen an Action event generated when item is selected from this list.
public void addItemListener(ItemListener al)	Adds a specific ItemListener to listen an Item event generated when item is selected from this list.
public String getItem(int *index*)	Gets an *item* from a specific *index* in the list.
public String getSelectedItem()	Gets the selected item in the list.
public String[] getSelectedItems()	Gets the selected item in the list.
public int getRows()	Gets the total number of rows in the list.

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

public class ListExample
{
String [] seasons;
Frame f;
List list;
Label label1;

public ListExample()
{
f= new Frame("List");
list= new List(7);
label1 = new Label("Select your favorite Book from List :");

list.add("Python Programming");
list.add("Programming with Java");
list.add("C Programming");
list.add("First Look of C");
list.add(".NET 4.5 Programming");
list.add("Pro C# 10 with .NET 6");
list.add("Advanced Java Programming");

f.add(label1);
```

```
f.add(list);

f.setLayout(new FlowLayout());
f.setSize(260,220);
f.setVisible(true);
}

public static void main(String... ar)
{
new ListExample();
}
}
```



h. Scrollbar

Scrollbar class is used to create a **horizontal and vertical Scrollbar**. A Scrollbar can be added to a top-level container like Frame or a component like Panel. Scrollbar class is another component in the AWT package which extends the **Component** class.

Constructors of Scrollbar

Constructor	Description
public Scrollbar(int orientation, int *value* int *extent, int min, int max)*	Creates a Scrollbar with the specified *orientation, value, extent, minimum, and maximum*

A few methods of Scrollbar class

Methods	Description
public addAdjustmentListener(AdjustmentListener al)	Adds an AdjustmentListener.
public int getValue()	Gets the Scrollbar's current position value.
public int setValue(int value)	Sets the Scrollbar's current position value.

i. Dialog

Using the Dialog class, a top-level container Dialog window with a number of components can be created.

There are two kinds of Dialog windows –

Modal Dialog window

All user input is focused toward an active modal dialog window, and until the model dialog is closed, all other areas of the application are unavailable.

Modeless Dialog window

While a modeless dialog window is open, input can be forwarded to the other areas of the application without having to close this modeless dialog window.

Constructors of Dialog

Constructor	Description
public Dialog()	Creates a modeless Dialog window without a Frame owner or title..
public Dialog(Dialog owner, String title)	Creates a modeless Dialog window with a Frame owner and a title.
public Dialog(Dialog owner, String title, boolean modal)	Creates a Dialog window with a Frame owner, title and let's you define modality of Dialog window.

j. Menu, MenuItems and MenuBar class

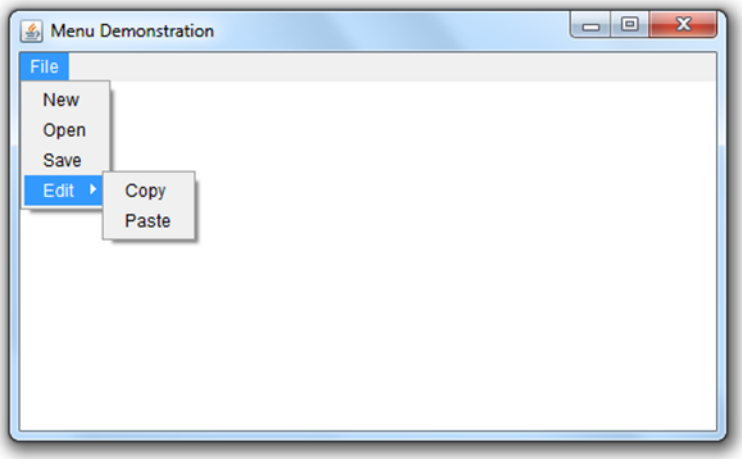
A menu is a Java component that lets programmers arrange elements inside a container in Java applications. You can have more than one menu item on your menu bar, which is where menus are put. These menu items allow users to interact with a program using a graphical user interface (GUI), much like they do in desktop applications and websites.

AWT provides by the following classes:

- **MenuBar :** To create MenuBar in Java application.
- **Menu :** To create Menus in Java application.
- **MenuItem :** To create Menu items in Menu

Constructor of the MenuBar class are:

- MenuBar(): creates a new empty menubar.
- MenuBar(Menu… m): creates a new menubar with the given set of menu.



1.10 Layout Manager

Using Layout Manager all components are automatically positioned within the container. If we don’t specify any layout, the components are also positioned by the default layout

manager. Java provides us with various layouts to position the controls. The properties like size shape and arrangement varies from one layout manager to other layout manager. When the size of 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 application window.

Following the different Layout Manager:

1. FlowLayout
2. BoarderLayout
3. GridLayout
4. CardLayout
5. GridBagLayout

The layout manager is set by the setLayout( ) method. If no call to setLayout( ) is made, then the default layout manager is used.
The setLayout( ) method has the following general form:

- void setLayout(LayoutManager layout)

1. FlowLayout:

It is the default layout manager. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom.
Constructors:

- FlowLayout( )
- FlowLayout(int align)
- FlowLayout(int align, int horz, int vert)

The first form of constructor creates the default layout, which centers components and leaves five pixels of space between each component.
The second form lets you specify how each line is aligned using constants:
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING

2. BorderLayout

For top-level windows, a standard layout style is implemented via the BorderLayout class. Its four borders are made up of one huge region in the center and four narrow components with predetermined widths. The terms north, south, east, and west designate the four sides. The center is the region in the middle.

The constructors defined by BorderLayout are:

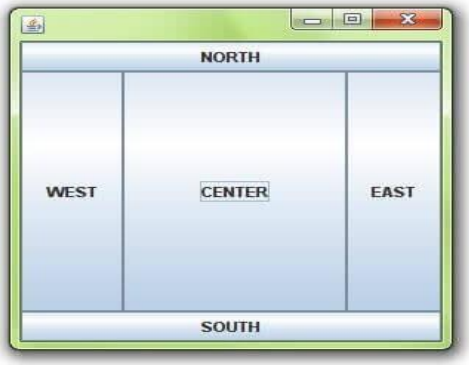
- BorderLayout( )

- BorderLayout(int horz, int vert)

The first form of constructor creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER
BorderLayout.SOUTH
BorderLayout.EAST
BorderLayout.WEST
BorderLayout.NORTH

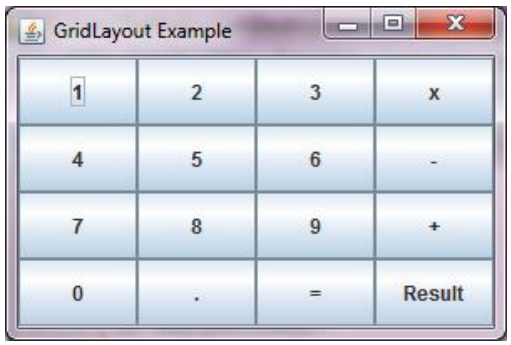


3. GridLayout

GridLayout manager is used to position the components in the manner of *rows and columns*.

The constructors supported by GridLayout are shown here:

- GridLayout( )
- GridLayout(int numRows, int numColumns)
- GridLayout(int numRows, int numColumns, int horz, int vert)



4. CardLayout Manager

CardLayout manager is used to position the components in a manner of *deck of cards, with only the card at the top of deck is visible at a time*. The CardLayout class is unique among the other layout managers in that it stores several
different layouts

CardLayout provides these two constructors:

- CardLayout( )
- CardLayout(int horz, int vert)

The first form of constructor creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

After you have created a deck, your program activates a card by calling one of the following methods defined by CardLayout:

- void first(Container deck)
- void last(Container deck)
- void next(Container deck)
- void previous(Container deck)
- void show(Container deck, String cardName)

5. GridBagLayout

The components are arranged in rows and columns using the GridBagLayout manager, with a variable number of columns in each row. Every component added with GridBagLayout may differ from the others in terms of size and placement.

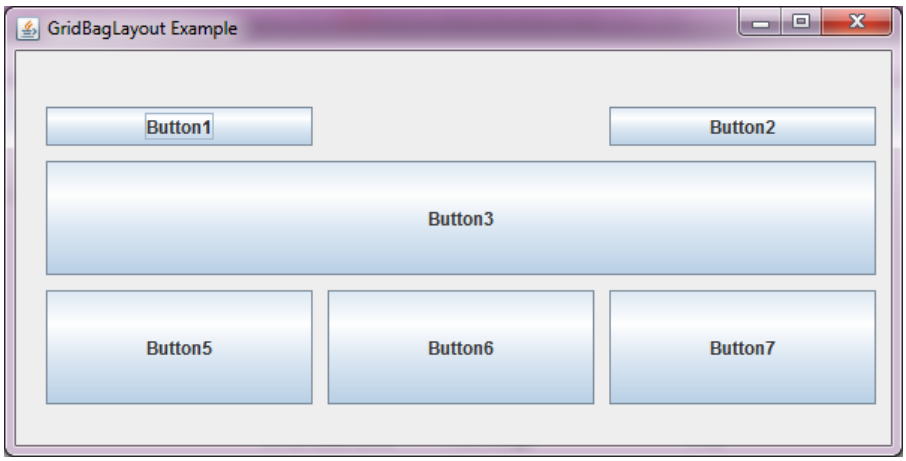
Each component's position dimensions, alignment, height, and width are set using the GridBagLayout constraints—constraints are objects of the GridBagConstraints class—and the fields they give.

GridBagLayout defines only one constructor, which is shown here:

- GridBagLayout( )

A few GridLayout methods

Methods	Description
public void setConstraints(Component comp, GridBagConstraints cons)	This method sets the constraints *cons* to a Component, *comp*
public GridBagConstraints getConstraints(Component comp)	This method gets the constraints applied to a Component, *comp*



EXERCISE:

Q. Multiple Choice Questions

1. Which of these packages contains all the classes and methods required for even handling in Java?
   a) java.applet b) java.awt
   c) java.event d) java.awt.event

2. Which of these methods are used to register a keyboard event listener?
   a) KeyListener() b) addKistener()
   c) addKeyListener() d) eventKeyboardListener()

3. Which of the following is the highest class in the event-delegation model?
   a) Java.util.EventListner b)java.util.ObjectEvent
   b) java.awt.AwtEvent d)java.awt.event.AwtEvent

4. Which of these are constants defined in WindowEvent class?
   a) WINDOW_ACTIVATED b) WINDOW_CLOSED
   c) WINDOW _DEICONIFIED d) All of the mentioned

5. _____is the superclass of all Adapter classes.
   a) Applet b) ComponentEvent
   c) Event d) InputEvent

Q. Answer in one sentence

1. What is Event Listener?
2. What is Event source?
3. What is Adapter Class
4. What is Button?
5. Which is the default Layout Manager, if you don't specify?

Q. Long Answer Questions

1. Explain Event Delegation Model.
2. Explain Keyboard Event handling mechanism.
3. Explain TextField and TextArea in AWT.
4. Describe class hierarchy of AWT Controls.
5. What is Layout Manager? Explain any one of them with example.