Unit-5

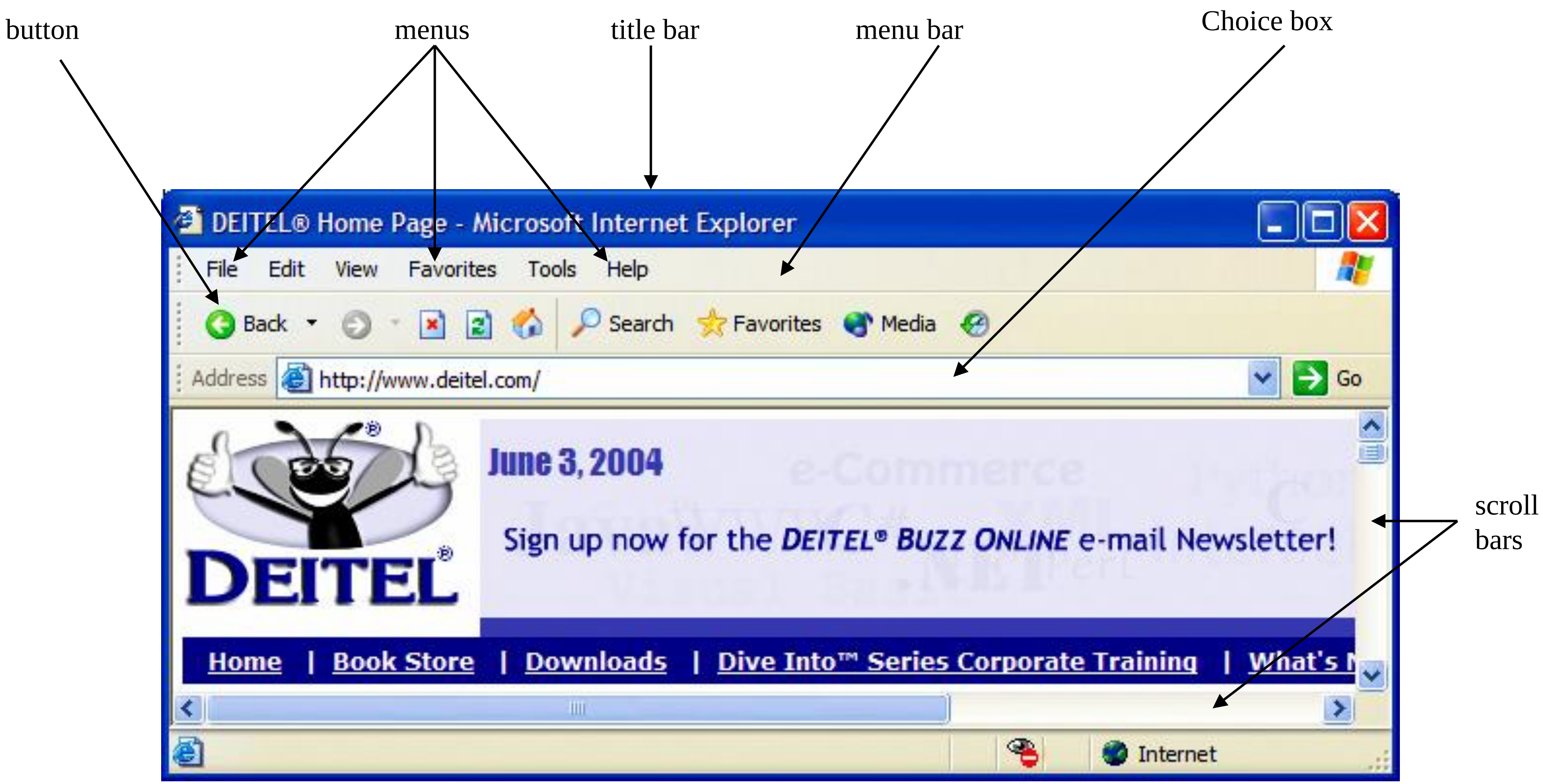
- **Graphical user interface (GUI)**
  - Presents a user-friendly mechanism for interacting with an application.
  - Built from GUI components.
- **Command Line interface (or)**

  Command user interface(CUI)
  - In a command Line interface the commands are entered from the keyboard.
  - It is not user-friendly.
  - Difficult to remember commands.

- Most modern applications use a GUI (pronounced “gooey”):
- **G**raphical: Not just text or characters but windows, menus, buttons, ..
- **U**ser: Person using the program
- **I**nterface: Way to interact with the program

GUI elements/components include:

- Window : Portion of screen that looks as a window.
- Menu : List of alternatives offered to user
- Button : Looks like a button that can be pressed.
- Text fields: The user can write something in etc.



Sign up now for the *DEITEL® BUZZ ONLINE* e-mail Newsletter!

Internet Explorer ***window*** with GUI components.

Abstract Window Toolkit (AWT)

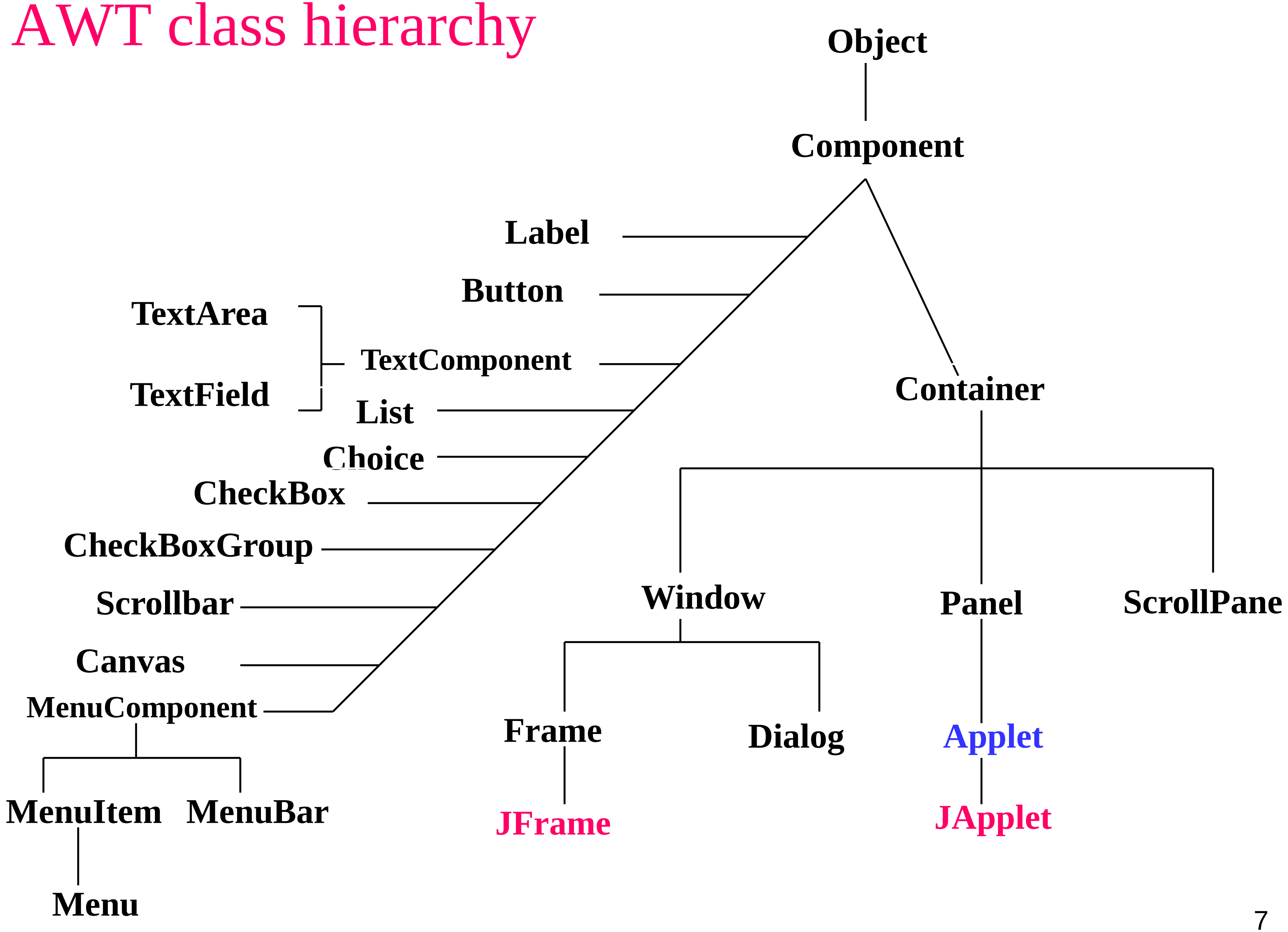
- The AWT contains several classes that allow you to create and manage windows/GUI (Graphical User Interface).

- The main purpose of the AWT is to support ***applet*** windows.

- It can also be used to create stand-alone GUI applications.

- import java.awt.*;

AWT class hierarchy



Introduction to Swings

- **Swing** is a GUI widget toolkit for **Java**.

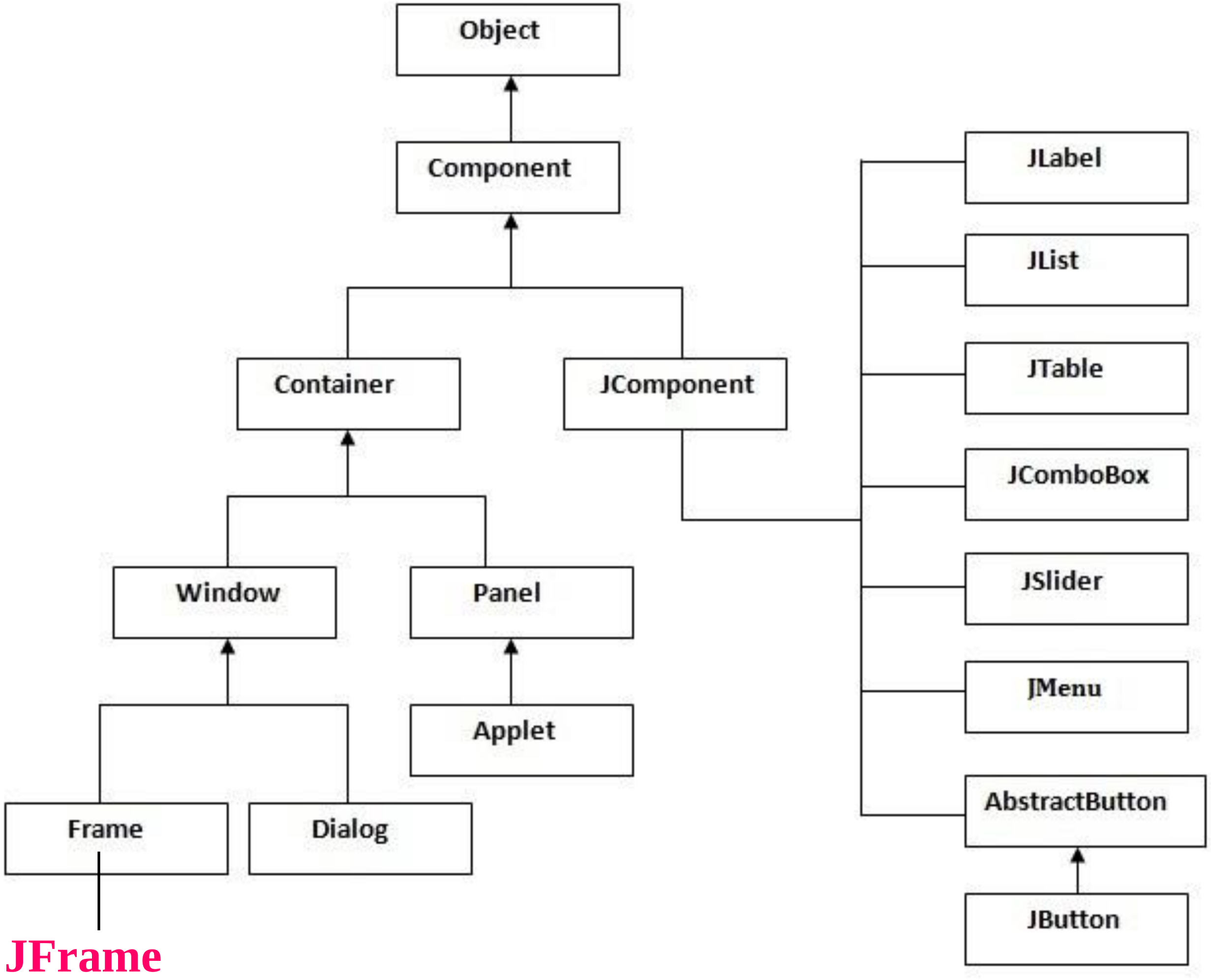
- **Swing** is a collection of libraries that contains primitive *widgets* or *controls* used for designing *Graphical User Interfaces* (GUIs).

- It is part of Oracle's **Java Foundation Classes** (JFC) - an API for providing a graphical user interface (GUI) for **Java** programs

- import javax.swing.*;

- Commonly used classes in javax.swing package: JButton, JTextBox, JTextArea, JPanel, JFrame, JMenu, JLabel, …

  Hierarchy of Java Swing classes



Component:

- This is an **abstract** super class of all AWT classes.
- This class defines some basic methods that will describe about presentation of a component.

- Methods defined in class ***Component*** are:
  - setLocation(int,int), getLocation() --- set and get component location
  - setSize(int,int), getSize() --- set and get component size
  - setVisible() ---show or hide the component
  - setForeground(Color), getForeground() ---set and get foreground color
  - setBackground(Color), getBackground() --- set and get background color.

- **There are two ways to create a frame**:

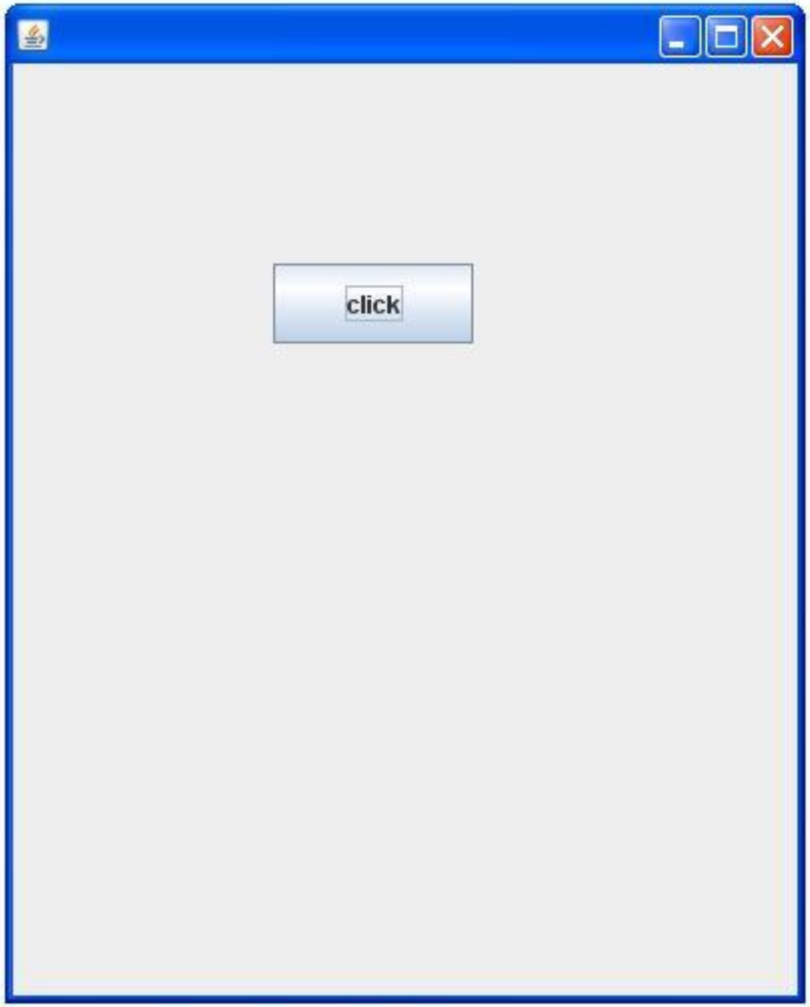
  1. By creating the object of Frame class (association)
  2. By extending Frame class (inheritance)

- We can write the code of swing inside the main(), constructor or any other method.

Creating jfames(Method-1) (using JFrame object)

```
import javax.swing.*;
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
}
}
```

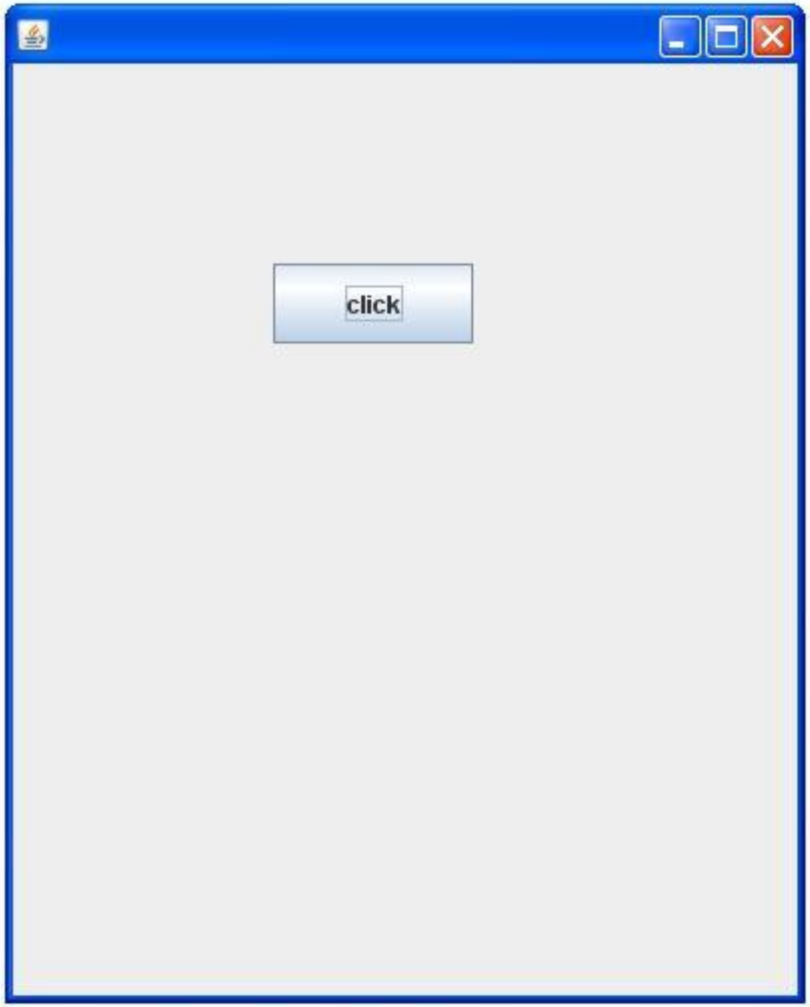
Output:



(Method-2)(by extending JFrame class)

```
import javax.swing.*;                                    //main is in different class
class MyFrame extends JFrame
{
        MyFrame()
        {
            JButton b=new JButton("click");
             b.setBounds(130,100,100, 40);
             add(b);
            setSize(400,500);
             setLayout(null);
             setVisible(true);
         }
}
class ExFrame
{
        public static void main( String args[] )
        {
            new MyFrame();
         }
}
```

Output:



(Method-2)(by extending JFrame class)

```
import javax.swing.*;                    //main is in same class
class MyFrame extends JFrame
{
        MyFrame()
        {
            JButton b=new JButton("click");
             b.setBounds(130,100,100, 40);
             add(b);
              setSize(400,500);
             setLayout(null);
             setVisible(true);
         }
        public static void main( String args[] )
        {
            new MyFrame();
         }
}
```

Container

- The **Container** class is a subclass of **Component**.
- It has additional methods that allow other **Component** objects to be nested within it.

  Ex:- **Window, Frame,** and **panel** are examples of containers.
- A container is responsible for laying out (that is, positioning) any components that it contains.
- It does this through the use of various layout managers

- Methods defined in a class ***Container*** are:
  - setLayout(LayoutManager) -- sets layout manager for display.
  - add(Component) -- adds component to the container
  - remove(Component) -- removes component from container.

Window :

- Window is a type of container, which has two-dimensional surface that can be displayed on an output device.
- It does not have **title bar**, **menu bar**, **borders**, and **resizing** corners.

Frame :

- It is a type of **Window** with a **title bar, borders,** and **resizing corners**.
- Methods defined in a *Frame* class are
  - setTitle(String), getTitle() --- set or get title
  - setMenuBar(MenuBar) --- set menu bar for window

Layout Manager:

- A Layout Manager is used to position and place **components** in a **Container**.

Frames

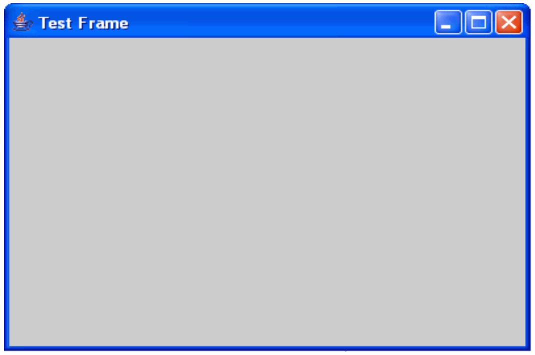
- Frame is a window that is not contained inside another window.

- Frame is the basis to contain other user interface components in **Java graphical applications**.

- **Frame**'s constructors:

  Frame( )

  Frame(String *title*)

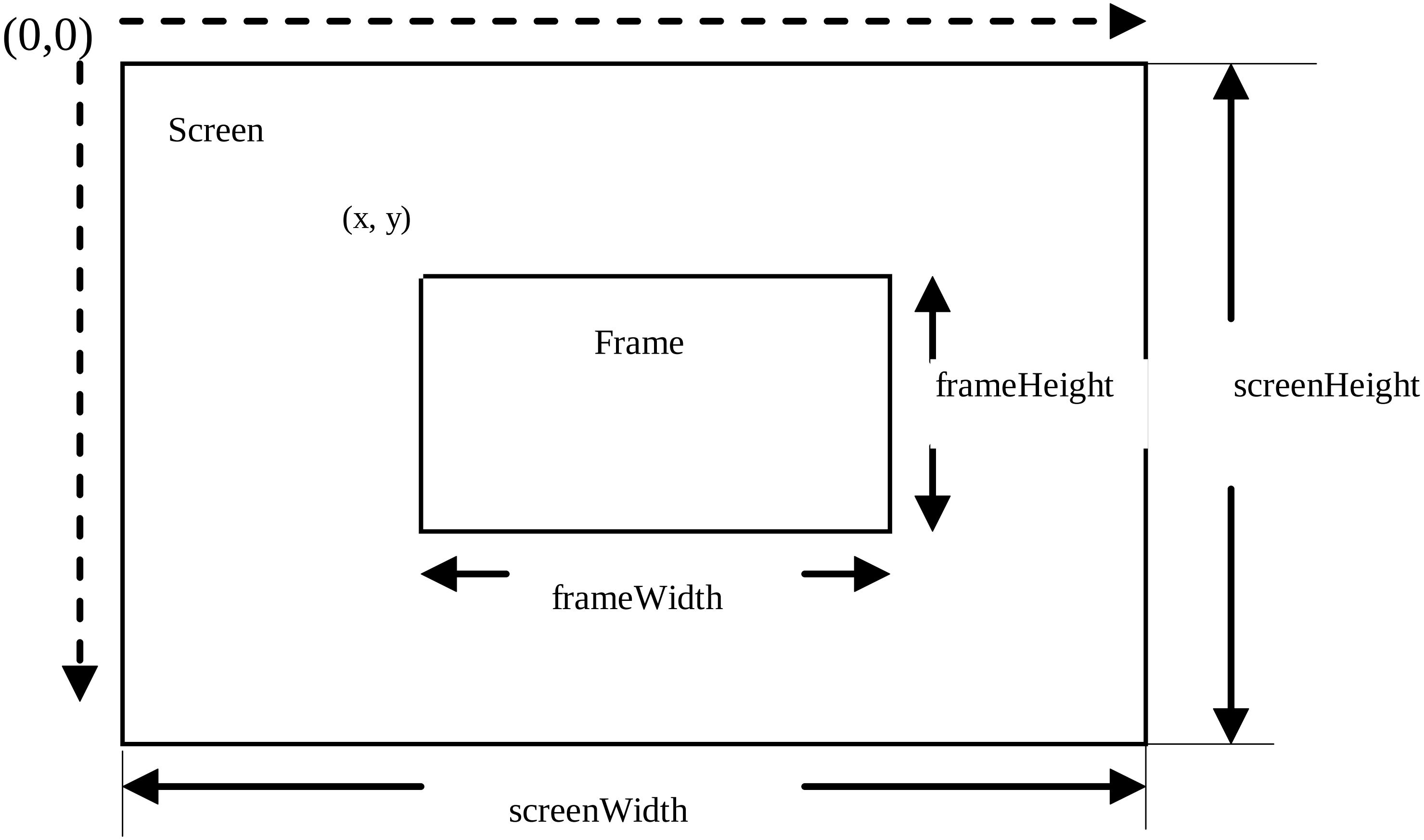


- After a frame window has been created, it will not be visible until you call **setVisible(true)**.

Frame Location

- By default, a frame is displayed in the upper-left corner of the screen.

- To display a frame at a specified location, use **setLocation(x,y)** method.


(0,0)
Screen
(x, y)
Frame
frameHeight
screenHeight
frameWidth
screenWidth

creating frames:
Method-I (using Frame object)

```
import java.awt.*;
public class MyFrame
{
    public static void main(String args[])
    {
            Label l=new Label("userId");
            Frame f = new Frame("MyFrame");
            f.add(l);
            f.setSize(300,200);
            f.setVisible(true);
    }
}
```

Method-II (by extending Frame class)

```
import java.awt.*;                                    //main is in different class
class MyFrame extends Frame
{
            MyFrame()
            {
                        super("title of Frame");
                        Label l=new Label("userId");
                        add(l);
                        setSize(300,200);
                        setVisible(true);
            }
}
class ExFrame
{
    public static void main( String args[] )
    {
            new MyFrame();
    }
}
```

Method - II (by extending Frame class)

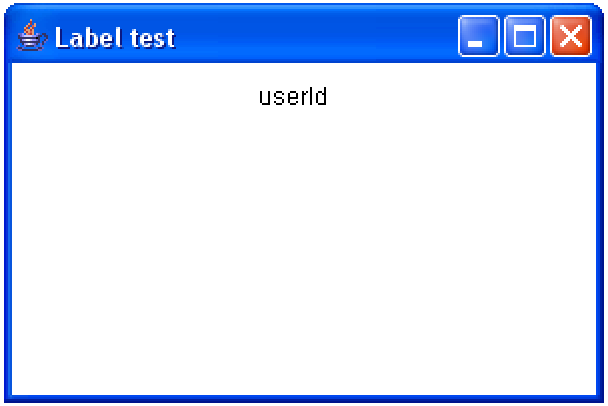
```
import java.awt.*;                          //main is in same class
class MyFrame extends Frame
{
        MyFrame()
        {       super("title of Frame");
                Label l=new Label("userId");
                add(l);
                 setSize(300,200);
                setVisible(true);
        }
        public static void main( String args[] )
         {
                 new MyFrame();
         }
}
```

User-Interface Components

Label

- Label()
- Label(String text)

- setText(String text)
- getText()



userId

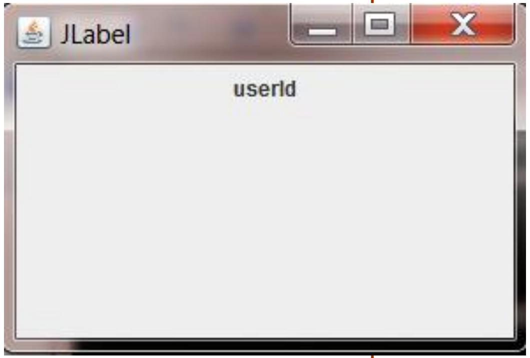
- Labels are passive controls that do not support any interaction with the user.

Label

```
import java.awt.*;
public class ExLabel extends Frame
{
        public ExLabel()
        {
            super("Label test");
            setLayout( new FlowLayout() );
            Label label1 = new Label("userId");
            add(label1);
            setSize(300,200);
            setVisible(true);
        }
        public static void main(String args[])
        {
            new ExLabel();
        }
}
```

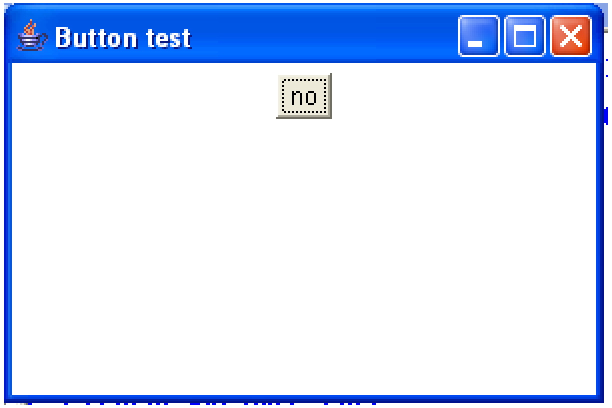
JLabel

```
import java.awt.*;
import javax.swing.*;
public class MyLabel extends JFrame
{
        public MyLabel()
        {
            super("JLabel");
            setLayout( new FlowLayout() );
            JLabel label1 = new JLabel("userId");
            add(label1);
            setSize(300,200);
            setVisible(true);
        }
        public static void main(String args[])
        {
            new MyLabel();
        }
}
```



Button

- ✓Button()
- ✓Button(String title)
- ✓getLabel()
- ✓setLabel()

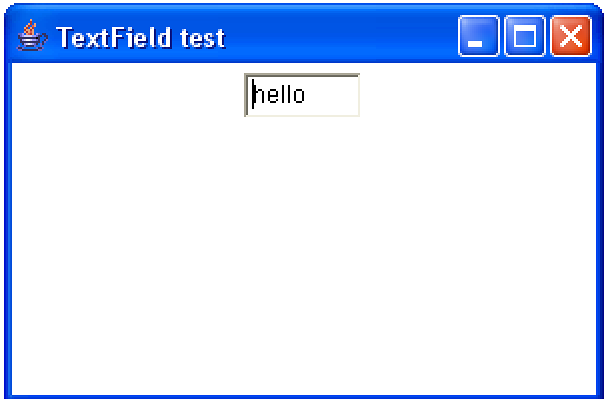


Button

```
import java.awt.*;
public class ExButton extends Frame
{
        public ExButton()
        {
            super("Button test");
            setLayout( new FlowLayout() );
            Button button1 = new Button("no");
            add(button1);
            setSize(300,200);
            setVisible(true);
        }
        public static void main(String args[])
        {
            new ExButton();
        }
}
```

Text Components

TextField



hello

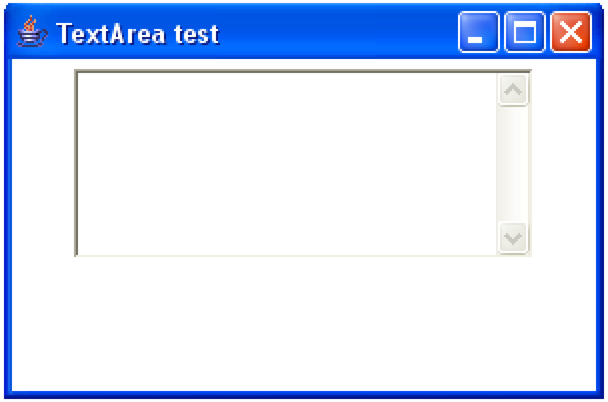
- TextField()
- TextField(int columns)
- TextField(String text)
- TextField(String text, int columns)

- setText(String)
- getText()

TextField

```
import java.awt.*;
public class ExTextField extends Frame
{
        public ExTextField()
        {
           super("TextField test");
           setLayout(new FlowLayout());
           TextField text1 = new TextField("hello",30);
           add( text1 );
           setSize(300,200);
           setVisible(true);
        }
        public static void main(String args[])
        {
             new ExTextField();
        }
}
```

TextArea



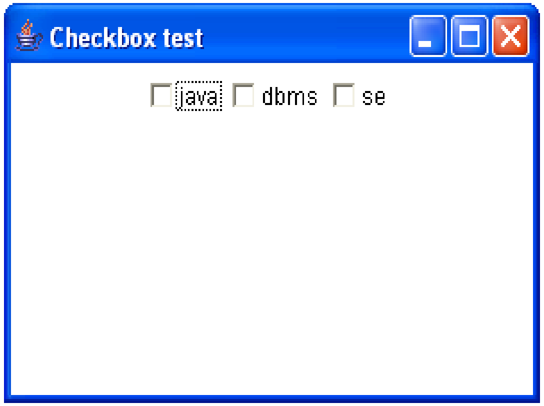
- TextArea()
- TextArea(String text)
- TextArea(int rows, int *numChars*)
- TextArea(String text , int rows, int *numChars*)

- setText(String)
- getText()

TextArea

```
import java.awt.*;
public class ExTextArea extends Frame
{
        public ExTextArea()
        {
          super("TextArea test");
          setLayout(new FlowLayout());
         TextArea text1 = new TextArea(5,30);
          add(text1);
          setSize(300,200);
          setVisible(true);
        }
        public static void main(String args[])
        {
            new ExTextArea();
        }
}
```

Checkbox



  - Checkbox()
  - Checkbox(String text)
  - Checkbox(String text, boolean state)
  - Checkbox(String text, CheckboxGroup group, boolean state)
- getLabel() – String
- setLabel() – void
- getState() – boolean
- setState() – boolean

Checkbox

```
import java.awt.*;
public class ExCheckbox extends Frame
{
     public ExCheckbox()
    {
         super("Checkbox test");
         setLayout(new FlowLayout());
         Checkbox check1 = new Checkbox("java");
         Checkbox check2 = new Checkbox("dbms");
         Checkbox check3 = new Checkbox("se");
         add( check1 );
         add( check2 );
         add( check3 );
         setSize(300,200);
         setVisible(true);
     }   ............................
```

Checkbox Groups, Choices, and Lists

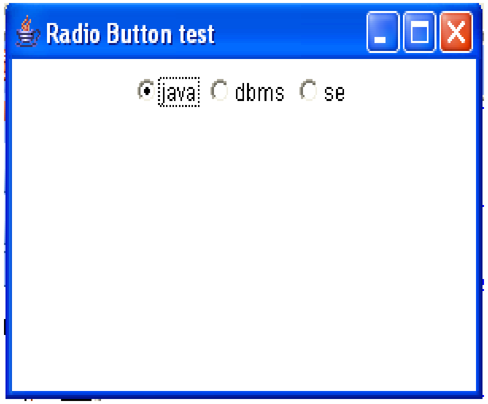
- Three types of interface components are used to select one item from a set of possibilities.
  - First is a group of connected check boxes with the property that only one can be selected at a time.(also called radio buttons).
  - Second is choice. A choice displays only one selection, but when the user clicks in the selection area, a pop-up menu appears that allows the choice to be changed to a different selection.
  - A third is a List. A List is similar to a choice, but several items out of the range can be displayed at a time.
- Use checkbox Group when the number of alternatives is small.

Differences:

Checkbox group	Choice	List
It is a collection of items	It is a collection of items	It is a collection of items
Select only one item	Select only one item	Can be selected more than one item
Can see all items	Can see only one item	Can see more than one item

CheckboxGroup / Radio buttons

  - CheckboxGroup()
  - Checkbox(String text, CheckboxGroup group, boolean state)
- Checkbox getSelectedCheckbox( )
- void setSelectedCheckbox(Checkbox *which*)



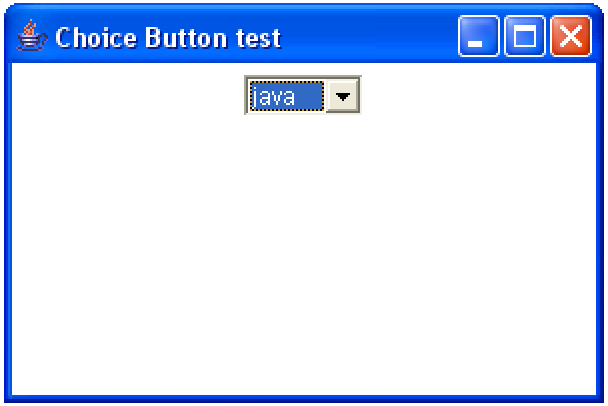
● java ○ dbms ○ se

CheckboxGroup/Radio Buttons

```
import java.awt.*;
public class ExRadioButton extends Frame
{
      public ExRadioButton()
     {
        super("Radio Button test");
        setLayout(new FlowLayout());
        CheckboxGroup cbg = new CheckboxGroup();
        Checkbox check1 = new Checkbox("java",cbg,true);
        Checkbox check2 = new Checkbox("dbms",cbg,false);
        Checkbox check3 = new Checkbox("se",cbg,false);
        add(check1);
        add(check2);
        add(check3);
        .....................
```

Choice

- Choice()
- add(String) – void ,Items are added to the list in the order in which calls to **add( )** occur.
- getItemCount() – int
- getItem(int) – String
- remove(int) –void
- getSelectedItem() – String



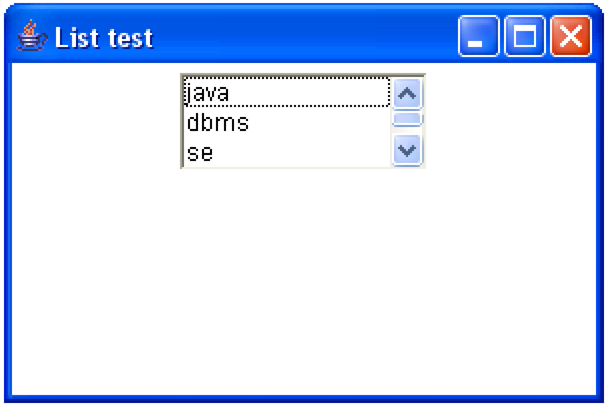
Choice

```
import java.awt.*;
public class ExChoice extends Frame
{
    public ExChoice()
    {
        super("Choice Button test");
        setLayout(new FlowLayout());
       Choice choice1 = new Choice();
       choice1.add("java");
       choice1.add("dbms");
       choice1.add("se");
       add(choice1);
       setSize(300,200);
       setVisible(true);
       ....................
```

List

- List()
- List(int rows)
- List(int rows, boolean multipleMode)

- add(String) – adds item to the end of the list
- add(String,int index) – adds item at index
- getItemCount()
- getItem(int)
- remove(int)
- getSelectedItem()
- getSelectedItems()



List

```
import java.awt.*;
public class ExList extends Frame
{
            public ExList()
            {
                        super("List test");
                        setLayout(new FlowLayout());
                        List List1 = new List(3,false);
                        List1.add("java");
                        List1.add("dbms");
                        List1.add("se");
                        List1.add("ppl");
                        List1.add("co");
                        add( List1 );
                        ...................
```

MenuBars , Menus, MenuItems

- To create a menu bar, first create an instance of **MenuBar**.

  Consturctor - MenuBar( )

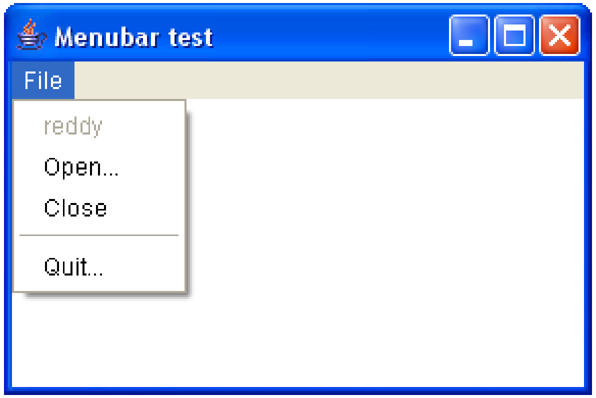
  – This class only defines the default constructor.

  – a menu bar contains one or more **Menu** objects.

- Next, create instances of **Menu (menus)**

  Constructors

  - Menu( )
  - Menu(String *menuName*)



- Individual menu items are of type **MenuItem**.

  Constructors

  - MenuItem( )
  - MenuItem(String itemName)
  - MenuItem(String *itemName*, MenuShortcut *keyAccel*)

MenuBars , Menus, MenuItems

```
import java.awt.*;
public class ExMenubar extends Frame
{
        public ExMenubar()
        {
           super("Menubar test");
           MenuBar mbar = new MenuBar();
           setMenuBar(mbar);
           Menu file = new Menu("File");
           file.add(new MenuItem("New..."));
           file.add(new MenuItem("Open..."));
           file.add(new MenuItem("Close"));
           file.add(new MenuItem("-"));
           file.add(new MenuItem("Quit..."));
           mbar.add(file);          ........................
```

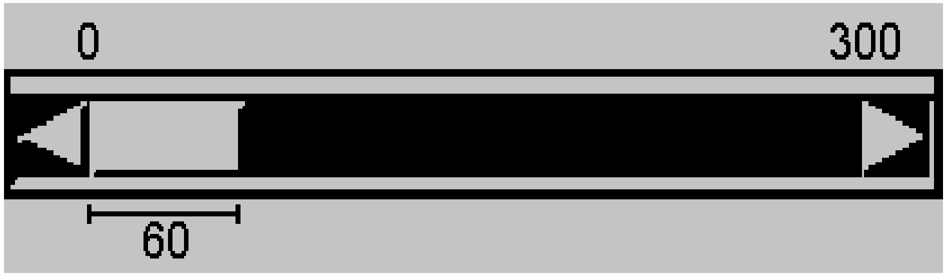
Scrollbar

- *Scroll bars* are used to select continuous values between a specified minimum and maximum.

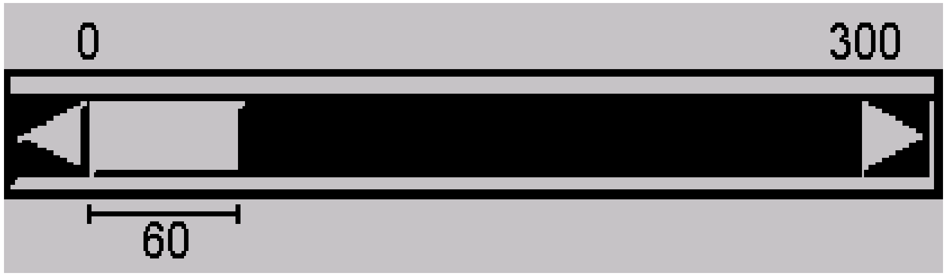
- The maximum and minimum values can be specified .

- The **line increment** can be specified (the amount scroll bar will move when touched the line ends).

- The **page increment** can be specified (the amount scroll bar will move when touched in the background area between the thumb and the end).



- Scrollbar()
  - Scrollbar(int *style*)
  - Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*)
  - Constants
    - Scrollbar.VERTICAL
    - Scrollbar.HORIZONTAL
- Scrollbar(Scrollbar.HORIZONTAL,0, 60, 0, 300);



- **Minimum : default 0.**
- **Maximum : default 100**

- **Default line increment is 1 unit**
- **Default page increment is 10 units.**

Scrollbar

```
import java.awt.*;
public class ExScrollbar extends Frame
{
    public ExScrollbar()
   {
      super("Scrollbar test");
      setLayout(new FlowLayout());
      Scrollbar scroll1 = new Scrollbar(Scrollbar.HORIZONTAL);
      Scrollbar scroll2 =
              new Scrollbar(Scrollbar.HORIZONTAL,100, 60, 0, 300);
      add( scroll1 );
      add( scroll2 );
      ...........................
```

Panel

Which are containers, used to organize and control the layout of o their components such as labels, buttons, text fields, and so on.

✓ A **Panel** looks like a window that does not contain a title bar, m enu bar, or border.

✓**It is recommended that you place the user interface compon ents in panels and place the panels in a frame.**

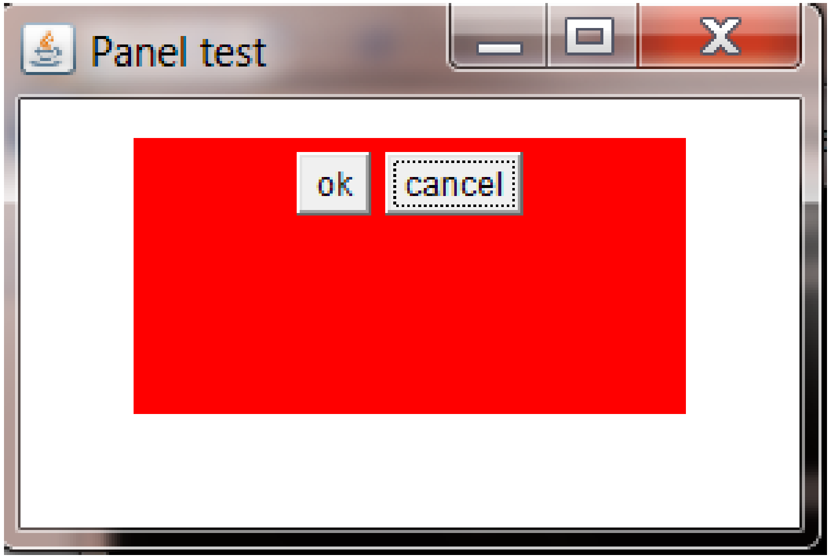
- You can also place panels in a panel.
- **FlowLayout** is the default layout for panel.

  - Panel()
  - Panel(LayoutManager layout)

Panel

```
import java.awt.*;
public class Ex013Panel extends Frame
{
        public Ex013Panel()
        {
                super("Panel test");
                setLayout(null);
                Panel pan1 = new Panel();
                pan1.setSize(200,100);
                pan1.setBackground(Color.red);
                pan1.setLocation(50,50);
                Button button1 = new Button("ok");
                Button button2 = new Button("cancel");
                pan1.add( button1 );
                pan1.add( button2 );
                add(pan1);
                ……………………
```

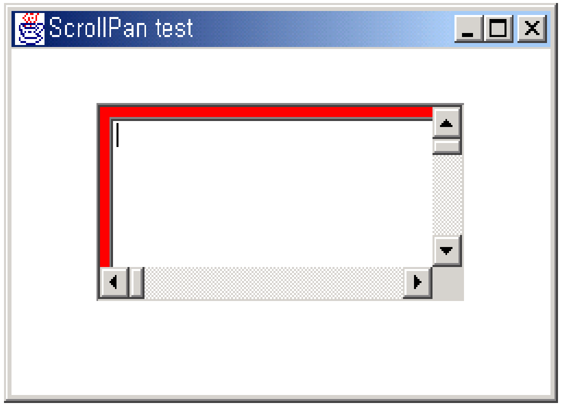
Panel Output



ScrollPane

- It is similar to a panel.
- It can hold only one Component.
- If size of the component held is larger than the size of the ScrollPane, scroll bars will be automatically generated.
- It does not have a LayoutManager

  - ScrollPane()



ScrollPane

```
import java.awt.*;
public class ExScrollPane extends Frame
{
        public ExScrollPane()
        {
                super("ScrollPane test");
                setLayout(new FlowLayout());
                ScrollPane sPane1 = new ScrollPane();
                sPane1.setSize(200,100);
                sPane1.setBackground(Color.red);
                sPane1.setLocation(50,50);
                TextArea text1 = new TextArea(300,500);
                sPane1.add(text1);
                add(sPane1);
                ......................
```

Graphics

- **Graphics** object draws pixels on the screen that represent text and other graphical shapes such as lines, ovals, rectangles, polygons etc.

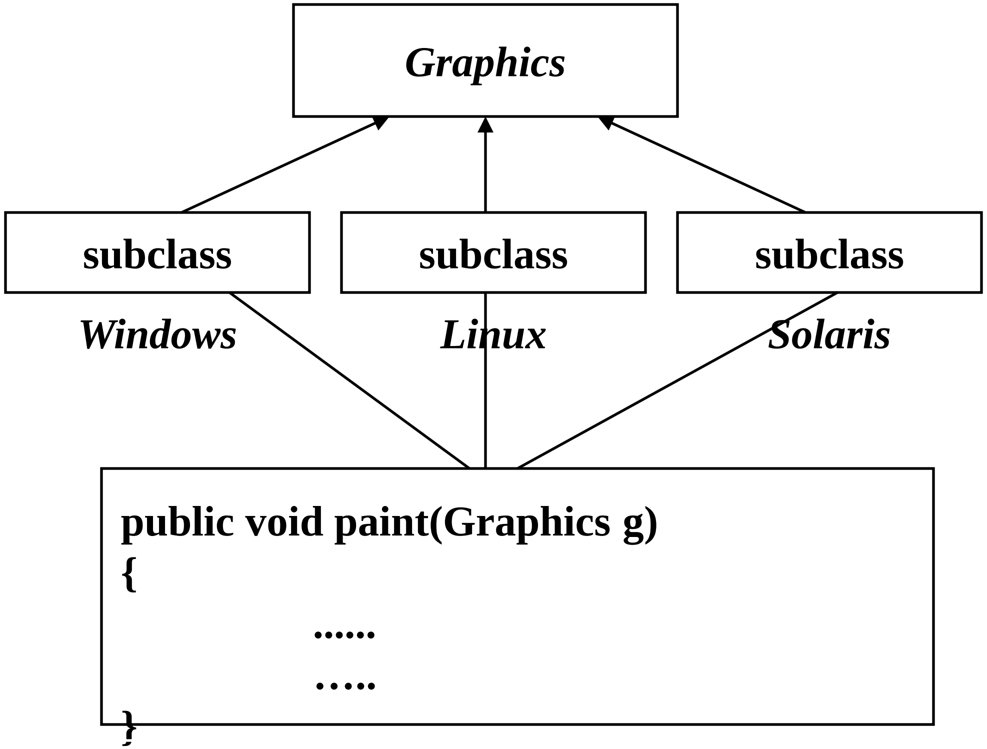
- The graphics class is an abstract class(i.e. Graphics objects can not be instantiated).

- When java is implemented on each platform, a subclass of Graphics is created that implements the drawing capabilities.

- Before you can do any drawing, you have to get a *graphics context* object (subclass of Graphics).

- The best way to do that is to place all the code that does your drawing in the paint method of a component that's added to a frame or panel.

- The paint method receives an instance of the system-specific subclass that extends Graphics for the component as a parameter.



```
public void paint(Graphics g)
{
                ......
                …..
}
```

Component (Frame / Panel / canvas / Applet)

```
Class myFrame extends Frame
{
              ……..
              ……..
    public void paint(Graphics g)
   {
              g.drawOval(x1,y1,width,height);
              ……
   }
}
```

public void paint(Graphics g)

- The ***paint(Graphics g)*** method is common to all *components* and *containers*.

- **Needed if you do any drawing or painting.**

- *Never call* paint(*Graphics*), *call* repaint( )

How does the *paint(Graphics g)* method get called?

- It is *called automatically by Java* whenever the component or container is loaded, resized, minimized, maximized.

- You can cause the paint method to be called at any time by calling the component's repaint() method.

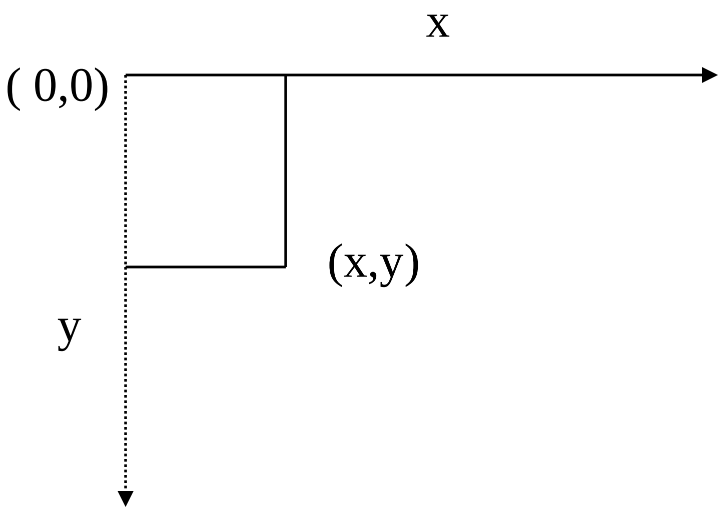
- Call repaint( ) when you have changed something and want your changes to show up on the screen.
  - You do *not* need to call repaint() when something happens in Java's own components (Buttons, TextFields, etc.)
  - You *do* need to call repaint() after drawing commands (drawRect(...), fillRect(...), drawString(...), etc.)

The *repaint()* method will do two things:

1. It calls *update(Graphics g)*, which writes over the old drawing in background color (thus erasing it).

2. It then calls *paint(Graphics g)* to do the drawing.

Java coordinate system



Graphics methods for drawing shapes

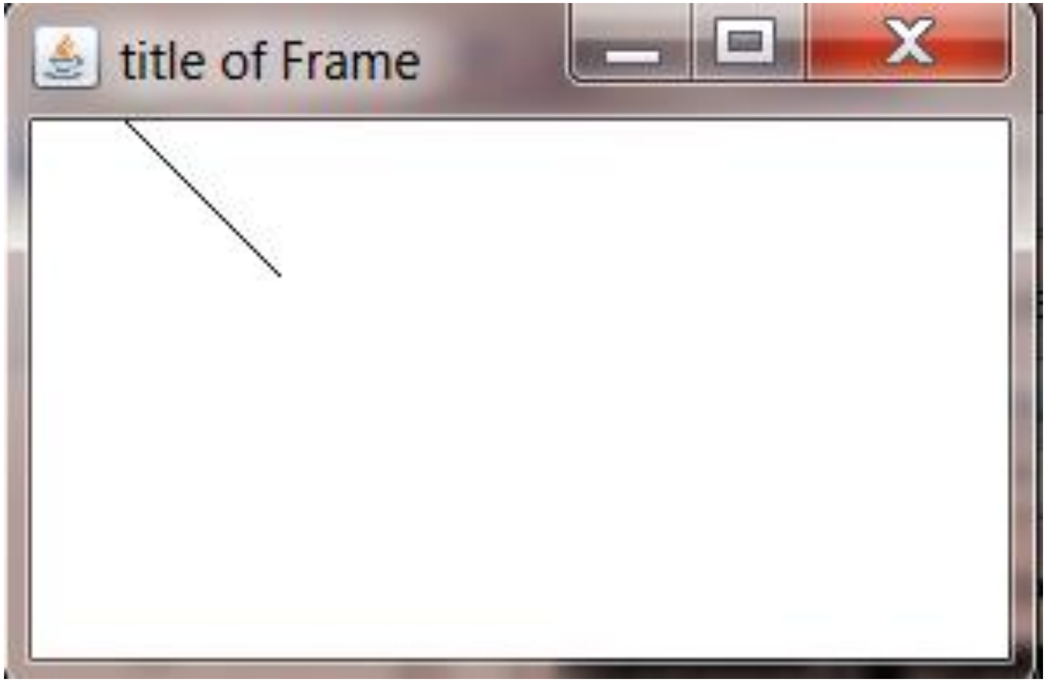
- **g.drawString (str, x, y);**
  - **Puts string at x,y**
- **g.drawLine(x1, y1, x2, y2)**
  - Line from **x1**, **y1** to **x2**, **y2**

- **g.drawRect(x1, y1, width, height)**
  - Draws rectangle with upper left corner **x1, y1**
- **g.fillRect(x1, y1, width, height)**
  - **Draws a solid rectangle.**

Example1:

```
import java.awt.*;
class MyGraphics extends Frame
{
          MyGraphics()  {
             super("title of Frame");
             setSize(300,200);
             setVisible(true);
          }
         public void paint(Graphics g) {
           g.drawLine(30, 30, 80, 80);
         }
}
class MainGraphics
{
          public static void main( String args[] )
          {
                     new MyGraphics();
           }
 }
```

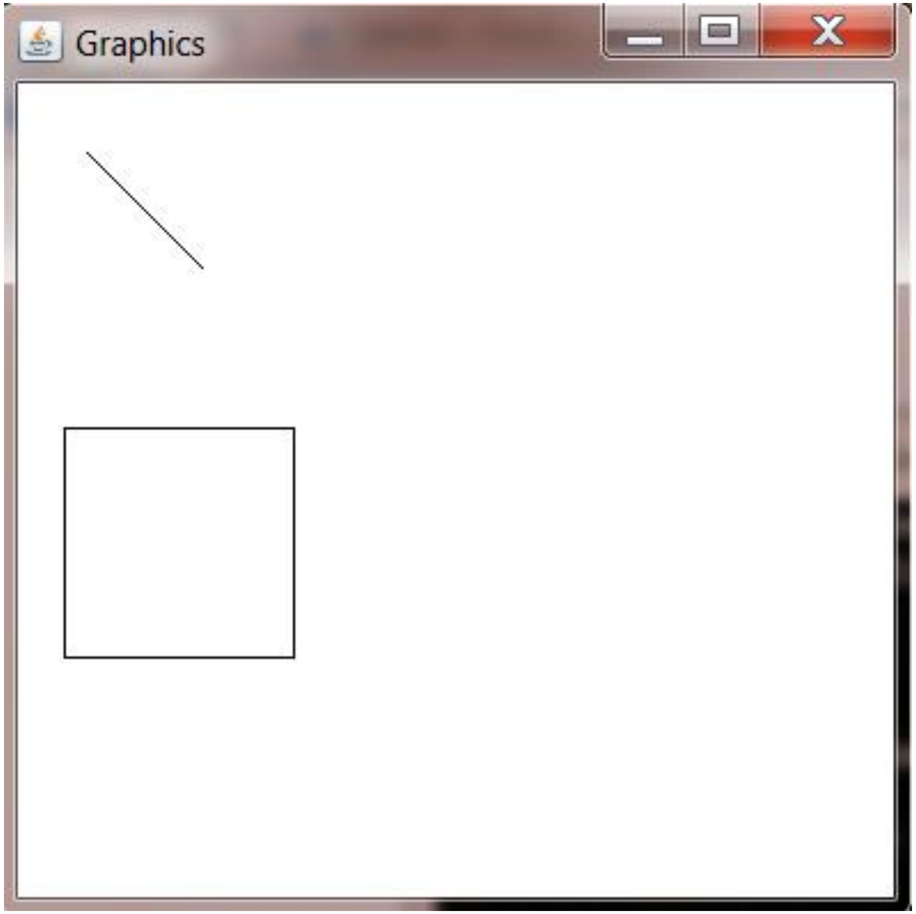
Output:



Example2:

```
import java.awt.*;
class Gcanvas extends Canvas{
    public Gcanvas(){   setSize(200, 200);  }
    public void paint(Graphics g)  {
         g.drawLine(30, 30, 80, 80);
        g.drawRect(20, 150, 100, 100);
   }
}
public class Gra extends Frame
{
            public Gra() {
                super("Graphics");
                Gcanvas g = new Gcanvas();
                add(g);
                setSize(400,400);
                setVisible(true);
            }
        public static void main(String args[])
        {
            new Gra();
        }
}
```

Output:



Layout Managers

- Arranges the GUI components on a container.

- Every container has a default Layout Manager:
  - Panels – FlowLayout
  - Window (e.g. Frames, etc.) – BorderLayout

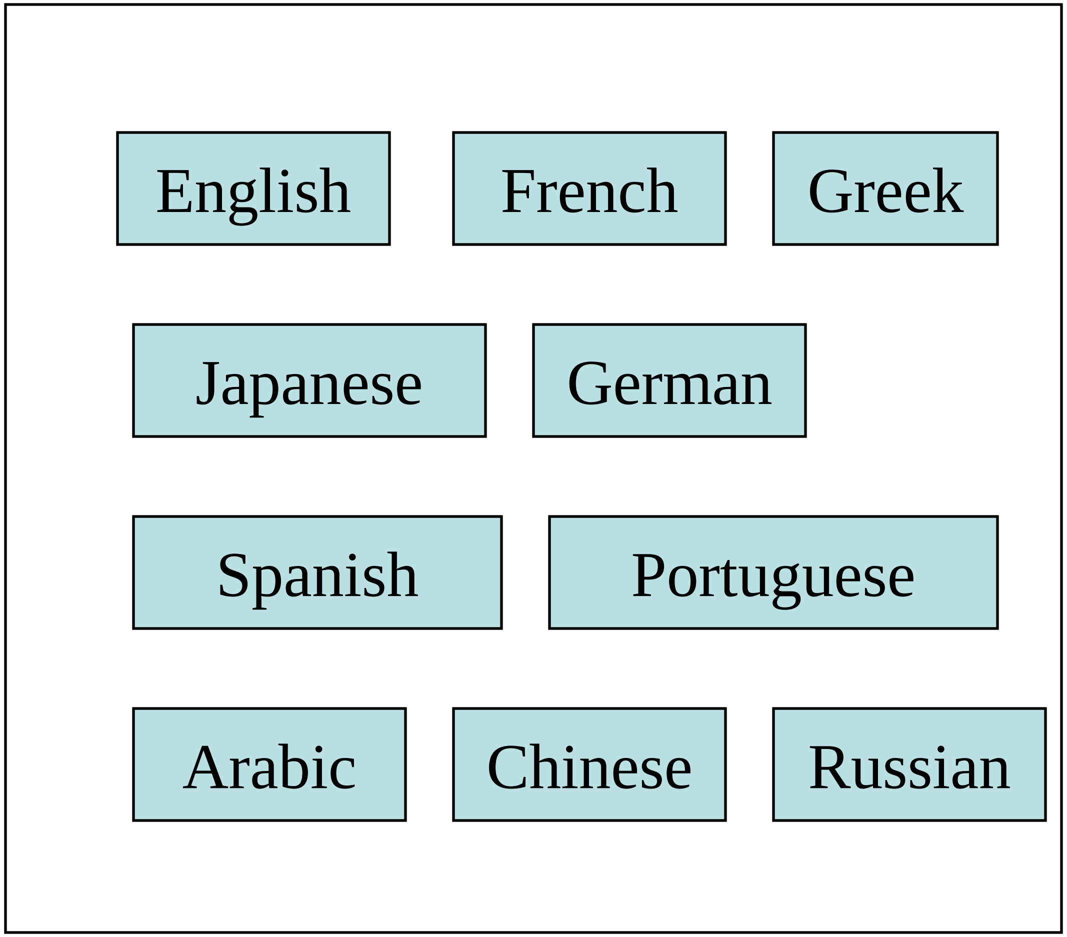
- Usage:
  - myContainer.setLayout(new LayoutManger());

- Layout Managers
  - Flow Layout
  - Grid Layout
  - Border Layout
  - Card Layout
  - Gridbag Layout

Flow Layout

- The Flow Layout manager arranges the components left-to-right, top-to-bottom in the order they were inserted into the container.
- When the container is not wide enough to display all the components, the remaining components are placed in the next row, etc.
- By default each row is centered.
- The line alignment can be:
  - **FlowLayout.LEFT**
  - **FlowLayout.CENTER**
  - **FlowLayout.RIGHT**

Flow Layout Example



English French Greek

Japanese German

Spanish Portuguese

Arabic Chinese Russian

Flow Layout Constructors

FlowLayout(align, hgap, vgap)

align – alignment used by the manager

hgap – horizontal gaps between components

vgap – vertical gaps between components

FlowLayout(align)

align – alignment used by the manager

A default 5-unit horizontal and vertical gap

FlowLayout()

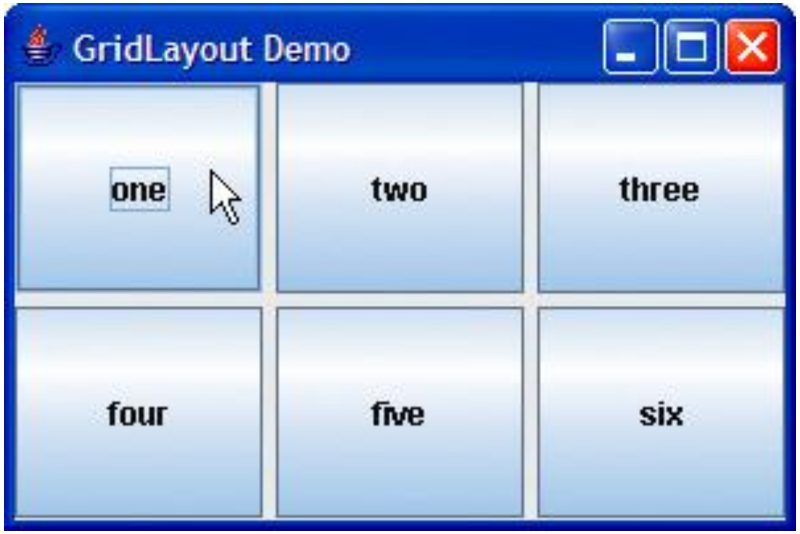
A centered alignment and a default 5-unit horizontal and vertical gap

Grid Layout

- Container is divided into a grid where components are placed in rows and columns.

- Every component has the same width and height.

Grid Layout Examples



Grid Layout Constructors

GridLayout(r, c, hgap, vgap)

r – number of rows in the layout

c – number of columns in the layout

hgap – horizontal gaps between components

vgap – vertical gaps between components

GridLayout(r, c)

r – number of rows in the layout

c – number of columns in the layout

No vertical or horizontal gaps.

GridLayout()

A single row and no vertical or horizontal gaps.

Border Layout

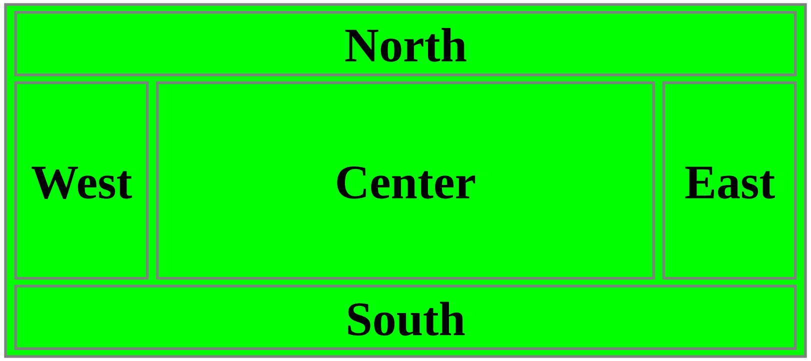
- The Border Layout manager arranges components into five regions: **North, South, East, West,** and **Center**.

- Components in the North and South are set to their natural heights and horizontally stretched to fill the entire width of the container.

- Components in the East and West are set to their natural widths and stretched vertically to fill the entire height of the container.

- The Center component fills the space left in the center of the container.

BorderLayout Manager



Usage:- add(new Button(“ok”), BorderLayout.NORTH);

Border Layout Constructors

BorderLayout(hgap, vgap)

hgap – horizontal gaps between components

vgap – vertical gaps between components

BorderLayout()

No vertical or horizontal gaps.

Border Layout Constraints

- The positional constraints are:
  - **BorderLayout.NORTH**
  - **BorderLayout.SOUTH**
  - **BorderLayout.EAST**
  - **BorderLayout.WEST**
  - **BorderLayout.CENTER**

CardLayouts

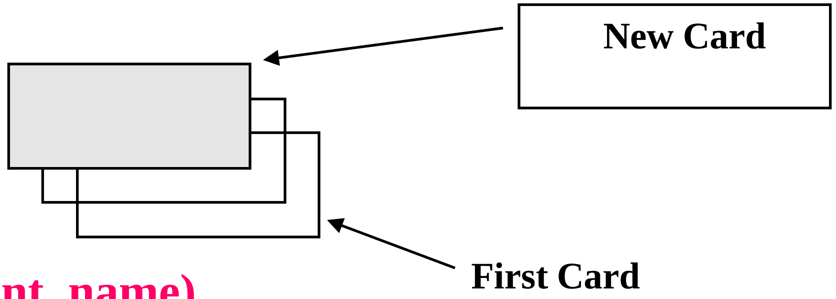
- CardLayout places components (usually panels) on top of each other like a stack.
- You can see only one card at a time.
- By default, the first card is visible.
- We can put any card on top using the methods **next(), previous(), first(), last(),** and **show().**

Constructor

-CardLayout()

Methods:-

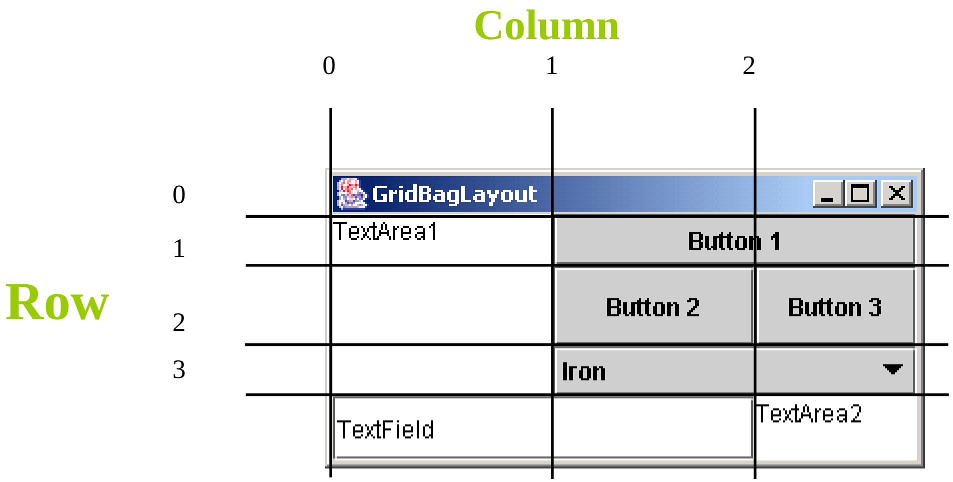
- public void first(Container c);
- public void next(Container c);
- public void previous(Container c);
- public void last(Container c);
- public void show(Container c, String name);

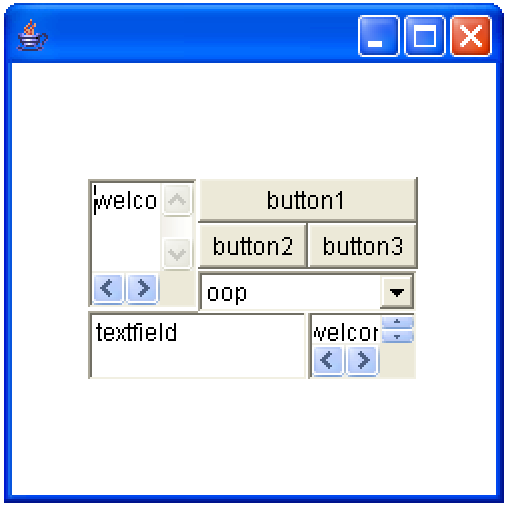


add(component, name)

GridBagLayout

- Flexible GridBagLayout
  - Components can vary in size
  - Components can occupy multiple rows and columns
  - Components can be added in any order


Column
0
1
2
Row
0
1
2
3
GridBagLayout
TextArea1
Button 1
Button 2
Button 3
Iron
TextField
TextArea2


welco
button1
button2
button3
oop
textfield
welcor

Events

The Delegation Event Model

- *The delegation event model* is a modern approach which defines mechanisms to generate and process events.

- Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*.

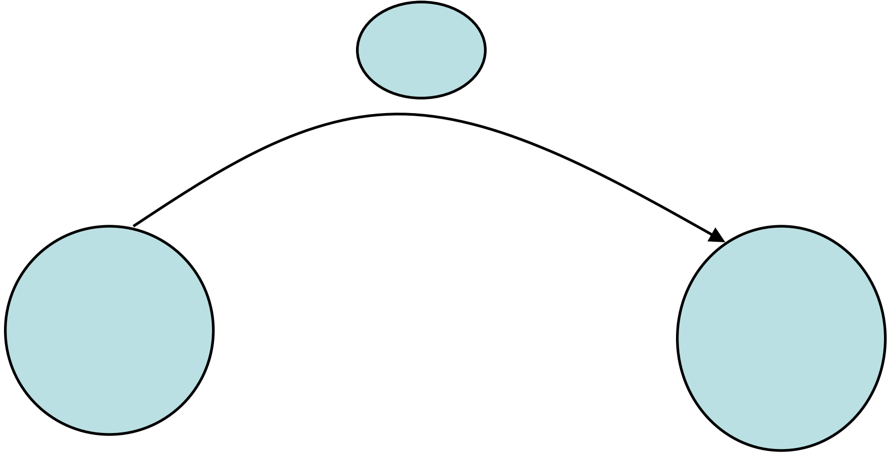
- In this scheme, the listener simply waits until it receives an event.

- Once 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.

- A user interface element is able to “delegate” the processing of an event to a separate piece of code.

Event object

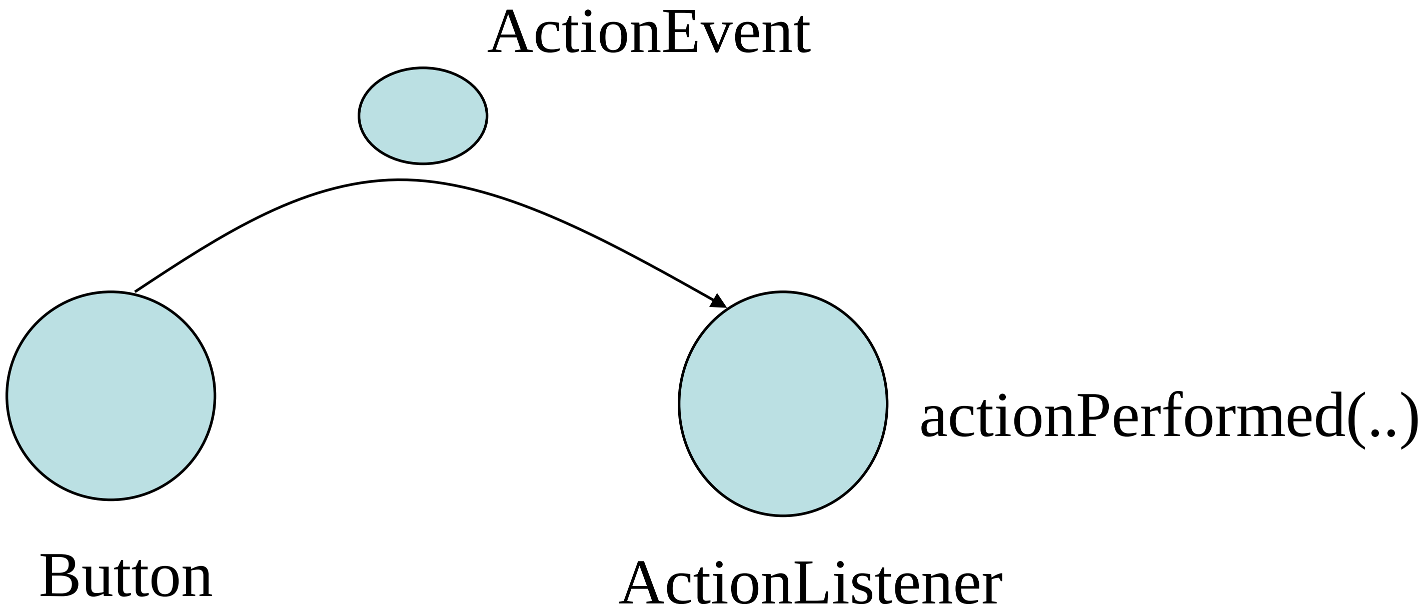


Events handling methods

Event source

Event listener

Action Events on Buttons



Example:

```
import java.awt.*;
import java.awt.event.*;
public class ButtonAction
    extends Frame
{
    ButtonAction()
    {
    setLayout(new
    FlowLayout());
    Button b=new Button("ok");
           add(b);
    b.addActionListener(new
    MyActionListener( )  );
    setSize(200,200);
    setVisible(true);
}
```

```
public static void main(String[]
    args)
{
    new ButtonAction();
}
}
class MyActionListener
    implements
           ActionListener
{
 public void
    actionPerformed(ActionEve
    nt ae )
   {
     System.out.println("button
    clicked");
    }
  }
```

- All Events are objects of Event Classes.

- Events can be generated as a result of a person interacting with elements in a graphical user interface.

- Events may also occur without person interactions with a user interface.

  - For example, an event may be generated.
    - ✓ when a timer expires.
    - ✓ a counter exceeds a value.
    - ✓ a software or hardware failure occurs.

Event sources

- Event source is a GUI component with which user interacts.
- Event 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 listener has its own registration method.
- Here is the general form:
- public void add*Type*Listener(*Type*Listener *el*)
- Here, *Type* is the name of the event and *el* is a reference to the event listener.

- For example,

- The method that registers a keyboard event listener is called **addKeyListener( )**.

- The method that registers a mouse event listener is called **addMouseListener( )**.

- When an event occurs, all registered listeners are notified and receive a copy of the event object.

Event Classes

- The classes that represent events .
- All event classes are defined in *java.awt.event* package.
- The following are the most important event classes.

Event Class	***Description***
ActionEvent	Generated when a button is clicked, a list item is double-clicked, or a menu item is selected.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
KeyEvent	Generated when the key is pressed , key is released, or key is typed.

TextEvent	Generated when the value of a text area or text field is changed.
MouseWheelEvent	Generated when the mouse wheel is moved.
WindowEvent	Generated when a window is activated, deactivated, deiconified, iconified, opened, closing, closed.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
FocusEvent	Generated when a component gains or loses keyboard focus.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ContainerEvent	Generated when a component is added to or removed from a container.

Event Listeners

- A listener is a class that is notified when an event occurs.
- It has two major requirements:
    - It must have been registered with one or more sources to receive notifications about specific types of events
    - It must implement methods to receive and process these notifications.
- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**.
- The listener class must implement listener interface to handle events.

Listener Interfaces

- In addition to the Event classes, there are Listener interfaces corresponding to each Event class.
- Some of them are listed here:

ActionEvent	**ActionListener**
MouseEvent	**MouseListener** **MouseMotionListener**
KeyEvent	**KeyListener**
TextEvent	**TextListener**
AdjustmentEvent	**AdjustmentListener**
ContainerEvent	**ContainerListener**
FocusEvent	**FocusListener**
ItemEvent	**ItemListener**
TextEvent	**TextListener**
WindowEvent	**WindowListener**

ActionEvent

ActionListener

```
public interface ActionListener
{
void actionPerformed(ActionEvent ae)

}
```

Registration method:
C_ref.addActionListener(
Listener);

```
class MyActionListener implements ActionListener
{
  public void actionPerformed(ActionEvent ae)
  {
              // Handler_ code
  }

}
```

MouseEvent

MouseListener

```
public interface MousListener{
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
}
```

```
class MyMouseListener implements MouseListener
{
 public void mouseClicked(MouseEvent me)
 { // Handler_ code }
 public void mouseEntered(MouseEvent me)
 { // Handler_ code }
 public void mouseExited(MouseEvent me)
 { // Handler_ code }
 public void mousePressed(MouseEvent me)
 { // Handler_ code }
 public void mouseReleased(MouseEvent me)
 { // Handler_ code }
}
```

Registration methods:
C_ref.addMouseListener(
 Listener);

MouseEvent

MouseMotionListener

```
public interface MousMotionListener{
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)

}
```

Registration method:
C_ref.addMouseMotion
Listener(Listener);

```
class MyMouseMotionListener implements MouseMotionListener
{
   public void mouseDragged(MouseEvent me)
  { // Handler_ code }
  public void mouseMoved(MouseEvent me)
  { // Handler_ code }

}
```

KeyEvent

KeyListener

```
public interface KeyListener{
  void keyPressed(KeyEvent ke)
  void keyReleased(KeyEvent ke)
  void keyTyped(KeyEvent ke)

}
```

```
class MyKeyListener implements KeyListener
{
  public void keyPressed(KeyEvent me)
  { // Handler_ code }
  public void keyReleased(KeyEvent me)
  { // Handler_ code }
  public void keyTyped(KeyEvent me)
  { // Handler_ code }
}
```

Registration method:
C_ref.addKeyListener(
Listener);

Adapter classes

- An adapter class provides an empty implementation of all methods in an event listener interface.
- Adapter classes are useful when you want to receive and process only some of the events.
- You can define a new class to act as an event listener by extending one of the adapter classes and overriding only those methods in which you are interested.

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