**Unit-IV**

**Syllabus** Event Handling: Events, Event sources, Event classes, Event Listeners, Delegation event model, handling mouse and keyboard events, Adapter classes. The AWT class hierarchy, user interface components- labels, button, canvas, scrollbars, text components, check box, checkbox groups, choices, lists panels – scrollpane, dialogs, menubar, graphics, layout manager – layout manager types – border, grid, flow, card and grid bag.

**Event:**-An event is an object that describes a state change in source.

**Event sources :**A source is an object that generates an event.

A source must register listeners in order for the listener to receive notifications about a specific type of event. Each type of event has its own registration method.

**General form**  Public void addTypeListener (TypeListener el).

**Sources from which we can generate events are given below**

1. **Button:**  it generates ***ActionEvent*** when the user presses a button.
2. **Checkbox :** it generates ***ItemEvent*** when the checkbox is selected or deselected.
3. **List :** it generates ***ItemEvent*** when the user selects or deselect an item from Listbox. it also generates ***ActionEvent*** when an item is double clicked.
4. **MenuItem:** like List, it generates both **ActionEvent** and ***ItemEvent***. ActionEvent is generates when an menu item is selected and an item event is generated when a checkable menu item is selected/deselected.

**Event Listeners**

A listener is an object that is notified when an event occurs. It has two major requirements

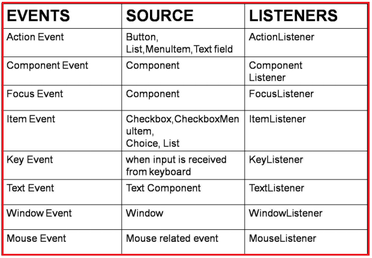
1. It must have been registered with one or more sources to receive notification about specific types of events.
2. It must implement methods to receive and process these notifications.

**Event classes**

The classes that represent events are at the core of java’s event handling mechanism. At the root of the java event classes hierarchy is **EventObject,** which is in java.util. It is the super class for all events. Its one of the constructor is

**EventObject(Object src) here src is the object that generates this event.**

The class **AWTEvent,** defined within the java.awt package is a subclass of **EventObject.** It is super class of all AWT based events used by the delegation event model.



**Event Delegation Model**

When we create a component, generally the component is displayed on the screen but is not capable of performing any actions. For example, we created a button which can be displayed but cannot perform any action, even when someone click on it. But expectation will be different. A user wants the button to perform some action.

Clicking like this is called event. An event represents a specific action done on a component. Clicking, double clicking, typing data inside component, mouse over etc are all examples of events.

When an event is generated on the component, the component will not know about it because it cannot listen to the event. To let the component understand that an event is generated on it, we should add some listener to the components. A listener is an interface which listens to an event coming from a component. A listener will have some abstract methods which need to be implemented by the programmer.

When an event is generated by the user on the component, the event is not handled by the component. On the otherhand, the component sends (delegates) that event to listener that attached to it. The listener will not handle the event. It hands over (delegates) the event to an appropriate method. Finally the method is executed and the event is handled. This is called “**event-delegation model**”.

Listener

Component

Component part Action part

Event

This method handles event

In this model the component is separated from the event is handled.

**Advantage**

1. The component and the action part can be developed in two separate environments.
2. We can modify the code for creating the component without modifying the code for action part of the component. Similarly , we can modify the action part without modifying the code for the component. Thus, we can modify one part without effecting any modification to other part. This makes debugging and maintenance of code very easy.

**/\* Demo program on Action Event\*/**

import java.awt.\*;

import java.awt.event.\*;

public class ButtonDemo extends Frame implements ActionListener

{

Button b1,b2,b3;

ButtonDemo()

{

super("Button Demo");

setSize(400,400);

setVisible(true);

b1=new Button("Red");

b2=new Button("Green");

b3=new Button("Blue");

setLayout(new FlowLayout());

add(b1);

add(b2);

add(b3);

b1.addActionListener(this);

b2.addActionListener(this);

b3.addActionListener(this);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent we)

{

System.exit(0);

}});

}

public void actionPerformed(ActionEvent ae)

{

Color c=null;

if(ae.getSource()==b1)

setBackground(Color.red);

else if(ae.getSource()==b2)

setBackground(Color.green);

else

setBackground(Color.blue);

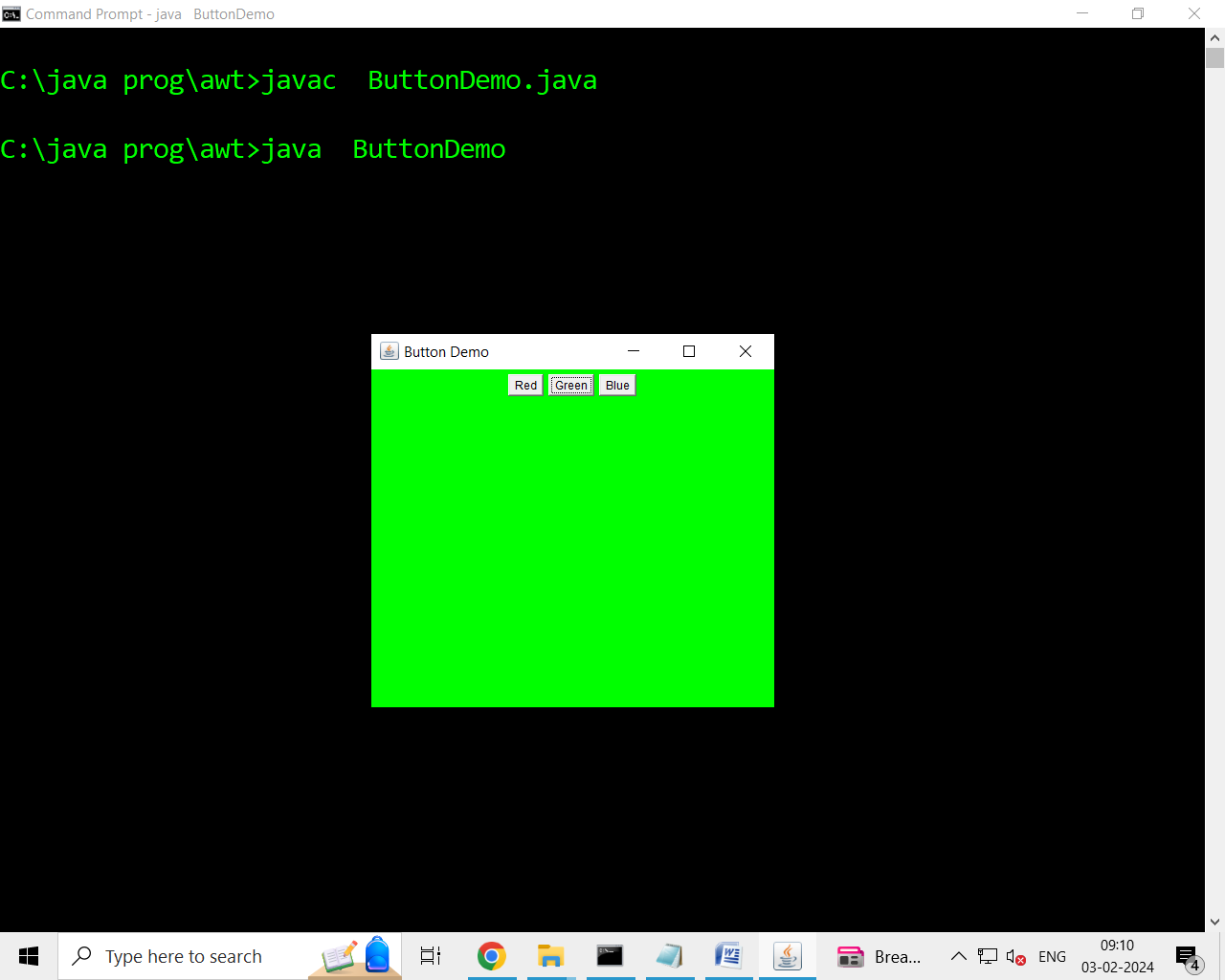
}

public static void main(String args[])

{

new ButtonDemo();

}}



**/\* Demo program on TextEvent\*/**

import java.awt.\*;

import java.awt.event.\*;

public class TextDemo extends Frame implements TextListener

{

TextField t1,t2;

TextDemo()

{

super("TextDemo");

setVisible(true);

setSize(400,400);

setLocation(300,300);

t1=new TextField(30);

t2=new TextField(30);

setLayout(new FlowLayout());

add(t1);

add(t2);

t1.addTextListener(this);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent we)

{

System.exit(0);

}});}

public void textValueChanged(TextEvent te)

{

String str=t1.getText();

t2.setText(str.toUpperCase());

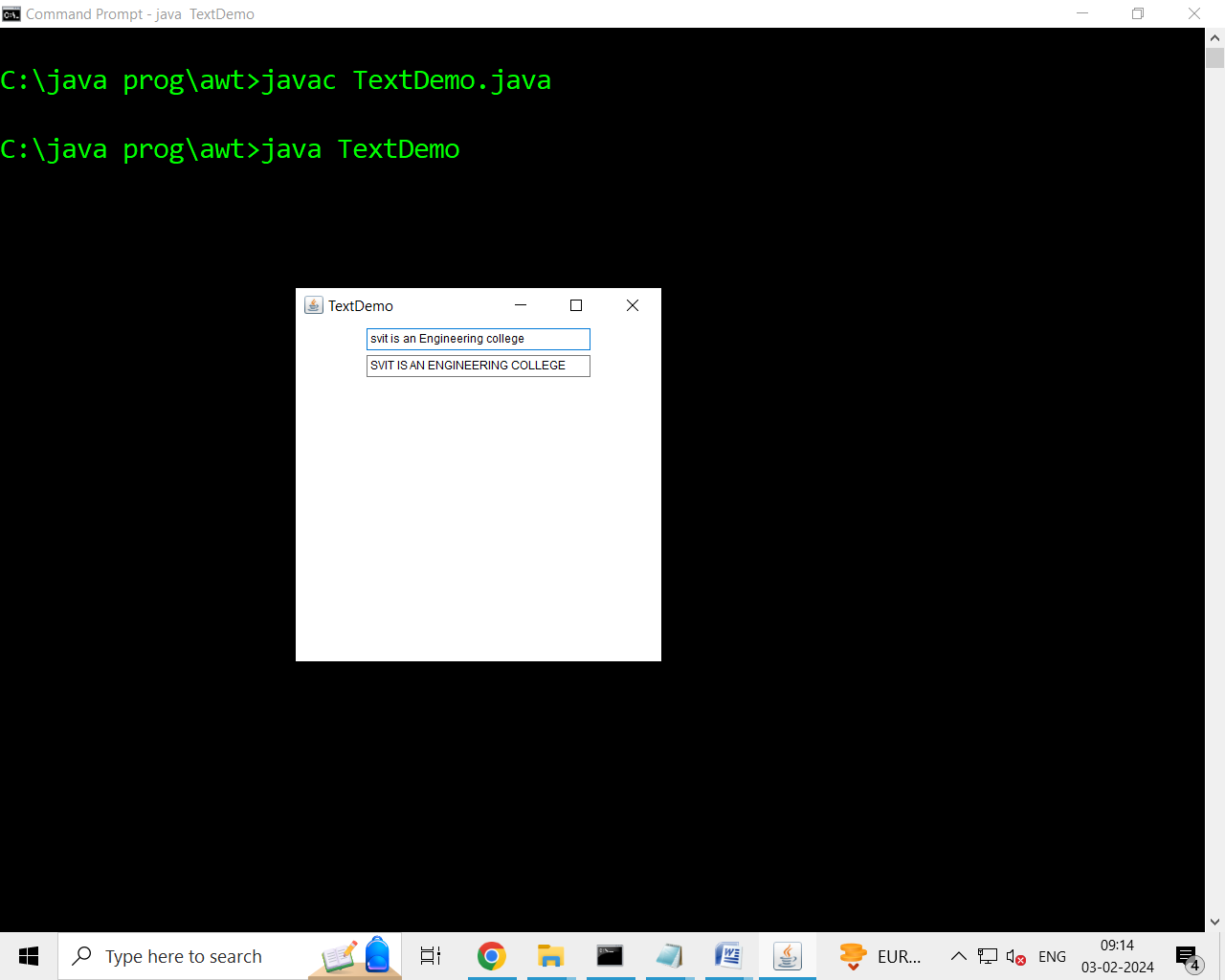
}

public static void main(String args[])

{

new TextDemo();

}}



**/\* Demo program on MouseEvent\*/**

import java.awt.\*;

import java.awt.event.\*;

public class MyTest extends Frame implements MouseListener, MouseMotionListener

{

Label l1;

MyTest()

{

l1=new Label("welcome to mouse programming");

setVisible(true);

setSize(300,300);

setLayout(new FlowLayout());

add(l1);

addMouseListener(this);

addMouseMotionListener(this);

}

public void mouseClicked(MouseEvent me)

{

l1.setText("Mouse Clicked");

}

public void mousePressed(MouseEvent Me)

{

l1.setText("Mouse Pressed");

}

public void mouseReleased(MouseEvent Me)

{

l1.setText("Mouse Released");

}

public void mouseEntered(MouseEvent Me)

{

l1.setText("Mouse Entered");

}

public void mouseExited(MouseEvent Me)

{

l1.setText("Mouse Exited");

}

public void mouseDragged(MouseEvent me)

{

l1.setText("Mouse Dragged");

}

public void mouseMoved(MouseEvent me)

{

l1.setText("MOuse MOved");

}

public static void main(String args[])

{

new MyTest();

}

}

**/\* Demo program on KeyEvent \*/**

import java.awt.\*;

import java.awt.event.\*;

public class Keydemo extends Frame implements KeyListener

{

Label l1;

Keydemo()

{

setVisible(true);

setSize(300,300);

setLayout(new FlowLayout());

addKeyListener(this);

}

public void keyTyped(KeyEvent ke)

{

}

public void keyPressed(KeyEvent ke)

{

}

public void keyReleased(KeyEvent ke)

{

int k=ke.getKeyCode();

Color c=null;

if(k==KeyEvent.VK\_F1)

{

c=Color.red;

}

if(k==KeyEvent.VK\_F1)

{

c=Color.red;

}

if(k==KeyEvent.VK\_F2)

{

c=Color.green;

}

if(k==KeyEvent.VK\_F3)

{

c=Color.blue;

}

if(k==KeyEvent.VK\_F4)

{

System.exit(0);

}

setBackground(c);

}

public static void main(String args[])

{

new Keydemo();

}

}

**Event Adapters**

When an event listener interface defines more than one method, it is often accompanied by an eventAdapter class that provides empty implementations for each of the methods. For example, the MouseListener interface defines five different methods. If your program is interested only in the mouseClicked() method, it may be easier for you to subclass the MouseAdapter class and override mouseClicked() than to implement all five methods of the MouseListener interface directly.

**/\* demo window Adapter\*/**

import java.awt.\*;

import java.awt.event.\*;

public class windowadapterDemo extends Frame

{

public windowadapterDemo()

{

super("Window Adapter Demo");

setSize(200,200);

setVisible(true);

addWindowListener(new callme());

}

public static void main(String args[])

{

new windowadapterDemo();

}}

class callme extends WindowAdapter

{

public void windowClosing(WindowEvent we)

{

System.exit(0);

}}

## List of Adapter classes in java.awt.event Adapter classes

|  |  |
| --- | --- |
| **Adapter class** | **Listener interface** |
| WindowAdapter | [WindowListener](https://www.javatpoint.com/java-windowlistener) |
| KeyAdapter | [KeyListener](https://www.javatpoint.com/java-keylistener) |
| MouseAdapter | [MouseListener](https://www.javatpoint.com/java-mouselistener) |
| MouseMotionAdapter | [MouseMotionListener](https://www.javatpoint.com/java-mousemotionlistener) |
| FocusAdapter | FocusListener |
| ComponentAdapter | ComponentListener |
| ContainerAdapter | ContainerListener |
| HierarchyBoundsAdapter | HierarchyBoundsListener |

**Introduction to AWT( Abstract Window Toolkit)**

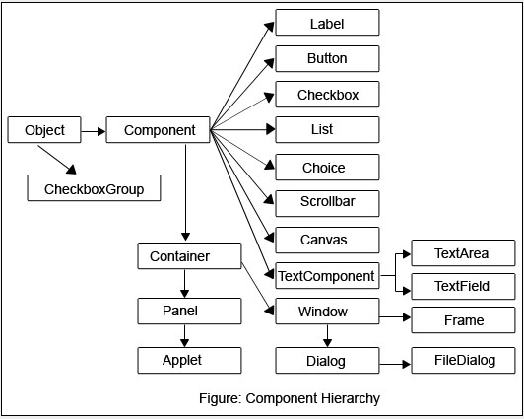
**Java AWT** (Abstract Window Toolkit) is *an API to develop Graphical User Interface (GUI) or windows-based applications* in Java.

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](https://www.javatpoint.com/package) provides [classes](https://www.javatpoint.com/object-and-class-in-java) for AWT API such as [TextField](https://www.javatpoint.com/java-awt-textfield), [Label](https://www.javatpoint.com/java-awt-label), [TextArea](https://www.javatpoint.com/java-awt-textarea), RadioButton, [CheckBox](https://www.javatpoint.com/java-awt-checkbox), [Choice](https://www.javatpoint.com/java-awt-choice), [List](https://www.javatpoint.com/java-awt-list) etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



### Components

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

### Container

The Container is a component in AWT that can contain another components like [buttons](https://www.javatpoint.com/java-awt-button), textfields, labels etc. The classes that extends Container class are known as container such as **Frame, Dialog** and **Panel**.

It is basically a screen where the where the components are placed at their specific locations. Thus it contains and controls the layout of components.

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

### Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

### Panel

The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

### Frame

The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

**Some of the classes available in AWT package**

1. **Frame (Container class)**

***Constructors***

1. **Frame() // creates a Frame without title**
2. **Frame(String) // Creates a frame with title.**

**Methods**

1. setVisible(boolean)// makes the frame to be visible
2. setSize(int,int) // sets the size of the frame
3. setLocation(int,int) //display the frame at the specified position on the screen.
4. setResizable(boolean); // make the frame resizable by passing true value

else no resizable

**User Interface Components**

**b) Label**

***Constructors***

1. Label(); //default constructor
2. Label(String); //
3. Label(String str , int alignment)
4. Label.LEFT
5. Label.Right
6. Label.CENTER

The first version creates a blank label. The second version creates a label that contains the string specified by *str.* This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how.* The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**

***Methods***

1. void setText(String);
2. string getText();
3. void setAlignment(int how);
4. int getAlignment();

we can set or change the text in a label by using the **setText( )** method. we can obtain the current label by calling **getText( )**. To obtain the current alignment, call **getAlignment( )**. Here, *how* must be one of the alignment constants

**Button**

***Constructors***

1. Button(); //creates Button with no caption on it.
2. Button(String);// creates Button with caption as given string

A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. The first version creates an empty button. The second creates a button that contains *str* as a label. After a button has been created, you can set its label by calling **setLabel( )**. we can retrieve its label by calling **getLabel( ).**

***Methods***

1. void setLabel(String);
2. String getLabel();
3. **Check boxes**

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group

***Constructors***

1. Checkbox(); //creates check box without any caption or Label
2. Checkbox(String); //create checkbox with the caption as string parameter
3. Checkbox(String, boolean); //creates check box with caption as String parameter and Boolean parameter tells whether it is checked or unchecked based on Boolean value
4. Checkbox(String, boolean, CheckboxGroup); //creates check box in mutual exclusive manner

To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. You can obtain the current label associated with a check box by calling **getLabel( )**. To set the label, call **setLabel( )**.

***Methods***

1. booleangetState();
2. void setState(Boolean);
3. String getLabel();
4. Void setLabel(String)
5. **CheckboxGroup**

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons,* because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first  
define the group to which they will belong and then specify that group when you construct the check boxes.

***Constructors***

1. CheckboxGroup();

we can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**. we can set a check box by calling **setSelectedCheckbox( ).**

***methods***

1. Checkbox getSelectedCheckbox();
2. Void setSelectedCheckbox(Checkbox);
3. **List**

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

***Constructors***

1. List();
2. List(int);
3. List(int,boolean);

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelec*t is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

***Methods***

1. void add(String); //adds an item to the end of list
2. void add(String, int); //adds the item at the index specified by *index.* Indexing begins

at zero.

1. String getSelectedItem();//determine which item is currently selected by calling

either **getSelectedItem( )** or **getSelectedIndex( )**.

1. Int getSelectedIndex();
2. String [] getSelectedItems();
3. Int[] getSelctedIndexes(); // returns an array of selected indexes
4. Int getItemCount(); // return no. of items in list
5. void select(int index); // select an item at an index specified.
6. String getItem(int); // return an item at the specified index
7. **TextField**

The **TextField** class implements a single-line text-entry area, usually called an *edit control.* Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextField** is a subclass of **TextComponent**.

***Constructors***

1. TextField();
2. TextField(int numchars);
3. TextField(String str);
4. TextField(String, int);

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str.* The fourth form initializes a text field and sets its width.

To obtain the string currently contained in the text field, call **getText( )**. To set the text, call **setText( )**.

***Methods***

1. String getText();
2. void setText(String);
3. String getSelectedText(); //select a portion of the text in a text field
4. void select(int startindex, int endindex); // programmatically you can select text
5. Boolean isEditable(); //we can control whether the contents of a text field may be modified by the

user by calling **setEditable( )**.

Void setEditable(boolean); // We can determine editability by calling **isEditable( )**.

1. Void setEchoChar(char); // specifies a single character that the **TextField** will display when characters

are entered (thus, the actual characters typed will not be shown)

1. Boolean echoCharisSet();
2. Char getEchoChar();
3. **TextArea**

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

***Constructors***

1. TextArea();
2. TextArea(int numlines ,int numchars);
3. TextArea(String str);
4. TextArea(String str ,int numlines, int numchars);

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str.* In the fifth form, you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

1. TextArea(String,int numlines,int numchars, int scrollbars);
2. SCROLLBARS\_BOTH
3. SCROLLBARS\_HORIZONTAL\_ONLY
4. SCROLLBARS\_NONE
5. SCROLLBARS\_VERTICAL\_ONLY.
6. **Choice**

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. When inactive, a **Choice** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object.

**Choice** only defines the default constructor, which creates an empty list.

**Methods**

1. void add( String name) // *name* is the name of the item being added.
2. String getSelectedItem( ) // returns the selected item.
3. int getSelectedIndex( ) // returns selected index.
4. **Scrollbar**

*Scroll bars* are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position.

**Scrollbar** defines the following constructors:  
1. Scrollbar( )  
2. Scrollbar(int *style*)   
3. Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*)

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue.* The number of units represented by the height of the thumb is passed in *thumbSize.* The minimum and maximum values for the scroll bar are specified by *min* and *max*

**Methods**

1. int getMinimum( );
2. int getMaximum( );
3. void setUnitIncrement(int *newIncr*);
4. void setBlockIncrement(int *newIncr*);

**Creating Menus in AWT**

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes **MenuBar,** **Menu** and **MenuItem.** In general, a menu bar contains one or more Menu Objects. Each Menu object contains a list of MenuItem objects. Each MenuItem object represents something that can be selected by the user. It is also possible to include Checkable menuitems. These are menu options of type **CheckboxMenuItem**  and will have a check marks next to them when they are selected.

To create a menu bar first create an instance of MenuBar. This class only defines the default constructor. Next, create instances of Menu that will define the selections displayed on the bar.

1. **MenuBar(class)**

**Constructor**

MenuBar()

**Methods**

1. Menu add(Menu)
2. **Menu(class)**

**Constructors**

1. Menu()
2. Menu(String optionName)
3. Menu(String ,Boolean removable);
4. **MenuItem(class)**

**Constructors**

1. MenuItem()
2. MenuItem(String itemName)
3. MenuItem(String, MenuShortcut)

**Methods**

1. void setEnabled(boolean)
2. boolean isEnabled();
3. Void setLabel(String )
4. String getLabel();
5. MenuItem add(MenuItem)
6. **CheckboxMenuItem**

Creates a checkable menu item by using subclass of MenuItem

**Constructors**

1. CheckboxMenuItem()
2. CheckboxMenuItem(String itemName)
3. CheckboxMenuItem(String itemName,booelan)

**Methods**

1. boolean getState();
2. void setState(boolean);

**/\* Demo program on MenuEvent\*/**

import java.awt.\*;

public class MenuDemo extends Frame

{

MenuBar mb;

Menu m1,m2,m3;

MenuItem i1,i2,i3,i4,i5;

MenuDemo()

{

super("MenuDemo");

setSize(300,300);

setVisible(true);

mb=new MenuBar();

setMenuBar(mb);

CheckboxMenuItem mm=new CheckboxMenuItem("display options",true);

MenuShortcut ms=new MenuShortcut('N');

m1=new Menu("File");

m2=new Menu("Edit");

m3=new Menu("View");

i1=new MenuItem("New",ms);

i2=new MenuItem("Open");

i3=new MenuItem("Save");

i4=new MenuItem("Cut");

i5=new MenuItem("Copy");

m1.add(i1);

m1.add(i2);

m2.add(i3);

m2.addSeparator();

m2.add(i4);

m3.add(i5);

m3.add(mm);

mb.add(m1);

mb.add(m2);

mb.add(m3);

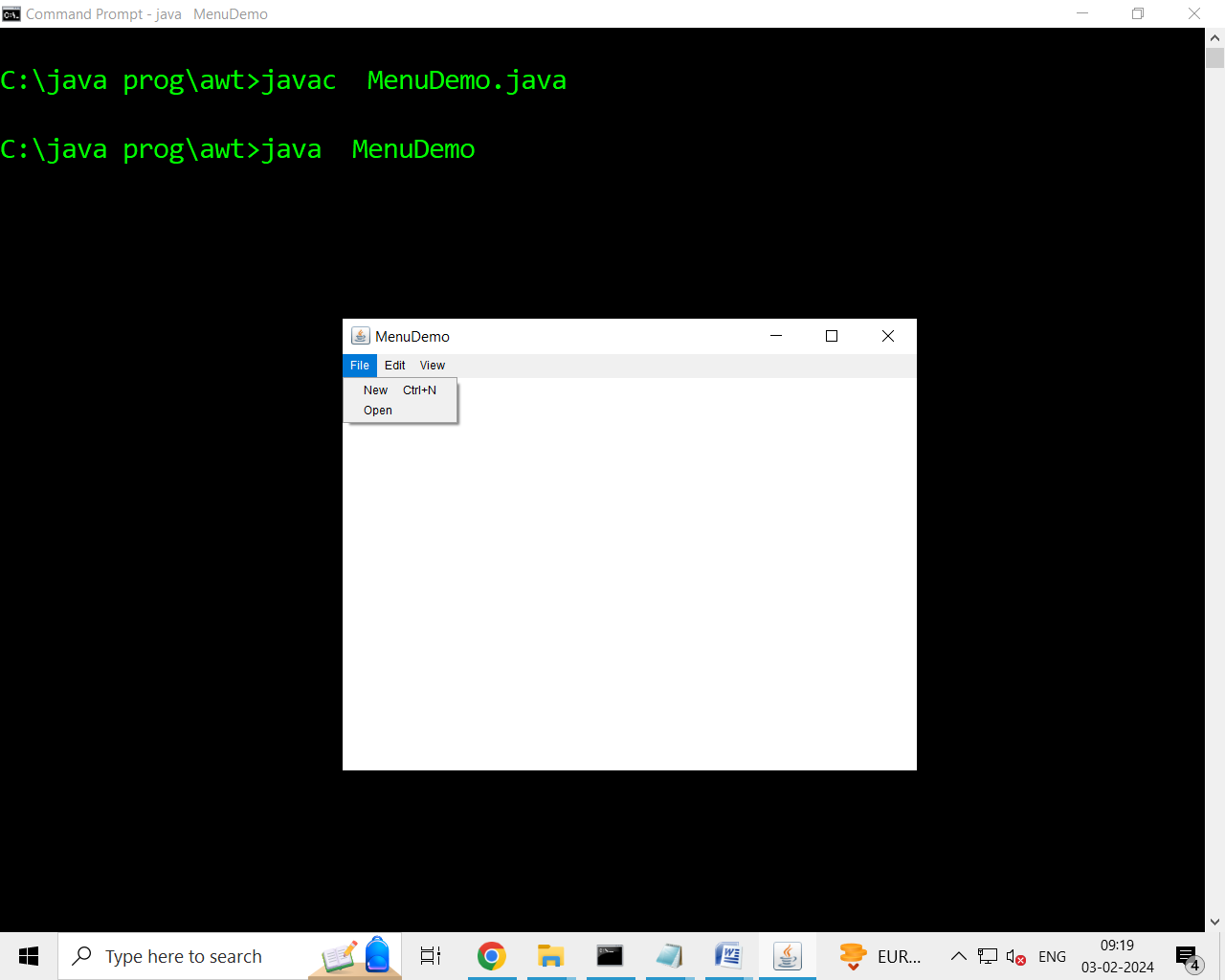
}

public static void main(String args[])

{

new MenuDemo();

}}

****

**Anonymous class**

An Anonymous class is one that is not assigned a name. we can use anonymous inner class can be used in writing the event handlers.

import java.awt.\*;

import java.awt.event.\*;

class annon extends Frame

{

annon()

{

setSize(300,300);

setVisible(true);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent we)

{

System.exit(0);

}

});

}

public static void main(String args[])

{

new annon();

}

}

The syntax new MouseAdapter(){..} indicates to compiler that the code between the braces defines an anonymous inner class. Furthermore the class extends MouseAdapter. This new class not named, but it is automatically instantiated when this expression is executed.

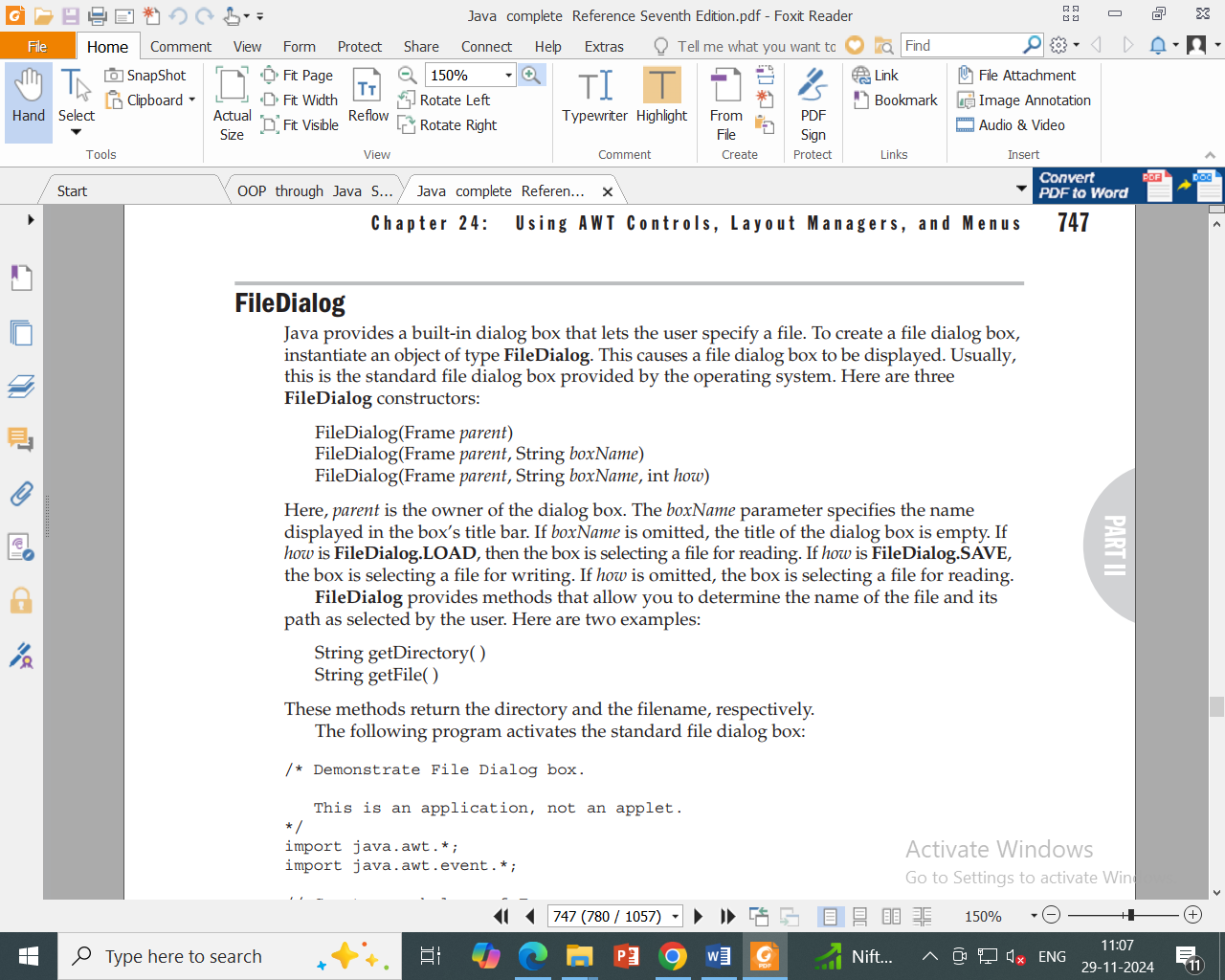
**Dialogs**

Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window. Dialog boxes don’t have menu bars, but in other respects, they function like frame windows. Dialog boxes may be modal or modeless. When a m*odal* dialog box is active, all input is directed to it until it is closed. This means that you cannot access other parts of your program until you have closed the dialog box. When a *modeless* dialog box is active, input focus can be directed to another window in your program.

Two commonly used constructors are shown here:

Dialog(Frame *parentWindow*, boolean *mode*)  
Dialog(Frame *parentWindow*, String *title*, boolean *mode*)

Here, *parentWindow* is the owner of the dialog box. If *mode* is **true**, the dialog box is modal. Otherwise, it is modeless. The title of the dialog box can be passed in *title.* Generally, you will subclass **Dialog**, adding the functionality required by your application.



**Layout Managers**

**Layout Manager**:-A Layout manager is a class that is useful to arrange components in a particular manner in a frame or container.

The following classes represent the layout managers in java.

1. FlowLayout.
2. BorderLayout.
3. CardLayout.
4. GridLayout

**FlowLayout**

FlowLayout is useful to arrange the components in a line one after the other. When a line is filled with components, they are automatically placed in the next line. This is the default layout in applets and panels.

To create FlowLayout, we can the following ways.

1. **FlowLayout f=new FlowLayout();**

This creates FlowLayout. By default, the gap between components will be 5 pixels and the components are centered in the first line.

1. **FlowLayout f=new FlowLayout(int alignment);**

Here, the alignment of components can be specified. To arrange the components starting from left to right, we can use ***FlowLayout.LEFT*** . to adjust the components towards right, we can use ***FlowLayout.RIGHT*** and for center alignment, we can use ***FlowLayout.CENTER .***

1. **FlowLayout f=new FlowLayout(int alignment, int hgap, int vgap);**

Here, the hgap specify the space between components. hgap represents horizontal gap and vgap represents vertical gap in pixels.

**/\* Demo on FlowLayout\*/**

import java.awt.\*;

import java.awt.event.\*;

class FrameLayoutDemo extends Frame

{

Button b1,b2,b3,b4,b5;

FrameLayoutDemo()

{

setTitle("FrameLayout Demo");

setSize(300,300);

setVisible(true);

b1=new Button("1");

b2=new Button("2");

b3=new Button("3");

b4=new Button("4");

b5=new Button("5");

FlowLayout f= new FlowLayout();

setLayout(f);

add(b1);

add(b2);

add(b3);

add(b4);

add(b5);

}

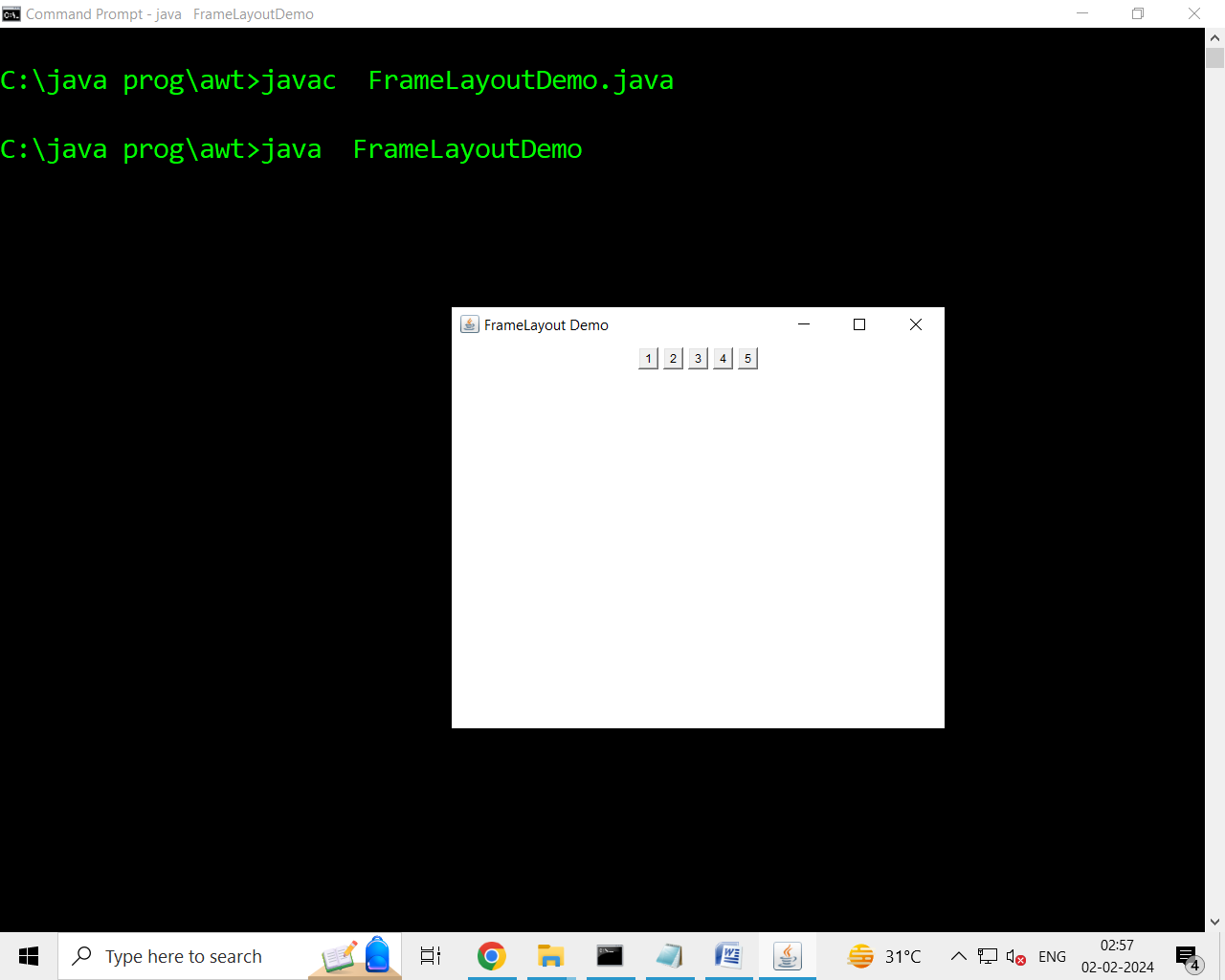
public static void main(String args[])

{

new FrameLayoutDemo();

}

}



**BorderLayout**

BorderLayout is useful to arrange the components in the 4 borders of the frame as well as in the center. The border are identical with the names of directions. The top border is specified as ‘North’, the right side border ‘East’, the bottom one as ‘South’, and the left one as ‘West’. The center is represented as ’Center’.

To create the BorderLayout we can use the following ways.

1. **BorderLayout b=new BorderLayout();**

This creates a BorderLayout without ant gaps between the components.

1. **BorderLayout b=new BorderLayout(int hgap, int vgap);**

Here, the hgap specify the space between components. hgap represents horizontal gap and vgap represents vertical gap in pixels.While adding the components to the container the direction should be specified as

***C.add(“North”,component);*** //C is container.

Here the component is added in the container in North direction.We can also add the componenet in North direction as shown here

***C.add(component, BorderLayout.NORTH);***

|  |  |  |
| --- | --- | --- |
|  | **NORTH** |  |
| **WEST** | **CENTER** | **EAST** |
|  | **SOUTH** |  |

***Figure 1:*** Direction of components in BorderLayout.

**/\* Demo program on BorderLayout\*/**

import java.awt.\*;

import java.awt.event.\*;

class BorderLayoutDemo extends Frame

{

Button b1,b2,b3,b4,b5;

BorderLayoutDemo()

{

setTitle("BordeLayout Demo");

setSize(300,300);

setVisible(true);

b1=new Button("1");

b2=new Button("2");

b3=new Button("3");

b4=new Button("4");

b5=new Button("5");

//FlowLayout f= new FlowLayout();

//setLayout(f);

add(b1, BorderLayout.NORTH);

add(b2, BorderLayout.SOUTH);

add(b3, BorderLayout.EAST);

add(b4, BorderLayout.WEST);

add(b5, BorderLayout.CENTER);

}

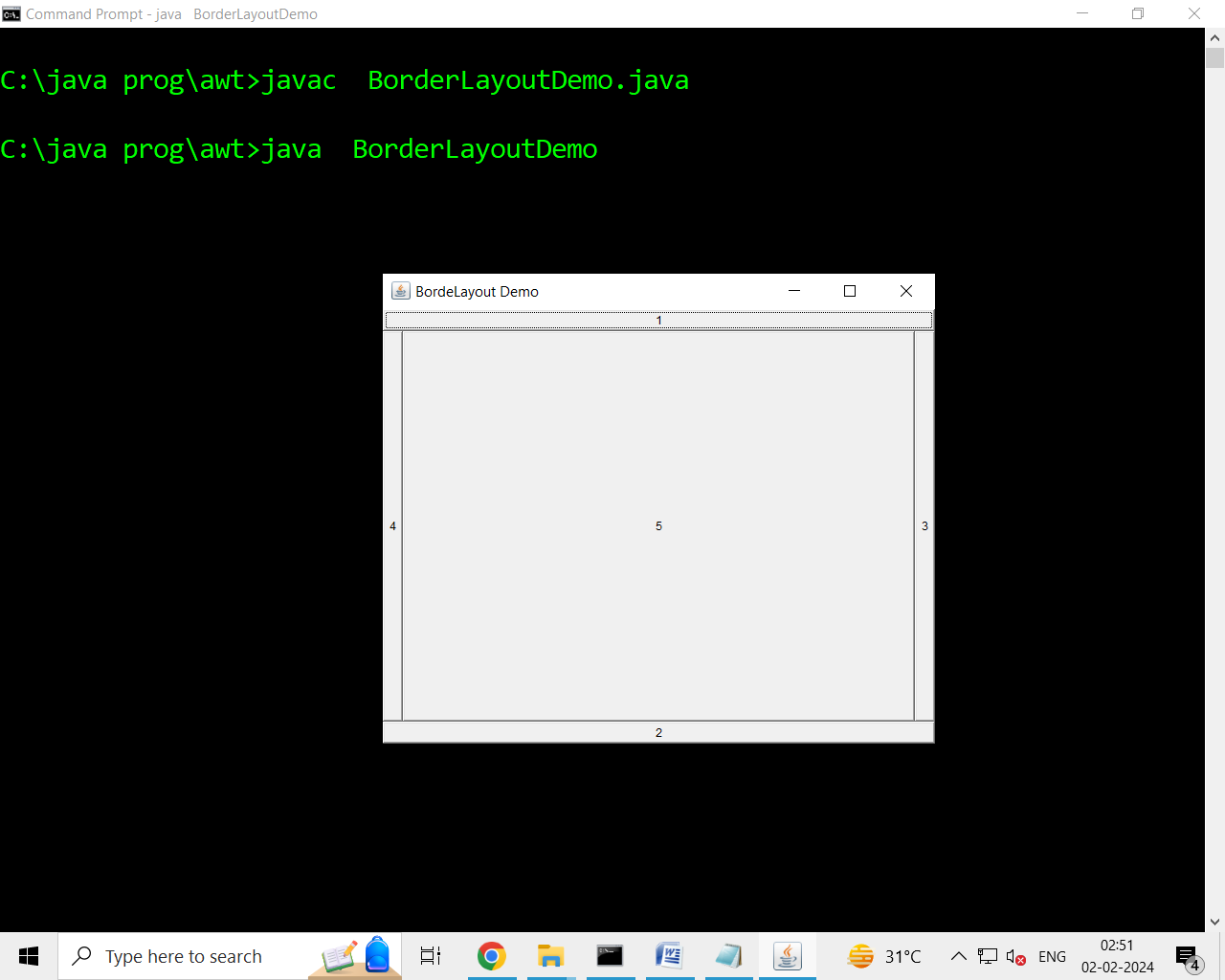
public static void main(String args[])

{

new BorderLayoutDemo();

}

}



**CardLayout**

A CardLayout object is a layout manager which treats each component as a card. Only one card is visible at a time, and the container acts as a stack of cards. The first component added to a CardLayout object is the visible component when the container is first displayed.

To create CardLayout object, we can use the following ways

1. **CardLayout c=new CardLayout();**

Here, the CardLayout object is created without any gaps between the components.

1. **CardLayout c=new CardLayout(int hgap, int vgap);**

The proceeding statement creates a card layout with the specified horizontal and vertical gaps between the components.

While adding the components to the container, we can use add() method as:

Eg: C.add(“card\_name”, component);

To retrieve the cards one by one , the following methods can be used.

1. void first(container); to retrieve the first card.
2. void last(container); to retrieve the last card.
3. void next(container); to retrieve the next card.
4. void previous(container); to retrieve the previous card.
5. void show(container,”card\_name”); to see a particular card with name specified.

**GridLayout**

GridLayout is useful to divide the container into a two-dimensional grid form that contains several rows and columns. The container is divided into equal-sized rectangles, and one component is placed in each rectangle.

To create GridLayout

Object we can write as:

1. **GridLayout g=new GridLayout();**

This creates a Grid Layout with a default of one column per component, in a single row.

1. **GridLayout g=new new GridLayout (int rows, int cols);**

This creates a Grid Layout with specified number of rows and columns.

1. **GridLayout g=new GridLayout (int rows, int cols, int hgap, int vgap);**

Here, the hgap represents horizontal gap between component and vgap represents vertical gap between components.

**/\* Demo on GridLayout\*/**

import java.awt.\*;

class GridLayoutDemo extends Frame

{

Button b1,b2,b3,b4;

GridLayoutDemo()

{

setSize(300,300);

setVisible(true);

setTitle("GridLayout Demo");

GridLayout g1=new GridLayout(2,2);

setLayout(g1);

b1=new Button("1");

b2=new Button("2");

b3=new Button("3");

b4=new Button("4");

add(b1);

add(b2);

add(b3);

add(b4);

}

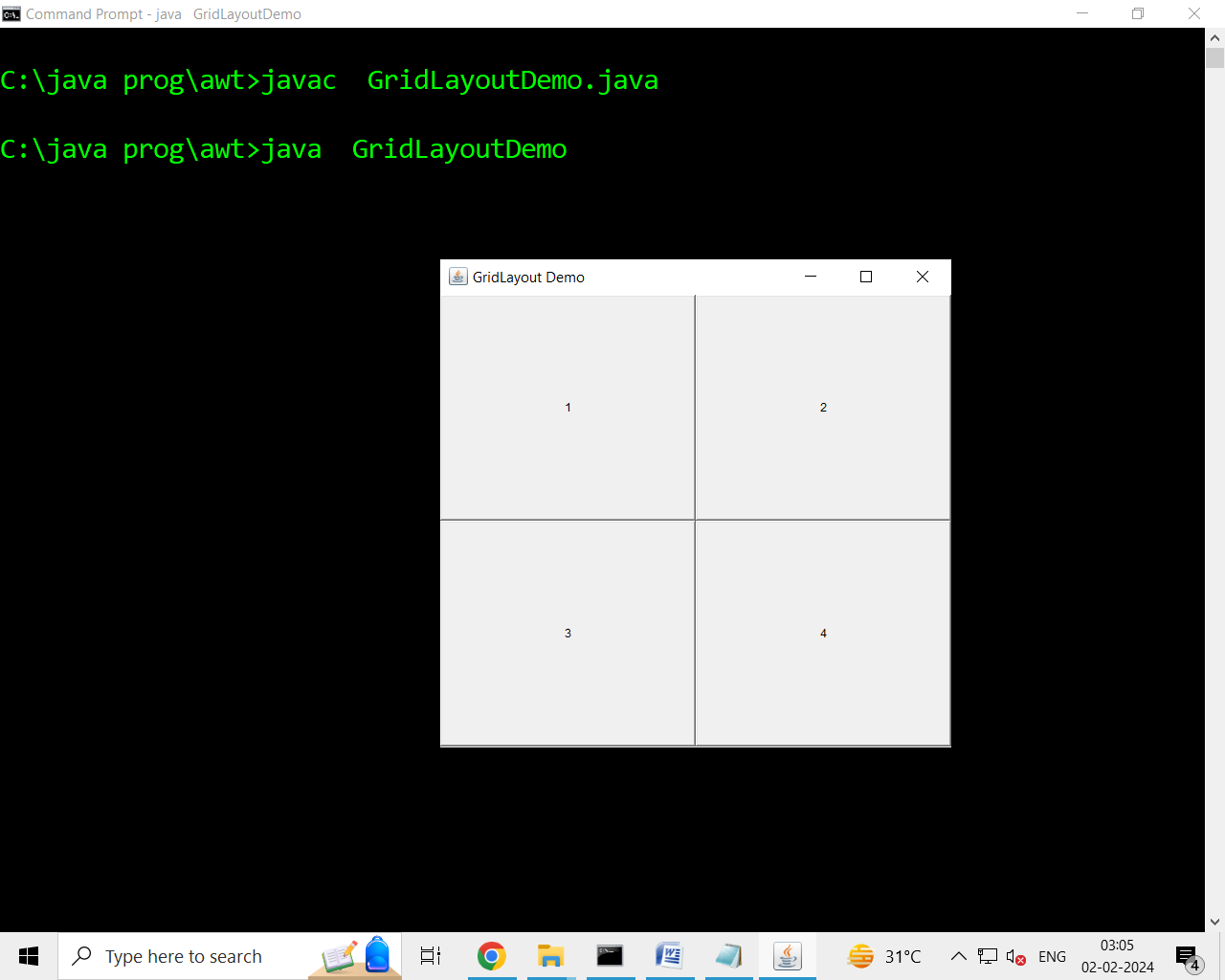
public static void main(String args[])

{

new GridLayoutDemo();

}

}



**GridBagLayout**

We can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns. This is why the layout is called a *grid bag.* It’s a collection of small grids joined together. The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

**GridBagLayout** defines only one constructor, which is shown here:

GridBagLayout( );

**GridBagLayout** defines several methods, of which many are protected and not for general use. There is one method, however, that you must use: **setConstraints( )**.

It is shown here:

void setConstraints(Component *comp*, GridBagConstraints *cons*)

Here, *comp* is the component for which the constraints specified by *cons* apply. This method sets the constraints that apply to each component in the grid bag.